MQTT / VDA 5050 Integration Guide

Send orders to the cutting machines and follow them as VDA 5050 vehicles over MQTT

Interface	MQTT / VDA 5050
Document version	1.1
Date	October 2026

Contents

1. Introduction 4
1.1 What the MQTT interface is 4
1.2 How it works 4
1.3 What you need from the Link administrator 5
1.4 Conventions in this guide 5
2. Quick start 6
3. Topics and connection 8
3.1 Topic structure 8
3.2 Identity and connections 8
3.3 The factsheet 8
4. Orders 10
4.1 Structure 10
4.2 The parameters of a cut action 10
4.3 Updating and repeating an order 10
4.4 Which machine, and when 11
5. The state 12
5.1 Structure 12
5.2 Action status 13
5.3 Errors in the state 13
5.4 Live data 14
6. Instant actions 15
6.1 cancelOrder 15
6.2 Cable diameter 15
7. Security and operation 17
8. Debugging and troubleshooting 18
8.1 A method: follow the order step by step 18
8.2 Where the Link leaves its traces 18
8.3 My line does not start 19
8.4 Common problems with this interface 20
8.5 Tracing the traffic 20
8.6 What to send to LOIMEX support 20
9. Testing without a machine 22
9.1 Set up the test machine 22
9.2 Useful simulator options 23
10. Going to production 24

Appendix A. Example client (Python) 25
Appendix B. Full factsheet and state 27

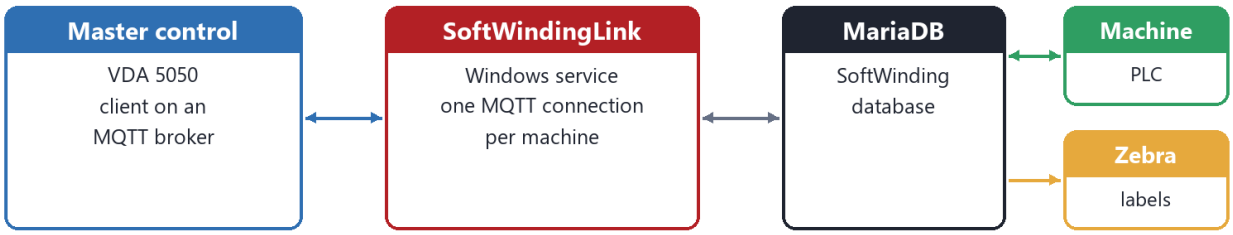
1. Introduction

1.1 What the MQTT interface is

The **MQTT interface** lets a logistics **master control** treat every cutting machine as a **VDA 5050 vehicle**. VDA 5050 is the standard JSON interface between a master control and automated vehicles, carried over MQTT. It is the natural choice if your systems already work with that standard, or run on a message broker such as **AWS IoT Core**, Mosquitto, EMQX or HiveMQ.

- **Each machine is a vehicle** with its own MQTT connection, identity and topics (`serialNumber` = the machine identifier).
- **You publish orders** (VDA `order`) whose nodes carry a `cut` action, and **instant actions** (cancel, cable diameter, state request).
- **The machine publishes its `state`** continuously: what each action is doing (`WAITING`, `RUNNING`, `FINISHED`...), the metres measured, errors, and live data. It also publishes `connection` and a `factsheet`.
- Behind the messages the rules are exactly those of the other interfaces: the order becomes lines for the machine, the operator cuts, labels are printed.

1.2 How it works



1. Your master control publishes an order to <prefix>/M1/order, with one cut action per node.
2. The Link validates it and creates the lines for machine M1. It publishes the machine state.
3. In the state, each action goes WAITING, INITIALIZING, RUNNING and then FINISHED (or FAILED).
4. When an action is FINISHED, the state carries the result: metres measured and cuts.
5. You read the state (QoS 0) and the connection (retained) of each machine.

Figure 1. The path of an order through MQTT / VDA 5050. The broker sits between the two.

> **Warning. It is asynchronous and event driven.** Publishing an order says nothing about the result. Watch the `state` of the machine: the status of each action tells you what happens. A state is published at every change, and at least every few seconds.

1.3 What you need from the Link administrator

Table 1. Data to ask the administrator for.

You need	Example	Where it is set
The MQTT broker	`broker.example.com:8883`	Both the Link and your master control connect **to the same broker**: `[Mqtt] Host`, `Port` (8883) in `SoftWindingLink.ini`. The channel is **off by default**: `[Mqtt] Enabled=1`, then restart the service.
TLS and credentials	CA, client certificate and key	`[Mqtt] Tls`, `CAFile`, `CertFile`, `KeyFile` (AWS IoT Core requires a client certificate) or `User` / `Password`. You use your own credentials on the broker.
The topic prefix	`uagv/v2/LOIMEX`	`[Mqtt] InterfaceName` (`uagv`), `MajorVersion` (`v2`), `Manufacturer` (`LOIMEX`).
A client with the MQTT channel	`MQTT`	Administration window, **Clients** tab (`[Mqtt] ClientID`, default `MQTT`). The orders of this channel belong to that client; its `AllowedMachines` apply.
Machine identifiers	`M1`, `M2`	The `serialNumber` of each vehicle. Only `A-Z a-z 0-9 _ . : -` . In the topics: `<prefix>/M1/...`.
`libssl-3-x64.dll`, `libcrypto-3-x64.dll`		OpenSSL next to the service (they come with the installer) when you use TLS.

1.4 Conventions in this guide

- The messages shown are real, captured with `docs\examples\mqtt\vda_client.py` (Python, `paho-mqtt`) on a test system with two simulated machines (`M1`, `M2`) and a local broker, with the prefix `uagv/v2/LOIMEX`.
- JSON is shown formatted; on the wire it is compact. `headerId` and `timestamp` change in every message.
- VDA 5050 **version 2.0.0** over **MQTT 3.1.1** is implemented. Not in this version: `visualization`, VDA 2.1 restrictions and actions other than `cut` in nodes.

2. Quick start

A complete cycle in a few minutes. Without your own broker, start the little test broker (`pip install amqtt`, then `python mini_broker.py --port=1883`) and point the Link at it.

```
pip install paho-mqtt
set MQTT_HOST=broker.example.com
set MQTT_PORT=8883
set MQTT_TOPIC_PREFIX=uagv/v2/LOIMEX
set LINK_MACHINE=M1
python vda_client.py
```

(Without TLS, for a test broker: `set MQTT_TLS=0`. With a client certificate set `MQTT_CA`, `MQTT_CERT` and `MQTT_KEY`.)

2.1 Step 1. The machine announces itself

As soon as the Link connects, each machine publishes a **retained** `connection` message (`ONLINE`), its `factsheet` and a first `state`. Subscribe to `uagv/v2/LOIMEX/+/connection` and you know which machines exist and are alive:

```
topic: uagv/v2/LOIMEX/M1/connection   (QoS 1, retained)
```

```
{
  "headerId": 0,
  "timestamp": "2026-10-28T12:30:41.327Z",
  "version": "2.0.0",
  "manufacturer": "LOIMEX",
  "serialNumber": "M1",
  "connectionState": "ONLINE"
}
```

2.2 Step 2. Publish an order

Topic `uagv/v2/LOIMEX/M1/order`. Two nodes, one `cut` action each (item 10: one cut of 3 m; item 20: two cuts of 2 m):

```
{
  "headerId": 1,
  "timestamp": "2026-10-28T12:32:42.000Z",
  "version": "2.0.0",
  "manufacturer": "LOIMEX",
  "serialNumber": "M1",
  "orderId": "MQ-5001",
  "orderUpdateId": 0,
  "edges": [],
  "nodes": [
    {
      "nodeId": "N1",
      "sequenceId": 0,
      "released": true,
      "actions": [
        {
          "actionType": "cut",
          "actionId": "MQ-5001-1",
          "blockingType": "HARD",
          "actionParameters": [
            {"key": "item", "value": 10},
            {"key": "productId", "value": "H07V-K-1.5-BK"},
            {"key": "lengthM", "value": 3.0},
            {"key": "cuts", "value": 1},
            {"key": "reelId", "value": "R-77881"},
            {"key": "description", "value": "H07V-K 1.5 mm2 black"},
            {"key": "customer", "value": "ACME Cables Ltd"}
          ]
```

```
        }
      ]
    },
    "... (second node like the first, actionId MQ-5001-2)"
  ]
}
```

2.3 Step 3. Follow the state

Topic `uagv/v2/LOIMEX/M1/state`. Reading `actionStates` over time:

Table 2. The state while the order is executed.

Message	Status of the actions (first / second)	lastNodeId
1	WAITING / WAITING	-
2	RUNNING / WAITING	-
3	FINISHED / WAITING	N1
4	FINISHED / RUNNING	N1
5	FINISHED / FINISHED	N2

Once the first action is `FINISHED` the state carries its result:

```
{
  "orderId": "MQ-5001",
  "lastNodeId": "N2",
  "nodeStates": [],
  "actionStates": [
    {
      "actionId": "MQ-5001-1",
      "actionType": "cut",
      "actionStatus": "FINISHED",
      "resultDescription": "measured 3.000 m, 1 cut(s)"
    },
    {
      "actionId": "MQ-5001-2",
      "actionType": "cut",
      "actionStatus": "FINISHED",
      "resultDescription": "measured 4.000 m, 2 cut(s)"
    }
  ],
  "information": [
    {
      "infoType": "cutResult",
      "infoReferences": [
        {"referenceKey": "orderId", "referenceValue": "MQ-5001"},
        {"referenceKey": "item", "referenceValue": "10"},
        {"referenceKey": "actionId", "referenceValue": "MQ-5001-1"},
        {"referenceKey": "measuredM", "referenceValue": "3.000"},
        {"referenceKey": "cuts", "referenceValue": "1"}
      ],
      "infoDescription": "measured 3.000 m, 1 cut(s)",
      "infoLevel": "INFO"
    },
    {
      "infoType": "cutResult",
      "infoReferences": [
        {"referenceKey": "orderId", "referenceValue": "MQ-5001"},
        {"referenceKey": "item", "referenceValue": "20"},
        {"referenceKey": "actionId", "referenceValue": "MQ-5001-2"},
        {"referenceKey": "measuredM", "referenceValue": "4.000"},
        {"referenceKey": "cuts", "referenceValue": "2"}
      ],
      "infoDescription": "measured 4.000 m, 2 cut(s)",
      "infoLevel": "INFO"
    }
  ]
}
```

`cutResult` carries what the machine really measured: 3.000 m in 1 cut for item 10 and 4.000 m in 2 cuts for item 20 (2 cuts of 2 m).

3. Topics and connection

3.1 Topic structure

```
<InterfaceName>/<MajorVersion>/<Manufacturer>/<serialNumber>/<topic>
uagv/v2/LOIMEX/M1/order
```

Table 3. Topics of each machine.

Topic	Direction	QoS	Content
`order`	you -> Link	0 or 1	A VDA order: nodes with `cut` actions.
`instantActions`	you -> Link	0 or 1	`cancelOrder`, `setCableDiameter`, `cableLaser`, `stateRequest`, `factsheetRequest`. Any other type ends `FAILED`.
`state`	Link -> you	0	The machine state, at every change (at least 250 ms apart) and every `[Mqtt] StateSeconds` (30 s by default; 5 s on the test system).
`connection`	Link -> you	1, retained	`ONLINE` when connected, `OFFLINE` when the Link stops, `CONNECTIONBROKEN` (the will message) if the Link dies.
`factsheet`	Link -> you	0, retained	Description of the machine and of the `cut` action. Published at connection and on `factsheetRequest`.

Subscribe with wildcards to see all the machines: `uagv/v2/LOIMEX/+/state`. Publish to the topic of one machine.

3.2 Identity and connections

- Every machine of the Link (enabled in the administration window) has its **own MQTT connection**. Its MQTT client id is `[Mqtt] ClientIdPrefix` + the machine id.
- **Your own connection** needs a different client id (for example `erp-master`) and rights on the topics above.
- The Link reconnects by itself with a growing delay (1 s to 30 s); on reconnect it subscribes again and publishes `ONLINE`, the `factsheet` and the `state`. QoS 1 messages are queued while it is disconnected (up to 500); QoS 0 are not.
- `KeepAlive` is 30 s by default (AWS IoT Core accepts 30 to 1200 s).

3.3 The factsheet

The factsheet tells your master control what the machine can do. The parts that matter (full message in the captured file):

```
{
  "serialNumber": "M1",
  "typeSpecification": {"seriesName": "SoftWindingLink cutting machine"},
  "protocolLimits": {
    "timing": {"minOrderInterval": 1.0, "minStateInterval": 0.25, "defaultStateInterval": 5.0},
    "maxArrayLens": {
      "order.nodes": 100,
      "node.actions": 1,
      "actions.actionsParameters": 20,
      "instantActions": 10
    }
  },
  "protocolFeatures": {
    "agvActions": [
```

```
      "cut",
      "cancelOrder",
      "setCableDiameter",
      "cableLaser",
      "stateRequest",
      "factsheetRequest"
    ]
  }
}
```

The limits that apply to you: at most **100 nodes** per order, **one action per node**, 20 parameters per action, 10 instant actions per message, and **one order per second** (`minOrderInterval`).

4. Orders

4.1 Structure

A standard VDA 5050 `order` (header, `orderId`, `orderUpdateId`, `nodes`, `edges`). The Link uses **only the nodes with a `cut` action**; edges and other actions are ignored. Each `cut` action is **one order line** for the machine of the topic:

Table 4. How an order becomes lines.

VDA element	Becomes
`orderId`	The order number (up to 30 characters, printable ASCII without spaces).
`orderUpdateId`	Version of the order (section 4.3).
nodes, in `sequenceId` order	The lines, numbered `item` 10, 20, 30... unless an `item` parameter is given.
`actions[0]` with `actionType = "cut"`	One line. The parameters are the data of the line.
`actionId`	Returned in `actionStates` so you can follow the line. Must be unique.

4.2 The parameters of a cut action

`actionParameters` is a list of `{"key": ..., "value": ...}`. The value is a number or a text (a decimal point or a comma is accepted):

Table 5. Parameters of the cut action.

key	Required	Meaning
`productId`	yes	Product (cable) code.
`lengthM`	yes	Length of **one** cut, in metres (> 0).
`cuts`	no (1)	Number of cuts of that length.
`item`	no	Line number. By default 10, 20, 30 in order of `sequenceId`. Two actions with the same item reject the order.
`reelId`	no	Reel (label and traceability).
`description`	no	Text for the label and the screen.
`customer`	no	Customer name.
`batchId`	no	Work package (default `MQTT_<date>`).
`priority`	no	Lower goes first.
`notes`, `free1` ... `free10`	no	Texts for the label template.
`cableDiameterMm`	no	Cable diameter, 0 to 1000 mm: sent to the machine when the line is prepared (manual mode).

4.3 Updating and repeating an order

- `orderUpdateId` **equal or higher** than the last one received for that `orderId`: the order is processed again, and it is idempotent: lines that exist and do not change are left alone, lines still pending are updated.
- `orderUpdateId` **lower** than the last received: **ignored**, and the state shows the error `orderUpdateError`.
- The same rules as the other interfaces apply to a line already taken by a machine (it cannot change).

4.4 Which machine, and when

The topic names the machine: an order published to `.../M1/order` is for M1. Each machine takes one line at a time (by priority, then item). The operator starts the machine: an action stays `INITIALIZING` while the line is prepared and nobody starts it.

5. The state

5.1 Structure


```
{
  "headerId": 28,
  "timestamp": "2026-10-28T12:32:41.913Z",
  "version": "2.0.0",
  "manufacturer": "LOIMEX",
  "serialNumber": "M1",
  "orderId": "MQ-1793190691",
  "orderUpdateId": 0,
  "lastNodeId": "N2",
  "nodeStates": [],
  "driving": false,
  "operatingMode": "AUTOMATIC",
  "actionStates": [
    {
      "actionId": "MQ-1793190691-1",
      "actionType": "cut",
      "actionStatus": "FINISHED",
      "resultDescription": "measured 3.000 m, 1 cut(s)"
    },
    {
      "actionId": "MQ-1793190691-2",
      "actionType": "cut",
      "actionStatus": "FINISHED",
      "resultDescription": "measured 4.000 m, 2 cut(s)"
    }
  ],
  "errors": [],
  "safetyState": {"eStop": "NONE", "fieldViolation": false}
}
```


Table 6. Fields of the state.

Field	Meaning for a cutting machine
`orderId`, `orderUpdateId`	The last order **received and accepted** from this machine's topic. A rejected message does not change it.
`actionStates`	One entry per `cut` action and per instant action received: `actionId`, `actionType`, `actionStatus`, `resultDescription`.
`lastNodeId`, `lastNodeSequenceId`	The node with the highest `sequenceId` whose actions are all `FINISHED`.
`nodeStates`	The nodes that still have an unfinished action.
`driving`	`true` while the machine is measuring (cable running).
`velocity.vx`	Working speed in m/s (m/min divided by 60). Neutral values for movement fields (`batteryState` 100 %, no `agvPosition`).
`errors`	Problems (section 5.3). **Warning:** the list can still contain the error of the previous message; match it by its text and time.
`information`	Extra data: `machineStatus`, one `cutResult` per finished line, `processTime`, `cableDiameter`.
`safetyState.eStop`	`NONE`, or `MANUAL` with the emergency stop pressed.

5.2 Action status

Table 7. Status of a cut action.

actionStatus	Line state	resultDescription
`WAITING`	Stored, waiting for the machine (pending).	
`INITIALIZING`	Taken by the machine and being prepared; the machine is not measuring yet.	
`RUNNING`	The machine is measuring or waiting for confirmation.	
`FINISHED`	Done.	`measured 3.000 m, 1 cut(s)`: the metres and cuts of the line (also in `cutResult`).
`FAILED`	Rejected, error or canceled.	The reason and the Link code: `invalid field (4004)`.

For an **instant action**, `FINISHED` or `FAILED` come at once, with the result in `resultDescription`.

5.3 Errors in the state

Table 8. Error types.

errorType	When	Example errorDescription
`orderError`	The order message was invalid or rejected as a whole.	`the message is not valid JSON`, `the order has no "cut" action`, `two cut actions have the same item 10`
`lineError`	A line was rejected or could not be cut (Link code in the text).	`invalid field (4004)`, `unknown machine (4001)`
`orderUpdateError`	An older `orderUpdateId`.	`orderUpdateId 2 is lower than the last received (5)`
`noOrderToCancel`	`cancelOrder` with nothing to cancel.	`there is no pending line in order MQ-E5`
`machineRunning`	`cancelOrder` while the machine is running.	`the machine is running: stop it before canceling`
`emergencyStop`	Emergency stop (level FATAL; `safetyState.eStop = MANUAL`).	
`machineError`, `plcOffline`	Machine error / the PLC does not answer.	

Examples (the part of the state that matters, as captured):

```
{
  "1. invalid JSON": [["orderError", "the message is not valid JSON"]],
  "2. no cut action": [["orderError", "the order has no \"cut\" action"]],
  "3. length 0": {
    "actionStates": [["MQ-E3-1", "FAILED", "invalid field (4004)"]],
    "errors": [["lineError", "invalid field (4004)"]]
  },
  "4. duplicate item": [
    ["lineError", "invalid field (4004)"],
    ["orderError", "two cut actions have the same item 10"]
  ],
  "5. old orderUpdateId": [
    ["orderUpdateError", "orderUpdateId 2 is lower than the last received (5)"]
  ]
}
```

5.4 Live data

`information` also carries (when the machine is online): `machineStatus` (`counterM`, `cutNr`, `state`), `processTime` (`mm:ss` plus `seconds` and `speedMPerMin`) and `cableDiameter` (`diameterMm` and `mode`: `laser`, `manual` or `unknown`). They are refreshed with the next state message, so they may lag a few seconds behind an instant action.


```
{
  "information": [
    {
      "infoType": "machineStatus",
      "infoReferences": [
        {"referenceKey": "counterM", "referenceValue": "0.000"},
        {"referenceKey": "cutNr", "referenceValue": "0"},
        {"referenceKey": "state", "referenceValue": "I"}
      ],
      "infoDescription": "ready",
      "infoLevel": "INFO"
    },
    {
      "infoType": "processTime",
      "infoReferences": [
        {"referenceKey": "seconds", "referenceValue": "1"},
        {"referenceKey": "speedMPerMin", "referenceValue": "0.000"}
      ],
      "infoDescription": "00:01",
      "infoLevel": "INFO"
    },
    {
      "infoType": "cableDiameter",
      "infoReferences": [
        {"referenceKey": "diameterMm", "referenceValue": "8.500"},
        {"referenceKey": "mode", "referenceValue": "manual"}
      ],
      "infoDescription": "8.500 mm, manual",
      "infoLevel": "INFO"
    }
  ]
}
```

6. Instant actions

Published to `.../<machine>/instantActions`. Each action appears in `actionStates` with its result:

```
{"headerId": 1, "timestamp": "...", "version": "2.0.0", "manufacturer": "LOIMEX", "serialNumber": "M1",
 "instantActions": [{"actionType": "cancelOrder", "actionId": "cancel-1", "blockingType": "NONE"}]}
```

Table 9. Instant actions.

actionType	Parameters	Effect
`cancelOrder`	none	Removes **what the machine has on its screen**, then cancels the other pending lines of the current order. See below.
`setCableDiameter`	`diameterMm` (0 to 1000)	Sends the diameter to the machine and switches to manual mode.
`cableLaser`	none	Back to the laser gauge.
`stateRequest`	none	Publishes a state now.
`factsheetRequest`	none	Publishes the factsheet again.
anything else		`FAILED`, `instant action not supported`.

6.1 cancelOrder

Table 10. cancelOrder.

Situation	Result
Pending lines of the current order	`FINISHED`, `N line(s) canceled`.
The machine is running	`FAILED`, `the machine is running`, and the error `machineRunning`.
Nothing to cancel	`FAILED`, `there is no order to cancel`, error `noOrderToCancel`.

```
{
  "actionStates": [
    ["MQ-6001-1", "WAITING", ""],
    ["MQ-6001-2", "WAITING", ""],
    ["cancel-1", "FAILED", "the machine is running"]
  ],
  "errors": [
    {
      "errorType": "machineRunning",
      "errorDescription": "the machine is running: stop it before canceling",
      "errorLevel": "WARNING"
    }
  ]
}
```

It works even if there is no current order. If the machine does not answer, only the pending lines are canceled.

6.2 Cable diameter

`setCableDiameter` with `diameterMm` 10.5; the result is in the action state at once, and the live data follow:

```
{
  "actionStates": [
    {
      "actionId": "cab-1",
      "actionType": "setCableDiameter",
```

```
      "actionStatus": "FINISHED",
      "resultDescription": "laser=N;mm=10.500;raw=105"
    },
    {
      "actionId": "cab-2",
      "actionType": "cableLaser",
      "actionStatus": "FINISHED",
      "resultDescription": "laser=Y;mm=12.500;raw=125"
    }
  ]
}
```

`laser=N;mm=10.500` means manual mode with 10.5 mm; `laser=Y;mm=12.500` means the laser gauge is back and reads 12.5 mm. A `cut` action can also carry `cableDiameterMm`.

7. Security and operation

Table 11. Security and connection.

Topic	How it works
TLS	TLS 1.2 or later, with OpenSSL 3 loaded by the service (`libssl-3-x64.dll`, `libcrypto-3-x64.dll` next to the program). `[Mqtt] Tls=1`, `Verify=1` (check the broker certificate), `CAFile`.
Client certificate	`CertFile` and `KeyFile` (+ `KeyPassword`). AWS IoT Core requires it. The Link says in the log if OpenSSL is missing or the broker rejects the certificate, and keeps retrying.
Credentials	`User` and `Password` (stored encrypted after the first start).
Who may do what	The broker decides who may publish and subscribe. Give your master control rights on `.../+/order`, `.../+/instantActions` and read rights on `state`, `connection` and `factsheet`.
Client of the orders	All the orders of this channel belong to the client `[Mqtt] ClientID`. Its `AllowedMachines` limit the machines.
Last will	If the Link dies, the broker publishes `CONNECTIONBROKEN` on each machine's `connection` topic. Watch it.

Settings in `SoftWindingLink.ini`, section `[Mqtt]`: `Enabled`, `Host`, `Port` (8883), `Tls` (1), `Verify` (1), `CAFile`, `CertFile`, `KeyFile`, `KeyPassword`, `User`, `Password`, `ClientIdPrefix`, `InterfaceName` (`uagv`), `MajorVersion` (`v2`), `Manufacturer` (`LOIMEX`), `ClientID` (`MQTT`), `KeepAlive` (30), `StateSeconds` (30).

8. Debugging and troubleshooting

8.1 A method: follow the order step by step

When "nothing happens", do not guess: follow the order along its path, from your software to the machine, and find the first step where it stops.

Table 12. The five steps of a diagnosis.

Step	Question	Where to look
1	**Did the Link read my order?** The machine publishes its `state`: `actionStates` per action, and `errors` (`orderError`, `lineError`...).	The `state` and `connection` messages on the broker and the `MQTT` lines of the Link log.
2	**Is the order stored?**	Administration window, **Orders** tab (filter by order number), or the view `sw_link_v_orders` in the database.
3	**Is a machine working on it?**	Administration window, **Machines** tab: state and current order. In the database: the view `sw_link_v_machines`.
4	**What did the Link do?**	Administration window, **Log** and **Events** tabs; the daily log file `logs\link_YYYYMMDD.log`.
5	**What did the machine answer?**	The log lines tagged with the machine, for example `[M1] ...`, and the machine state text.

8.2 Where the Link leaves its traces

Table 13. Diagnostic tools.

Place	What it shows
Administration window	**Status**: engine, database and licence. **Machines**: state, counter and current line of each machine, with a connection test. **Orders**: every line with its status, searchable. **Clients**: your client, its channels, allowed IPs and machines. **Events**: the events your system receives. **Log**: the Link log with a level filter and a button that opens the log folder.
Log file `logs\link_YYYYMMDD.log`	One line per fact: starts, connections, every request of the channel, every order accepted, every line sent to the PLC, every cut confirmed, warnings and errors. One file per day, kept 30 days.
Table `sw_link_log`	The engine part of the log, queryable with SQL: `SELECT * FROM sw_link_log ORDER BY LogID DESC LIMIT 50`.
Events (`sw_link_events`, API, files, messages)	The business-level history: `line.done`, `line.error`, `line.canceled`, `label.printed`, `machine.status`, `line.waste`.
Views `sw_link_v_orders`, `_cuts`, `_waste`, `_machines`	The current state of lines, cuts made, wastes and machines, as plain SQL.
`SoftWindingLinkSvc.exe --console`	Runs the engine in a window instead of as a service, to watch it live (stop with Enter). Combine with `--ini=<path>` to use a test configuration.

Typical engine lines for an order that goes well (excerpt of the log of the test system):

```
12:46:16 INF REST order SO-100234 from ACME: 2 line(s) accepted
12:46:17 INF [M1] claimed ACME_20261028|SO-100234|H07V-K-1.5-BK|10
12:46:17 INF [M1] line ACME_20261028|SO-100234|H07V-K-1.5-BK|10 sent to the PLC:
              H07V-K-1.5-BK x 3 cuts of 12.500 m (preselection 1250)
12:46:25 INF [M1] cut 1/3 confirmed: 12.500 m
```

```
12:46:38 INF [M1] line ACME_20261028|SO-100234|H07V-K-1.5-BK|10 DONE: 3 cuts, 37.500 m
```

8.3 My line does not start

The line is stored (`pending`) but no machine takes it, or the machine takes it and nothing is cut. Check, in this order:

Table 14. Why a line does not start.

What you see	Probable cause	What to do
Machine state `O` (offline) or `D` (disabled), or a heartbeat older than 30 s	The Link cannot reach the PLC, the machine is switched off in the Link, or **the Link service is not running**.	Check the service, the machine network and the **Machines** tab (use the connection test).
The line stays `pending`; the machine is free (`I`)	The line is for another machine, or has no `machineId` and the machine accepts only addressed lines, or an earlier line is still ahead of it (batch priority, `priority`, `item`).	Compare `machineId` with the machine; look at the **Orders** tab ordering.
The machine is `P` (`prepared ...`) and nothing moves	The line was sent to the PLC and the machine waits for the **operator to start it**. The Link does not start machines.	Ask the operator to start. This is normal.
The machine shows the order list on its screen and waits	The machine works in **screen mode**: the operator chooses the order on the machine panel.	The operator picks and starts the order on the screen.
The machine is `S` (`waiting for confirmation`)	The machine is set to external confirmation and waits for your system.	Confirm the cut (REST `confirm`, mailbox action `K`...).
The machine is `E`	Emergency stop, open guard or a PLC error.	The operator clears the cause; the Link continues.
The line is `error`	The PLC refused it or the data were invalid (see `code` and the `line.error` event).	Read the code in chapter 7 (REST guide) / the error table of this guide and the Link log.
The first line works but the next one never starts	The machine waits for the operator after each line (reset, next order) or a previous line is still `claimed`.	Look at the machine state and current order.

8.4 Common problems with this interface

Table 15. Common problems.

Symptom	Cause	Solution
No `connection` messages from the machines	The channel is off, the broker is unreachable, TLS fails, or your subscription is for another prefix.	Check `[Mqtt] Enabled`, `Host`, `Port`. The Link log says `MQTT connected to ... as M1` or the TLS error.
I publish an order and the state does not change	The topic is not the machine's (`.../M1/order`), your publish rights are missing, or the message is not valid JSON.	Subscribe to `uagv/v2/LOIMEX/#` and look at what the broker routes. Check `errors` in the state.
`orderError: the order has no "cut" action`	The node action is not `actionType: "cut"`, or the actions are in `edges`.	Put one `cut` action in each node.
`lineError ... (4001)`	Wrong machine or the machine is not allowed to the MQTT client.	Check the topic and `AllowedMachines`.
`orderUpdateError`	You sent a lower `orderUpdateId`.	Increase it for every change of the same order.
The action stays `INITIALIZING`	The line is on the machine and the operator has not started it.	Normal: the Link never starts machines.
`FAILED instant action not supported`	An action type outside the five supported.	Use only those listed in the factsheet.
`CONNECTIONBROKEN` for all machines	The Link died or lost the network.	Check the Link service; on reconnect it publishes `ONLINE` again.
The state shows an old error	The `errors` list keeps the previous message's error until the next state refresh.	Compare the text with what you just did.
Live data (cable, speed) look stale	They are refreshed with the next state message (up to `StateSeconds`).	Send a `stateRequest` or wait.

8.5 Tracing the traffic

MQTT is easy to observe: connect any MQTT client to the **same broker** and subscribe to everything.

```
mosquitto_sub -h broker -p 8883 --cafile ca.pem --cert client.pem --key client.key -t
"uagv/v2/LOIMEX/#" -v
```

(`mosquitto_sub` and `mosquitto_pub` were not installed on the test computer: the topics and payloads are the ones verified with the Python client. A graphical client such as **MQTT Explorer** shows the same.) The Link log has a line for every connection and every order:

```
13:30:41 INF MQTT connected to broker.example.com:8883 as M1
13:30:41 INF MQTT M1 connected
```

Publish a test order by hand with `mosquitto_pub -t uagv/v2/LOIMEX/M1/order -f order.json -q 1`, where `order.json` is the message of section 2.2.

8.6 What to send to LOIMEX support

- Link version (`GET /health` or the window title) and Windows version.
- The order number, the line (`item`) and the machine.
- The exact request and the **complete answer** (headers and body), or the row / file / telegram / message you sent.
- The log file of the day (`logs\link_YYYYMMDD.log`) and, if possible, an export of the **Events** tab.

- What you expected and what happened, with the time (local time of the server).

9. Testing without a machine

You do not need a real machine to develop. LOIMEX provides **simulators** that behave like a cutting machine and like a customer system. With them you can run every example in this guide at your desk, repeat a failing case as many times as you like and see events appear in the Link window.

Table 16. Simulators (installed in the Link folder with the "Simulators and diagnostic tools" component of the installer).

Program	What it is	Typical use
`SoftWindingLinkSim.exe`	Machine simulator (console). Speaks the PLC protocols of the Omron NX/CP (FINS over UDP, port 9600) and of PLCs without FINS (SFW over TCP, port 63000) and simulates a Zebra printer on TCP 9100. With `--auto` an automatic "operator" starts, measures and resets the counter by itself.	Unattended tests and scripts.
`SoftWindingLink_MachineSimulator.exe`	The same simulator with a **window**: counter, preselection and cut count, with buttons to start, stop, reset the counter, reach the length, cut and edit values. The right panel shows the order the Link writes to the machine.	You play the operator by hand.
`SoftWindingLink_ServerSimulator.exe`	Customer-system simulator with a window: sends orders by database, REST, TCP (AHP) and MQTT and shows the answers.	Compare your integration with a known good client.

9.1 Set up the test machine

1. **Start the machine simulator** in a command prompt: `SoftWindingLinkSim.exe --auto --speed=400 --start-delay=1500 --cut-delay=2000 --printer-port=9100`. It listens on `127.0.0.1`. `--speed` is the counter speed in counter units per second (hundredths of a metre: 400 is 4 m/s; default 2000), `--start-delay` the time before the automatic operator starts, `--cut-delay` the time the operator takes to cut and reset. Add `--noconsole` to run it in the background (otherwise it waits for commands typed in its console) and `--run=SECONDS` to make it stop by itself.
2. **Add the machine to the Link.** In the administration window, **Machines** tab, create `M1`: PLC type *Omron NX1 (FINS)*, address `127.0.0.1`, port `9600`, profile `OMRON_CP`, confirm mode *counter*, cut mode *cut by cut*, enabled. Use the connection test: it must answer. (The database equivalent is the `INSERT INTO sw_link_machines ...` shown in the Link manual, section 4.)
3. **Create your client** in the **Clients** tab (or `SoftWindingLinkSvc.exe --newclient=ACME --channels=MQTT`) and keep the access data it shows.
4. **Check the channel** is enabled in the **Settings** tab and that the engine is running (Status tab).
5. **Run the examples.** The line goes `pending`, `claimed` and `done` in a few seconds; the simulated machine counts every cut and the simulated Zebra receives the labels.

> **Warning.** **Never point a test client at a Link that drives real machines**, unless the machine identifiers you use are reserved for tests: on a real machine an operator could start your test line. Use a separate test Link, or machines that exist only for testing.

9.2 Useful simulator options

Table 17. Machine simulator options.

Option	Effect
`--auto`	Automatic operator: starts, measures up to the preselection, confirms the cut and resets.
`--speed=N`, `--start-delay=MS`, `--cut-delay=MS`	Counter speed and operator delays. Slow values let you watch a line in progress.
`--fins-port=N`, `--sfw-port=N`	Ports. Use different ports to run several simulated machines on one PC (`M1` on 9600, `M2` on 9601).
`--printer-port=9100`, `--printer-file=labels.txt`	Simulated Zebra: the received ZPL is appended to the file.
`--waste=N`, `--wastes=K`	The cut is interrupted at N counter units and the operator resets the counter: a waste (a reset before the preselection). K = how many times (1 by default).
`--noconsole`, `--run=SECONDS`	Without them the simulator waits for typed commands in its console. Use `--noconsole` to run it in the background, and `--run` to make it stop after that many seconds.
`--listen=IP`	Address to listen on (default `127.0.0.1`).

For the broker, use your own test broker or the one in `docs\examples\mqtt\mini_broker.py`. Configure the Link with `[Mqtt] Enabled=1`, `Host=127.0.0.1`, `Port=1883`, `Tls=0`, create the client `MQTT` with the MQTT channel, restart the service and run `vda_client.py`.

The customer simulator `SoftWindingLink_ServerSimulator.exe` has an MQTT tab that acts as a master control against your broker.

10. Going to production

Table 18. Production checklist.

Topic	Recommendation
Broker	Use a managed or clustered broker. Both sides reconnect by themselves; QoS 1 for orders and instant actions.
One order per second	The factsheet limits the order rate (`minOrderInterval`). Send orders for a machine at that pace.
Track by actionId	Give every cut action a unique `actionId` and follow it in `actionStates` until `FINISHED` or `FAILED`.
Persist the result	Store `cutResult` (or the `resultDescription`) of the `FINISHED` action as you see it. A later state may no longer contain it.
Idempotent updates	Increase `orderUpdateId` for every change of an order; resend the same order safely after a failure.
Watch `connection`	Alarm on `CONNECTIONBROKEN` or `OFFLINE`; the machines resume by themselves when the Link comes back.
Errors list	Read `errors` in every state; do not rely on it being empty after a rejected message.
Credentials and rights	A separate client id and credentials for the master control; the broker should limit what it may publish.
Time	VDA timestamps are UTC; the dates inside `resultDescription` and other interfaces are the Link server's local time.

Appendix A. Example client (Python)

The reusable part of `vda_client.py` (`paho-mqtt` 2.x). The `main()` of the file runs the demo of chapter 2.

```
import json
import os
import sys
import threading
import time


import paho.mqtt.client as mqtt



def now():
    return time.strftime('%Y-%m-%dT%H:%M:%S.000Z', time.gmtime())



class Master:
    def __init__(self, host, port=8883, prefix='uagv/v2/LOIMEX', tls=True, ca=None, cert=None, key=None, user=None,
password=None, client_id='erp-master'):
        self.prefix = prefix.rstrip('/')
        self.manufacturer = self.prefix.split('/')[-1]
        self.states, self.connection, self.factsheets = {}, {}, {}
        self.history = []                                # every message received: (topic, payload)
        self.header = 0
        self.lock = threading.Lock()
        self.c = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2, client_id=client_id)
        if tls:
            self.c.tls_set(ca_certs=ca, certfile=cert, keyfile=key)
        if user:
            self.c.username_pw_set(user, password)
        self.c.on_message = self._on_message
        self.c.connect(host, port, 30)
        self.c.loop_start()
        self.c.subscribe(self.prefix + '/+/state', 0)
        self.c.subscribe(self.prefix + '/+/connection', 1)
        self.c.subscribe(self.prefix + '/+/factsheet', 0)


    def _on_message(self, client, userdata, msg):
        serial, leaf = msg.topic.split('/')[-2:]
        try:
            d = json.loads(msg.payload.decode('utf-8'))
        except ValueError:
            return
        with self.lock:
            self.history.append((msg.topic, d))
            {'state': self.states, 'connection': self.connection, 'factsheet': self.factsheets}.get(leaf, {})[serial] = d


    def _header(self, serial):
        self.header += 1
        return {'headerId': self.header, 'timestamp': now(), 'version': '2.0.0', 'manufacturer': self.manufacturer,
'serialNumber': serial}


    # ---- publish ----
    def send_order(self, serial, order_id, cuts, update_id=0):
        """cuts: list of dicts (productId, lengthM, cuts, item, reelId, description...). One node with one 'cut' action
each."""
        nodes = []
        for i, c in enumerate(cuts):
            params = [{'key': k, 'value': v} for k, v in c.items()]
            nodes.append({'nodeId': 'N%d' % (i + 1), 'sequenceId': i * 2, 'released': True, 'actions': [
                {'actionType': 'cut', 'actionId': '%s-%d' % (order_id, i + 1), 'blockingType': 'HARD', 'actionParameters':
params}]})
        msg = dict(self._header(serial), orderId=order_id, orderUpdateId=update_id, nodes=nodes, edges=[])
        self.c.publish('%s/%s/order' % (self.prefix, serial), json.dumps(msg), 1)
        return msg


    def instant(self, serial, action_type, params=None, action_id=None):
        a = {'actionType': action_type, 'actionId': action_id or '%s-%d' % (action_type, int(time.time())), 'blockingType':
'NONE'}
        if params:
            a['actionParameters'] = [{'key': k, 'value': v} for k, v in params.items()]
        msg = dict(self._header(serial), instantActions=[a])
        self.c.publish('%s/%s/instantActions' % (self.prefix, serial), json.dumps(msg), 1)
        return a['actionId']


    # ---- read ----
    def wait(self, cond, timeout=60, step=0.2):
        end = time.time() + timeout
        while time.time() < end:
```

```
            with self.lock:
                if cond():
                    return True
            time.sleep(step)
        return False


    def action_status(self, serial, order_id, action_id):
        s = self.states.get(serial)
        if not s or s.get('orderId') != order_id:
            return None
        return next((a.get('actionStatus') for a in s.get('actionStates', []) if a.get('actionId') == action_id), None)


    def close(self):
        self.c.loop_stop()
        self.c.disconnect()
```

Appendix B. Full factsheet and state

The factsheet of a machine, as published (retained on `.../M1/factsheet`):


```
{
  "headerId": 0,
  "timestamp": "2026-10-28T12:30:41.327Z",
  "version": "2.0.0",
  "manufacturer": "LOIMEX",
  "serialNumber": "M1",
  "typeSpecification": {
    "seriesName": "SoftWindingLink cutting machine",
    "seriesDescription": "Stationary cutting/winding machine controlled by SoftWindingLink (not a moving vehicle)",
    "agvKind": "DIFFERENTIAL",
    "agvClass": "CARRIER",
    "maxLoadMass": 0,
    "localizationTypes": ["NATURAL"],
    "navigationTypes": ["PHYSICAL_LINE_GUIDED"]
  },
  "physicalParameters": {
    "speedMin": 0,
    "speedMax": 0,
    "accelerationMax": 0,
    "decelerationMax": 0,
    "heightMax": 0,
    "width": 0,
    "length": 0
  },
  "protocolLimits": {
    "maxStringLens": {
      "msgLen": 65535,
      "topicSerialLen": 64,
      "topicElemLen": 64,
      "idLen": 64,
      "idNumericalOnly": false,
      "enumLen": 32,
      "loadIdLen": 32
    },
    "maxArrayLens": {
      "order.nodes": 100,
      "order.edges": 100,
      "node.actions": 1,
      "edge.actions": 0,
      "actions.actionsParameters": 20,
      "instantActions": 10,
      "trajectory.knotVector": 0,
      "trajectory.controlPoints": 0,
      "state.nodeStates": 100,
      "state.edgeStates": 100,
      "state.loads": 0,
      "state.actionStates": 200,
      "state.errors": 50,
      "state.information": 50,
      "error.errorReferences": 10,
      "information.infoReferences": 10
    },
    "timing": {"minOrderInterval": 1.0, "minStateInterval": 0.25, "defaultStateInterval": 5.0}
  },
  "protocolFeatures": {
    "optionalParameters": [],
    "agvActions": [
      {
        "actionType": "cut",
        "actionDescription": "Cut the product to length: one line of the SoftWindingLink order",
        "actionScopes": ["NODE"],
        "actionParameters": [
          {
            "key": "productId",
            "valueDataType": "STRING",
            "description": "Product to cut (Link productId)",
            "isOptional": false
          },
          {
            "key": "lengthM",
            "valueDataType": "FLOAT",
            "description": "Length of each cut in metres",
            "isOptional": false
          },
          {
            "key": "cuts",
            "valueDataType": "INT",
            "description": "Number of cuts of the line",
            "isOptional": false
          },
          {
            "key": "reelId",
            "valueDataType": "STRING",
```

```
            "description": "Reel the material is cut from",
            "isOptional": true
          },
          {
            "key": "item",
            "valueDataType": "INT",
            "description": "Line number inside the order (default: 10, 20, 30 ...)",
            "isOptional": true
          },
          {
            "key": "description",
            "valueDataType": "STRING",
            "description": "Line description (printed on the label)",
            "isOptional": true
          },
          {
            "key": "customer",
            "valueDataType": "STRING",
            "description": "End customer name",
            "isOptional": true
          },
          {
            "key": "batchId",
            "valueDataType": "STRING",
            "description": "Batch identifier (default: <client>_<date>)",
            "isOptional": true
          },
          {
            "key": "priority",
            "valueDataType": "INT",
            "description": "Priority of the line (0 = normal)",
            "isOptional": true
          },
          {
            "key": "free1",
            "valueDataType": "STRING",
            "description": "Free text fields free1 ... free10",
            "isOptional": true
          },
          {"key": "notes", "valueDataType": "STRING", "description": "Notes", "isOptional": true},
          {
            "key": "cableDiameterMm",
            "valueDataType": "FLOAT",
            "description": "Cable diameter in mm: the machine is set to manual diameter with it before the line (default: leave
as is)",
            "isOptional": true
          }
        ],
        "resultDescription": "state.actionStates and information report the measured length and the cuts made",
        "blockingTypes": ["HARD"]
      },
      {
        "actionType": "cancelOrder",
        "actionDescription": "Cancel what the machine has on its screen (and the pending lines of the current order); refused
while the machine is running",
        "actionScopes": ["INSTANT"],
        "blockingTypes": ["HARD"]
      },
      {
        "actionType": "setCableDiameter",
        "actionDescription": "Set the cable diameter in mm (manual mode)",
        "actionScopes": ["INSTANT"],
        "actionParameters": [
          {
            "key": "diameterMm",
            "valueDataType": "FLOAT",
            "description": "Cable diameter in mm",
            "isOptional": false
          }
        ],
        "blockingTypes": ["NONE"]
      },
      {
        "actionType": "cableLaser",
        "actionDescription": "Back to the laser gauge",
        "actionScopes": ["INSTANT"],
        "blockingTypes": ["NONE"]
      },
      {"actionType": "stateRequest", "actionScopes": ["INSTANT"], "blockingTypes": ["NONE"]},
      {"actionType": "factsheetRequest", "actionScopes": ["INSTANT"], "blockingTypes": ["NONE"]}
    ]
  },
  "agvGeometry": {},
  "loadSpecification": {}
}
```


The state of a machine at rest (`.../M2/state`):


```
{
  "headerId": 28,
```

```
  "timestamp": "2026-10-28T12:32:41.913Z",
  "version": "2.0.0",
  "manufacturer": "LOIMEX",
  "serialNumber": "M1",
  "orderId": "MQ-1793190691",
  "orderUpdateId": 0,
  "lastNodeId": "N2",
  "lastNodeSequenceId": 2,
  "nodeStates": [],
  "edgeStates": [],
  "driving": false,
  "paused": false,
  "newBaseRequest": false,
  "operatingMode": "AUTOMATIC",
  "actionStates": [
    {
      "actionId": "MQ-1793190691-1",
      "actionType": "cut",
      "actionStatus": "FINISHED",
      "resultDescription": "measured 3.000 m, 1 cut(s)"
    },
    {
      "actionId": "MQ-1793190691-2",
      "actionType": "cut",
      "actionStatus": "FINISHED",
      "resultDescription": "measured 4.000 m, 2 cut(s)"
    }
  ],
  "velocity": {"vx": 0.0, "vy": 0, "omega": 0},
  "batteryState": {"batteryCharge": 100.0, "charging": false},
  "errors": [],
  "information": [
    {
      "infoType": "cutResult",
      "infoReferences": [
        {"referenceKey": "orderId", "referenceValue": "MQ-1793190691"},
        {"referenceKey": "item", "referenceValue": "10"},
        {"referenceKey": "actionId", "referenceValue": "MQ-1793190691-1"},
        {"referenceKey": "measuredM", "referenceValue": "3.000"},
        {"referenceKey": "cuts", "referenceValue": "1"}
      ],
      "infoDescription": "measured 3.000 m, 1 cut(s)",
      "infoLevel": "INFO"
    },
    {
      "infoType": "cutResult",
      "infoReferences": [
        {"referenceKey": "orderId", "referenceValue": "MQ-1793190691"},
        {"referenceKey": "item", "referenceValue": "20"},
        {"referenceKey": "actionId", "referenceValue": "MQ-1793190691-2"},
        {"referenceKey": "measuredM", "referenceValue": "4.000"},
        {"referenceKey": "cuts", "referenceValue": "2"}
      ],
      "infoDescription": "measured 4.000 m, 2 cut(s)",
      "infoLevel": "INFO"
    },
    {
      "infoType": "machineStatus",
      "infoReferences": [
        {"referenceKey": "counterM", "referenceValue": "0.000"},
        {"referenceKey": "cutNr", "referenceValue": "1"},
        {"referenceKey": "state", "referenceValue": "I"}
      ],
      "infoDescription": "ready",
      "infoLevel": "INFO"
    },
    {
      "infoType": "processTime",
      "infoReferences": [
        {"referenceKey": "seconds", "referenceValue": "0"},
        {"referenceKey": "speedMPerMin", "referenceValue": "0.000"}
      ],
      "infoDescription": "00:00",
      "infoLevel": "INFO"
    },
    {
      "infoType": "cableDiameter",
      "infoReferences": [
        {"referenceKey": "diameterMm", "referenceValue": "12.500"},
        {"referenceKey": "mode", "referenceValue": "laser"}
      ],
      "infoDescription": "12.500 mm, laser",
      "infoLevel": "INFO"
    }
  ],
  "safetyState": {"eStop": "NONE", "fieldViolation": false}
}
```