Process Automation

Our product brands:
IMI TWTG

NEON Steam Trap Sensor v1

Advanced Communication Protocol

Breakthrough engineering for a better world

Revision A1
Date 2026-06-11

Contents

Revision History

Revision	Date	Description
A1	2026-06-11	STS v1 protocol.

Versioning scheme

- Major version changes (e.g. A to B) indicate functional and protocol updates.
- Minor version changes (e.g. A1 to A2) indicate document updates only, with no change to the protocol itself.

1 Introduction

This document describes the LoRaWAN communication protocol used by the NEON .

A basic understanding of LoRaWAN is assumed, but it is not necessary to fully understand this document. Its primary objective is to outline the protocol used for the communication between a TWTG's NEON sensor and a *LoRaWAN Network Server* (LNS). For detailed information about the device, please refer to the product manual.

This document is intended for experienced users and integrators who require a comprehensive understanding of the device's communication protocol. It can be used to implement custom codecs, diverging from the provided JavaScript implementation on the product support page. This document contains the complete technical specification, including all advanced configurations and internal protocol details. Users are advised to exercise caution when using some of the device configurations contained in this document, as they can lead to unintended behavior if not properly understood.

Warning:
The user is advised to excercise caution when using the following configurations:

- Transmitter Battery Reset
- Factory Reset
- Transmitter Configuration

The standard user is advised to use the other version of this document, which is provided on the product support page in the same location as this document.

The device implements the following LoRaWAN specifications:

- LoRaWAN® L2 1.0.4 (TS001-1.0.4)
- RP002-1.0.3 LoRaWAN® Regional Parameters
- LoRaWAN® Firmware Management Protocol Specification TS006-1.0.0.

To participate in a LoRaWAN network, *Over-The-Air Activation* (OTAA) is used. *Activation by Personalization* (ABP) is not supported.

1.1 Binary Data Encoding

This section explains the binary encoding of messages used by the device.

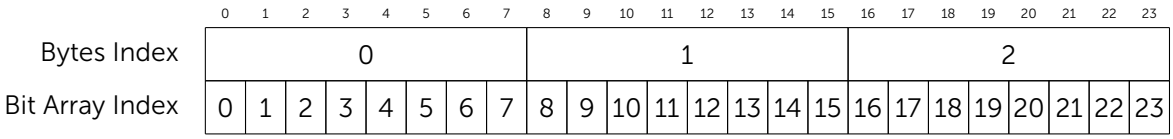

Figure 1.1: Mapping of Byte Index to Bit Index

For the purposes of decoding, the `input.bytes` array passed to the `decodeUplink()` codec function is treated as a contiguous array of bits. The most significant bit of the first byte has index 0 in the bit array. This is illustrated in Figure 1.1.

The process of decoding consists of taking bits from the bit array and converting them into a value based on the type. Reading a group of N bits consists of taking N bits from the bit array, starting at bit index 0. The bit with the lowest index in the bit array becomes the most significant bit in the taken bits. Figure 1.2 shows how bit groups of varying sizes are read in succession.

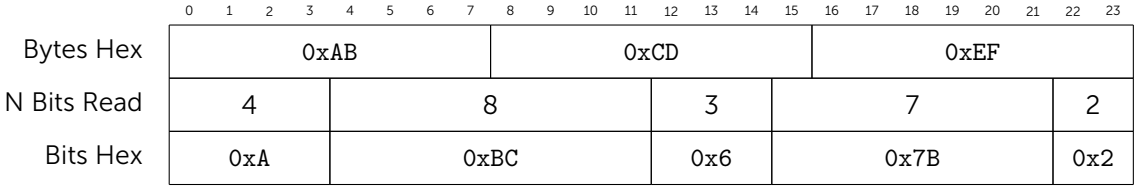

Figure 1.2: Bit Reading Example

The encoding process is symmetrical to the decoding. Bits are written starting from bit index 0 into the `output.data` array returned from the `encodeDownlink()` codec function. Code examples on how to read from, and to write bits to a byte array are provided in Appendix A.1.

1.2 Message Structure

This section explains the structure of the LoRaWAN uplinks and downlinks used by the device.

Both uplink and downlink messages share a common header format. The first byte (8 bits) contains two 4-bit fields: **Message ID** (bits 0-3) and **Message Version** (bits 4-7). Together with the LoRaWAN FPort, these fields uniquely identify the message type. The **Message Version** indicates the schema version for that specific message. The remaining bytes contain the data fields defined for that message.

Each field has an **ID** and **Type**. The **ID** serves as a key in the decoded object, while the **Type** determines the size of the field and value encoding. Table 1.1 defines the simple types used in the device messages, other types may be used by the device and are explained when relevant. All data is provided in SI units. Upon request, TWTG can provide a conversion to United States Customary (USC) units.

Table 1.1: Field Types

Field Type	Bit Size	Description	Examples
`bool`	1	Boolean value	`0 → false` `1 → true`
`uintN`	N	Unsigned integer	`uint8 0x8A → 138` `uint4 0xF → 15`
`intN`	N	Signed integer	`int8 0x8A → -118` `int4 0xF → -1`
`float32`	32	IEEE 754-2008 **binary32** floating point number (for code example see Appendix A.2)	`0x422A0000 → 42.5` `0xC22A0000 → -42.5`
`float16`	16	IEEE 754-2008 **binary16** floating point number (for code example see Appendix A.3)	`0x5150 → 42.5` `0xD150 → -42.5`
`pfloat15`	15	Same as `float16`, but without the sign bit (for code example see Appendix A.3)	`0x5150 → 42.5`

Figure 1.3 shows a decoding example of a message with five fields from `input.bytes`.

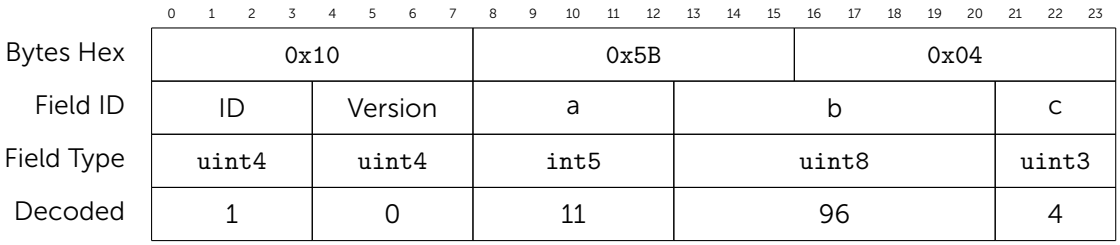

Figure 1.3: Message Decoding Example

1.3 Special Types Encoding

This section explains the encoding of special types of data. These types are used in the context of binary message encoding and decoding.

1.3.1 Short Timestamp

The short timestamp is a 16 bit unsigned integer representing the number of minutes since the Unix epoch (1970-01-01 00:00:00 UTC). The timestamp is truncated at 16 bits minus 1, which means it will overflow every 65535 minutes (45 days). The timestamp is used to represent the time of an event in the device, such as a measurement.

The last value, 0xffff, is reserved for the case when the timestamp is not available. In that case the server should use the receive time of the message as the timestamp.

The short timestamp is used only in uplink messages, and therefore only needs to be decoded. The short timestamp can be converted to an absolute timestamp by taking the message receive time into account as depicted in Figure 1.4. As specified in TS013-1.0.0, the receive time of the message (`recvTime`) must be provided for successful decoding.

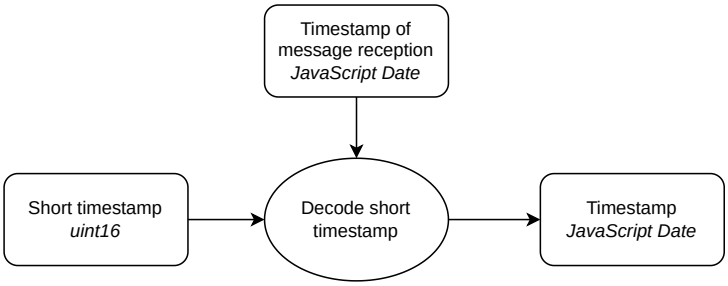

Figure 1.4: Using receive time to decode short timestamp

The short timestamp is used to represent the time on the device within a range of 21 days. The absolute timestamp can be calculated following the code as listed in Appendix A.4.

For example:

- A measurement on the device is performed at 2023-08-10T11:30:00.000Z.
- The device stores the short timestamp in minutes: 28194450.
- The device sends the measurement with the timestamp encoded in uint16: 14400 (28194450 wrapped at 65535),
- The server receives the message at 2023-08-10T11:31:00.000Z.
- The decoder uses the receive time and the short timestamp to calculate the absolute timestamp: 2024-02-07T20:08:29 UTC, as listed in Listing 29.
  - recvTime = 2023-08-10T11:31:00.000Z = 1691667060000
  - recvMin = recvTime / 60000 = 28194451
  - wrapOffset = recvMin % 65535 = 14401
  - wrapBase = recvMin - wrapOffset = 28194451 - 14401 = 28180050
  - gap = wrapOffset - shortTimestamp = 14401 - 14400 = 1
  - gap is positive and smaller than 21 days
  - msgMinutes = wrapBase + shortTimestamp = 28180050 + 14400 = 28194450
  - msgTime = msgMinutes * 60000 = 1691667000000 = 2023-08-10T11:30:00.000Z

1.3.2 Timing

Device actions can be configured by scheduling tasks, with the `timing` type determining when each task is executed.

Periodic	0	period	0
Cron	1	cron	

Figure 1.5: Encoding of timing

It is a 92-bit unsigned integer that can be interpreted as either a `cron` expression or a 15-bit time period in minutes (see Figure 1.5). If the most significant bit is set, the remaining bits are interpreted as a `cron` expression. If the most significant bit is not set, the next 15 bits represent the time period, and the remaining bits must be 0.

`cron`[1] is a time-based job scheduler in Unix-like operating systems. It is used to schedule jobs (commands or scripts) to run periodically at fixed times, dates, or intervals. The cron expression is a string representing the schedule for the job to be executed. For this device the cron expression is used to trigger the scheduled tasks.

The cron expression consists of 5 fields separated by spaces:

`"<minute> <hour> <day-of-month> <month> <day-of-week>"`

Each field can have a single value, a range of values, or a list of values separated by commas. The fields are as follows:

- `<minute>` - minute of the hour (0-59)
- `<hour>` - hour of the day (0-23)
- `<day-of-month>` - day of the month (1-31)
- `<month>` - month of the year (1-12)
- `<day-of-week>` - day of the week (0-6, 0 is Sunday)

For example, the cron expression" `"0 * * * *"` means "every hour at the beginning of the hour". Online tools are available to help generate cron expressions, such as https://crontab.guru/.

Cron binary encoding

The cron expression is encoded as a 91-bit bitmask:

- `<minute>` - 60 bits
- `<hour>` - 24 bits
- `<day-of-week>` - 7 bits

Note that `<day-of-month>` and `<month>` fields are not supported and must always be: `*`.

Examples

Cron Example 1 `"0 */4 * * *"` Schedule trigger every 4 hours

Cron Example 2 `"*/15 8-17 * * 1-5"` Schedule trigger every 15 minutes between 8:00 AM and 5:00 PM on weekdays

Cron Example 3 `"2,4,6 */2 * * *"` Schedule trigger on 2, 4 and 6 minutes past every second hour

Timing Example 1 `"0x00f00000000000000000000"` Periodic trigger every 4 hours

[1] https://en.wikipedia.org/wiki/Cron

Timing Example 2 `"0x80010002000400081ff803e"` Cron trigger of `"*/15 8-17 * * 1-5"` (every 15 minutes between 8:00 AM and 5:00 PM on weekdays)

See Appendix A.5 and Appendix A.6 for encoding code examples for `cron` and `timing` respectively.

1.3.3 Reserved Fields

Some fields in the protocol are designated as *Reserved for Future Use* (RFU). The encoded value of these fields must be 0. The device rejects downlinks where RFU fields are not 0.

1.4 Firmware and Hardware Versions

The firmware and hardware versions of the transmitter and the sensor can be checked using the **DevVersionReq** downlink as described in the LoRaWAN® Firmware Management Protocol Specification TS006-1.0.0. The `FW_version` field of the **DevVersionAns** uplink encodes the sensor firmware version in the upper two bytes and the transmitter firmware version in the lower two bytes. For example given a `FW_version` of `0xaabbccdd`, sensor firmware version is `aabb` and transmitter version is `ccdd`. The sensor and transmitter hardware versions are encoded the same way in the `HW_version` field.

The protocol described in this document applies to the firmware versions listed in Table 1.2 and Table 1.3.

Table 1.2: Sensor Firmware Versions

Version	Protocol Revision
91d2	A

Table 1.3: Transmitter Firmware Versions

Version	Protocol Revision
ed7a	A

The hardware versions of the sensor and transmitter are encoded in the `HW_version` field of the **DevVersionAns** uplink.

Table 1.4: Sensor Hardware Versions

Version	Sensor
0007	DS-SX-02-00

Table 1.5: Transmitter Hardware Versions

Version	Transmitter
0003	DS-LD-02-xx
0006	XT-01-xx

1.4.1 Example

The following is an example of a **DevVersionReq** downlink and a **DevVersionAns** uplink.

```
{
    "message_name": "dev_version_req"
}
```

Listing 1: Example of the JSON structure for **DevVersionReq** downlink.

```
{
    "message_name": "dev_version_ans",
    "FW_version": "0x91d2ed7a",
    "HW_version": "0x00070003"
}
```

Listing 2: Example of the JSON structure for **DevVersionAns** uplink.

1.5 Document Content

This document is structured according to the functionality provided by the device. Each section describes all the messages and configurations associated with a given functionality. The table below offers an overview of all messages and configurations with their associated sections.

Table 1.6: Overview of the device messages and configurations

Section	Message	Configuration
Configuration	**configuration_update_request** **configuration_update_answer**	
Scheduling	**schedule_update_request** **schedule_update_answer**	
Transmitter	**transmitter_deactivated** **transmitter_boot** **transmitter_status** **transmitter_battery** **transmitter_battery_reset_request** **transmitter_battery_reset_answer** **factory_reset_request**	**transmitter**
Sensor	**sts_measurement**	**sts_calibration**

2 Configuration

2.1 Update

The device can be configured over LoRaWAN with a **configuration_update_request** downlink. This section explains only the configuration mechanism. The supported configurations are explained in the relevant sections.

The **configuration_update_request** downlink is used to change the configuration of the device. It consists of a `payload` and an associated `tag`. The `payload` contains the desired configuration settings and the `tag` is a 32 bit number assigned to each configuration by the user.

The following should be considered when choosing a `tag`:

- It is an arbitrary value which is used to differentiate between configurations.
- It is recommended to set it to the hash of the `payload` or use a distinct value for each configuration.
- The `tags` within the range of `0xFFFF0000`–`0xFFFFFFFF` are reserved by the manufacturer.

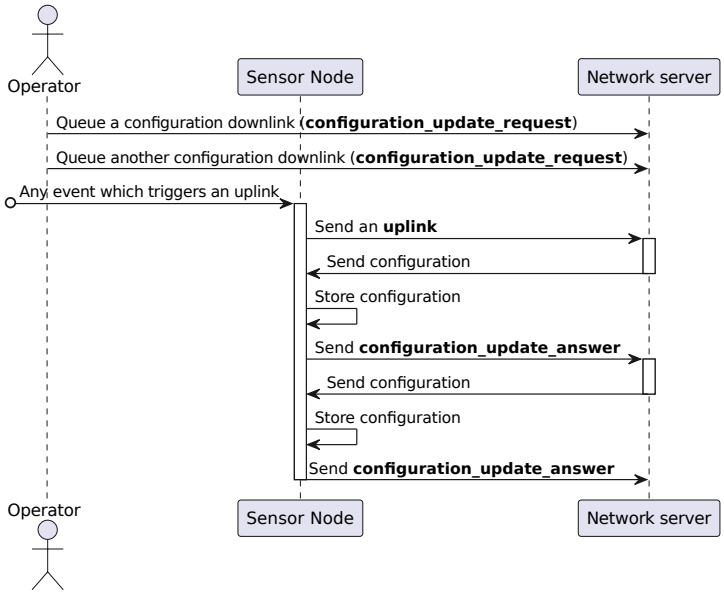

Figure 2.1: Update of Multiple Configurations

After a **configuration_update_request** is processed, the device sends a **configuration_update_answer** uplink indicating whether the configuration has been applied or not. Multiple **configuration_update_request** downlinks can be queued before the next uplink message, allowing the device to process them in a batch. An update of multiple configurations is shown Figure 2.1. If several **configuration_update_requests** of the same configuration type are handled in a batch, the device will adopt the configuration specified in the last processed **configuration_update_request** among those of the corresponding type.

Note: The *TWTG NEON Configuration Generator*[2] can be used to create and encode the **configuration_update_request** downlinks.

[2] `https://neon-configurator.twtg.io/`

2.1.1 configuration_update_request message

Message Structure

```
{
    "message_name": "configuration_update_request",
    "message_version": 0,
    "tag": <number>,
    "payload": <object>
}
```

Listing 3: JSON structure for **configuration_update_request** message

Binary Encoding and Description

Table 2.1: Binary encoding for **configuration_update_request** message sent on FPort 11

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 0.
message_version	uint4	4..7	Message version: 0.
tag	uint32	8..39	Value used to identify the configuration.
payload		40..	Configuration payload. The definition varies by configuration type, as explained in the relevant sections. All configurations share a common header structure, including 12 bits for the configuration type and 4 bits for the configuration version.

2.1.2 configuration_update_answer message

The **configuration_update_answer** is sent by the device in response to a **configuration_update_request**. It contains information about the success or failure of the configuration update.

The `tag` included in the message is of the currently used configuration.

Message Structure

```
{
    "message_name": "configuration_update_answer",
    "message_version": 0,
    "tag": <number>,
    "type": <string>,
    "status": <string>
}
```

Listing 4: JSON structure for **configuration_update_answer** message

Binary Encoding and Description

Table 2.2: Binary encoding for **configuration_update_answer** message sent on FPort 11

Field ID	Type	Bit Index	Description
`id`	`uint4`	0..3	Message ID: 0.
`message_version`	`uint4`	4..7	Message version: 0.
`tag`	`uint32`	8..39	Value used to identify the configuration.
`type`	`uint12`	40..51	Type of the configuration sent through the **configuration_update_request**. For value mapping see the table below.
`status`	`uint4`	52..55	Status of the configuration update. For value mapping see the table below.

Enum Value Mappings

Table 2.3: Value mapping of **device_configuration_type_v0**

Value mapping	Value
`"transmitter"`	1
`"sts_calibration"`	15

Table 2.4: Value mapping of **configuration_update_status_v0**

Value mapping	Value
`"success"`	0
`"rejected_unsupported_configuration_type"`	1
`"rejected_unsupported_configuration_version"`	2
`"rejected_invalid_configuration_values"`	3
`"rejected_decoding_failed"`	4
`"rejected_schedule_type_limit"`	5
`"sensor_communication_failure"`	6

3 Scheduling

The device executes tasks using a flexible scheduling system, allowing users to define when specific tasks should occur. This system supports multiple schedules, each with its own configuration. For example, a schedule could trigger a task every 15 minutes between 8:00 and 17:00 or every 4 hours. The following sections explain how these schedules operate and how they are configured for all tasks.

3.1 Synchronized Schedules

The *synchronized* schedules align with the real-time provided by the LoRa Network Server. A task triggered by a synchronized schedule is executed at a precise time (e.g., every day at 2 PM). They should be used when synchronization of measurements across multiple devices is required.

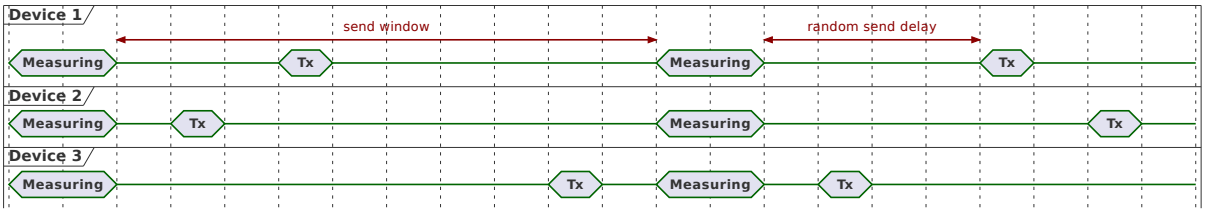

Figure 3.1: Synchronized Measurement

Figure 3.1 shows three devices with the same synchronized measurement schedule. Note that the synchronized mode introduces a *random send delay* post-measurement and the transmission always occurs before the next scheduled measurement. This is done to avoid network congestion.

3.2 Unsynchronized Schedules

The *unsynchronized* schedules run at an offset relative to real time, and this offset is randomized per device. They should be used whenever synchronization across devices is not required.

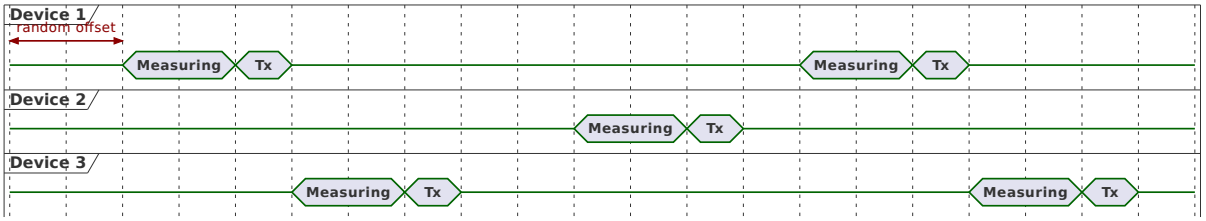

Figure 3.2: Unsynchronized Measurement

Figure 3.2 shows three devices configured with the same unsynchronized measurement schedule. The measurements occur at a fixed interval, with a randomized offset.

3.3 Schedule Update

Tasks can be scheduled over LoRaWAN with a **schedule_update_request** downlink. This section explains only the scheduling mechanism. The supported tasks are explained in the relevant sections.

The **schedule_update_request** downlink is used to change the schedules of the device. It puts a task in a specific slot on the device. Slots are defined per task type. The maximum number of available slots varies per task type, as listed in Table 3.1.

Table 3.1: Task types

Task Type	Number of schedule slots	Encoded Value
`transmitter_status`	1	0
`transmitter_battery`	1	15
`sts_measurement`	1	25

Execution timing can be defined using either a `cron` expression or a time period in minutes. Schedules defined with a `cron` expression are synchronized across devices, while those using a time period in minutes are not.

3.3.1 schedule_update_request message

Message Structure

```
{
    "message_name": "schedule_update_request",
    "message_version": 0,
    "slot": <number>,
    "command": <string>,
    "timing": <string or number>,
    "triggered_on_button_press": <boolean>,
    "send": <boolean>,
    "task": <object>
}
```

Listing 5: JSON structure for **schedule_update_request** message

Binary Encoding and Description

Table 3.2: Binary encoding for **schedule_update_request** message sent on FPort 11

Field ID	Type	Bit Index	Description
`id`	`uint4`	0..3	Message ID: 2.
`message_version`	`uint4`	4..7	Message version: 0.
`slot`	`uint4`	8..11	The slot used for the request.
`command`	`uint4`	12..15	Instructs the device how the schedule should be handled. **set** Sets the schedule in the specified slot. Overwrites any existing schedule in the specified slot. **replace** Removes all schedules of the same task type and sets this one in the specified slot. **execute** Executes the schedule immediately without storing. Only the `task` field is considered. **reset** Removes all schedules of the specified type, ignoring all other fields. **remove** Removes the schedule from the specified slot. For value mapping see the table below.
`timing`		16..107	Specifies when the schedule is executed, which can be defined either as a cron expression or as a period in minutes (1–32767). Detailed information about the binary encoding is provided in Section 1.3.2.
`triggered_on_button_press`	`bool`	108	If `true`, the scheduled task is also triggered on a short button press by the user. If the `timing` field is 0, the task will only be executed on button press.
`send`	`bool`	109	If `true`, the scheduled task triggers a transmission.
Reserved for Future Use	`uint2`	110..111	

Continues on next page

Table 3.2 Continued: Binary encoding for **schedule_update_request** message sent on FPort 11

Field ID	Type	Bit Index	Description
`task`		112..	Scheduled task and its parameters. The definition varies by the task type, as explained in the relevant sections. All tasks share a common header structure, including 12 bits for the task type and 4 bits for the task version. The supported tasks are listed in Table 3.1.

Enum Value Mappings

Table 3.3: Value mapping of **schedule_set_command_v0**

Value mapping	Value
`"set"`	0
`"replace"`	1
`"execute"`	2
`"reset"`	3
`"remove"`	4

3.3.2 schedule_update_answer message

The **schedule_update_answer** is sent by the device in response to a **schedule_update_request**. It contains information about the success or failure of the schedule update.

Message Structure

```
{
    "message_name": "schedule_update_answer",
    "message_version": 0,
    "<task_type>_<slot>_status": <string>,
    "<task_type>_number_of_slots": <number>,
    "<task_type>_slot_0_available": <boolean>,
    "<task_type>_slot_1_available": <boolean>,
    "<task_type>_slot_2_available": <boolean>,
    "<task_type>_slot_3_available": <boolean>,
    "<task_type>_slot_4_available": <boolean>,
    "<task_type>_slot_5_available": <boolean>,
    "<task_type>_slot_6_available": <boolean>,
    "<task_type>_slot_7_available": <boolean>,
    "<task_type>_slot_8_available": <boolean>,
    "<task_type>_slot_9_available": <boolean>
}
```

Listing 6: JSON structure for **schedule_update_answer** message

Binary Encoding and Description

Table 3.4: Binary encoding for **schedule_update_answer** message sent on FPort 11

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 2.
message_version	uint4	4..7	Message version: 0.
slot	uint4	8..11	The slot that was used in the request.
task_type	uint12	12..23	Type of the schedule sent through the **schedule_update_request**. For value mapping see the table below.
status	uint4	24..27	Status of the schedule update. For value mapping see the table below.
number_of_slots	uint4	28..31	Total number of slots available for this task. See Table 3.1.
slot_0_available	bool	32	If `true`, the slot 0 is free. No schedule is saved in this slot. Similarly, `slot_N_available` indicates if the slot N is free. Only supported slots are included in the decoded object, based on `number_of_slots`.
slot_1_available	bool	33	See `slot_0_available`.
slot_2_available	bool	34	See `slot_0_available`.
slot_3_available	bool	35	See `slot_0_available`.
slot_4_available	bool	36	See `slot_0_available`.
slot_5_available	bool	37	See `slot_0_available`.

Continues on next page

Table 3.4 Continued: Binary encoding for **schedule_update_answer** message sent on FPort 11

Field ID	Type	Bit Index	Description
`slot_6_available`	`bool`	38	See `slot_0_available`.
`slot_7_available`	`bool`	39	See `slot_0_available`.
`slot_8_available`	`bool`	40	See `slot_0_available`.
`slot_9_available`	`bool`	41	See `slot_0_available`.
Reserved for Future Use	`uint6`	42..47	

Enum Value Mappings

Table 3.5: Value mapping of **schedule_type_v0**

Value mapping	Value
`"transmitter_status"`	0
`"transmitter_battery"`	15
`"sts_measurement"`	25

Table 3.6: Value mapping of **configuration_update_status_v0**

Value mapping	Value
`"success"`	0
`"rejected_unsupported_configuration_type"`	1
`"rejected_unsupported_configuration_version"`	2
`"rejected_invalid_configuration_values"`	3
`"rejected_decoding_failed"`	4
`"rejected_schedule_type_limit"`	5
`"sensor_communication_failure"`	6

4 Transmitter

4.1 Activation and Deactivation

Upon successful activation, the transmitter sends the **transmitter_status** uplink. When the transmitter is deactivated, it sends the **transmitter_deactivated** uplink. Afterwards, LoRaWAN communication is ceased until the transmitter is activated again. The activation and deactivation sequence is shown in Figure 4.1.

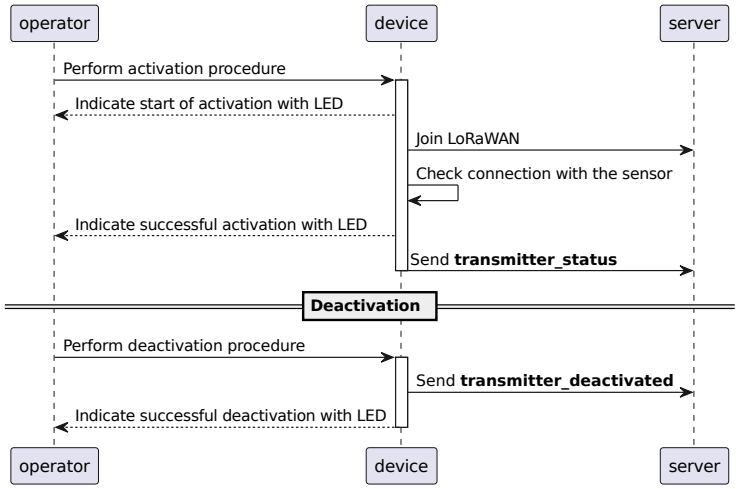

Figure 4.1: Transmitter Activation and Deactivation Sequence

During activation, the transmitter attempts to join the LoRaWAN Network Server. If the join procedure fails, the device will not be activated, and the process is aborted.

Additionally, during the activation, the transmitter verifies communication with the connected sensor. If this verification fails, activation is aborted, and a **transmitter_deactivated** uplink is sent with the reason: `"activation_sensor_comm_fail"`.

4.1.1 transmitter_deactivated message

Message Structure

```
{
    "message_name": "transmitter_deactivated",
    "message_version": 0,
    "deactivation_reason": <string>
}
```

Listing 7: JSON structure for **transmitter_deactivated** message

Binary Encoding and Description

Table 4.1: Binary encoding for **transmitter_deactivated** message sent on FPort 16

Field ID	Type	Bit Index	Description
`id`	`uint4`	0..3	Message ID: 3.
`message_version`	`uint4`	4..7	Message version: 0.
`deactivation_reason`	`uint8`	8..15	Reason for the device's deactivation. For value mapping see the table below.

Enum Value Mappings

Table 4.2: Value mapping of **deactivation_reason_v0**

Value mapping	Value
`"user_triggered"`	0
`"activation_sensor_comm_fail"`	1
`"activation_sensor_protocol_mismatch"`	2

4.2 Button Press

While the device is activated, pressing the button triggers the transmission of a **transmitter_status** message (see Section 4.4). It is sent confirmed and the transmission status is indicated by the device LED. For more information on device LED sequences and their meaning, please refer to the user manual.

Additionally, the button press triggers execution of all schedules with `"triggered_on_button_press": true` (see Section 3.3.1).

The device response to a button press is shown in Figure 4.2.

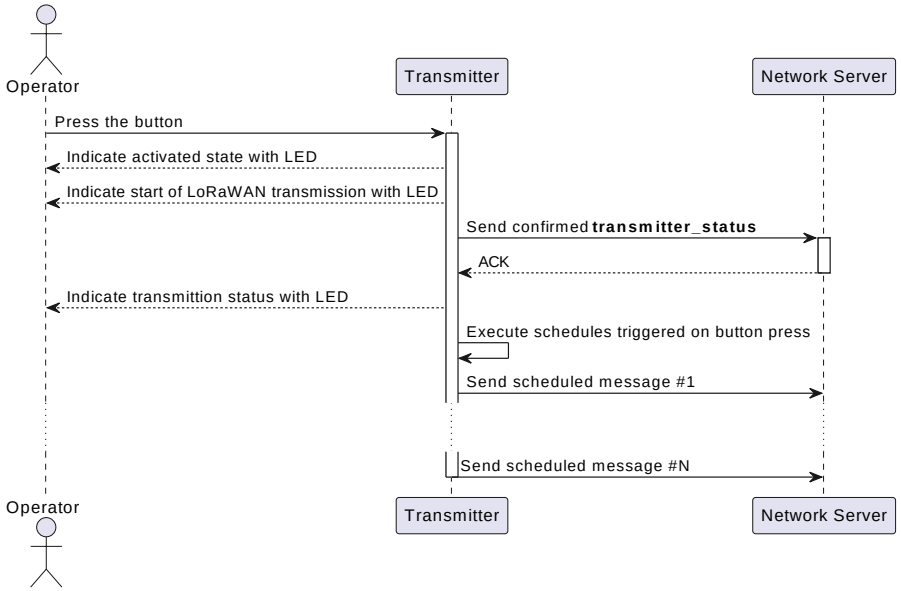

Figure 4.2: Transmitter Button Press Sequence

4.3 Boot

After every reboot, the transmitter reports the reboot reason with the **transmitter_boot** uplink.

Reboots might occur intentionally in case of activating a newly received configuration or a firmware update. Otherwise, the device might reboot due to a system error as a matter of recovery.

4.3.1 transmitter_boot message

Message Structure

```
{
    "message_name": "transmitter_boot",
    "message_version": 0,
    "transmitter_reboot_reason": <string>
}
```

Listing 8: JSON structure for **transmitter_boot** message

Binary Encoding and Description

Table 4.3: Binary encoding for **transmitter_boot** message sent on FPort 16

Field ID	Type	Bit Index	Description
`id`	`uint4`	0..3	Message ID: 0.
`message_version`	`uint4`	4..7	Message version: 0.
`transmitter_reboot_reason`	`uint16`	8..23	Reason for the device's reboot. For value mapping see the table below.

Enum Value Mappings

Table 4.4: Value mapping of **reboot_reason_v0**

Value mapping	Value
`"none"`	0
`"configuration_update"`	1
`"firmware_update_success"`	2
`"firmware_update_rejected"`	3
`"firmware_update_error"`	4
`"firmware_update_in_progress"`	5
`"button_reset"`	6
`"power_black_out"`	7
`"power_brown_out"`	8
`"power_safe_state"`	9
`"system_failure"`	10
`"factory_reset"`	11
`"reboot_request"`	12

4.4 Status

The transmitter reports maintenance and device health information in the **transmitter_status** uplink. It is sent periodically, on short button press, and on activation.

4.4.1 transmitter_status task

transmitter_status uplinks are scheduled using the `schedule_update_request` (as described in Section 3.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 3.2. The number of supported schedules of this type is 1.

Task Settings

```
{
    "type": "transmitter_status",
    "version": 1
}
```

Listing 9: JSON structure for **transmitter_status** task settings

Binary Encoding and Description

Table 4.5: Binary encoding for **transmitter_status** Task settings

Field ID	Type	Bit Index	Description
`type`	`uint12`	112..123	Task type, for **transmitter_status**: 0
`version`	`uint4`	124..127	Task version: 1.

4.4.2 transmitter_status message

Message Structure

```
{
    "message_name": "transmitter_status",
    "message_version": 1,
    "transmitter_temperature": <number>,
    "rssi": <number>,
    "bist_power_supply": <boolean>,
    "bist_configuration": <boolean>,
    "bist_sensor_connection": <boolean>,
    "bist_sensor_paired": <boolean>,
    "bist_flash_memory": <boolean>,
    "bist_internal_temperature_sensor": <boolean>,
    "bist_time_synchronized": <boolean>,
    "bist_passed": <boolean>,
    "use_case": <string>,
    "protocol_major": <number>,
    "protocol_minor": <number>
}
```

Listing 10: JSON structure for **transmitter_status** message

Binary Encoding and Description

Table 4.6: Binary encoding for **transmitter_status** message sent on FPort 16

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 1.
message_version	uint4	4..7	Message version: 1.
transmitter_temperature	int8	8..15	Measured transmitter temperature in °C [°F].
rssi	int8	16..23	RSSI (Received Signal Strength Indicator) of the latest downlink.
bist_power_supply	bool	24	Passed if the power supply module is operational.
bist_configuration	bool	25	Passed if the device has all mandatory configurations.
bist_sensor_connection	bool	26	Passed if the sensor is connected to the transmitter.
bist_sensor_paired	bool	27	Passed if the sensor is paired to the transmitter. See require_sensor_pairing in Section 4.7.1.
bist_flash_memory	bool	28	Passed if the flash memory is operational.
bist_internal_temperature_sensor	bool	29	Passed if the temperature of the microcontroller unit is within the acceptable range.

Continues on next page

Table 4.6 Continued: Binary encoding for **transmitter_status** message sent on FPort 16

Field ID	Type	Bit Index	Description
bist_time_synchronized	bool	30	Passed if the device clock is considered to have a valid time. The time is considered valid if it has been synchronized with the server at least once in the last three time synchronization intervals. After a reboot the time is considered invalid until the first time synchronization. If the device time is invalid, all tasks are scheduled with a randomized offset, and uplinks with a timestamp will indicate that the device time is invalid. In that case the server can use the message reception time as the timestamp.
bist_passed			Passed if all other BIST passed.
Reserved for Future Use	uint1	31	
use_case	uint8	32..39	Use case of the device. For value mapping see the table below.
protocol_major	uint4	40..43	Major version of the communication protocol implemented by the device. For the current revision of the communication protocol the major version is 1.
protocol_minor	uint4	44..47	Minor version of the communication protocol implemented by the device. For the current revision of the communication protocol the minor version is 0.

Enum Value Mappings

Table 4.7: Value mapping of **use_case_v0**

Value mapping	Value
"STS"	2

4.4.3 Example

The device is scheduled by default to send a `transmitter_status` uplink once per day. If no other schedule is set, the device will follow this default schedule.

```
{
    "message_name": "schedule_update_request",
    "message_version": 0,
    "slot": 0,
    "command": "replace",
    "timing": 1440,
    "triggered_on_button_press": true,
    "send": true,
    "task": {
        "type": "transmitter_status",
        "version": 1
```

```
    }
}
```

Listing 11: Example of the JSON structure for scheduling a **transmitter_status** uplink.

4.5 Battery

The energy consumption is continuously monitored to keep track of the remaining battery life. The transmitter reports these battery health statistics in the **transmitter_battery** uplink.

When the device battery is changed the **transmitter_battery_reset_request** should be used to reset the battery level. After the request is processed, the device sends a **transmitter_battery_reset_answer** indicating that the battery level has been reset.

4.5.1 transmitter_battery task

transmitter_battery uplinks are scheduled using the `schedule_update_request` (as described in Section 3.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 3.2. The number of supported schedules of this type is 1.

Task Settings

```
{
    "type": "transmitter_battery",
    "version": 1
}
```

Listing 12: JSON structure for **transmitter_battery** task settings

Binary Encoding and Description

Table 4.8: Binary encoding for **transmitter_battery** Task settings

Field ID	Type	Bit Index	Description
`type`	`uint12`	112..123	Task type, for **transmitter_battery**: 15
`version`	`uint4`	124..127	Task version: 1.

4.5.2 transmitter_battery message

Message Structure

```
{
    "message_name": "transmitter_battery",
    "message_version": 1,
    "transmitter_charge_used": <number>,
    "sensor_charge_used": <number>,
    "transmitter_average_temperature": <number>,
    "battery_level": <number>,
    "battery_voltage": <number>
}
```

Listing 13: JSON structure for **transmitter_battery** message

Binary Encoding and Description

Table 4.9: Binary encoding for **transmitter_battery** message sent on FPort 14

Field ID	Type	Bit Index	Description
`id`	`uint4`	0..3	Message ID: 1.
`message_version`	`uint4`	4..7	Message version: 1.
`transmitter_charge_used`	`pfloat15`	8..22	Amount of charge used by the transmitter in mWh.
`sensor_charge_used`	`pfloat15`	23..37	Amount of charge used by the sensor in mWh.
`transmitter_average_temperature`	`float16`	38..53	Average temperature of the device since the previous message, measured in °C [°F].
`battery_level`	`uint8`	54..61	Reported battery level at the current temperature. The binary encoding is the same as `Battery` in **DevStatusAns** defined in LoRaWAN® L2 1.0.4: 0 External power source 1..254 Remaining battery level 255 Battery level is not available
`battery_voltage`	`pfloat15`	62..76	Measured battery voltage in V.
`Reserved for Future Use`	`uint3`	77..79	

4.5.3 transmitter_battery_reset_request message

Message Structure

```
{
    "message_name": "transmitter_battery_reset_request",
    "message_version": 0,
    "magic_value": <number>
}
```

Listing 14: JSON structure for **transmitter_battery_reset_request** message

Binary Encoding and Description

Table 4.10: Binary encoding for **transmitter_battery_reset_request** message sent on FPort 14

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 3.
message_version	uint4	4..7	Message version: 0.
magic_value	uint16	8..23	A fixed value used to verify the request. Must be 43018.

4.5.4 transmitter_battery_reset_answer message

Message Structure

```
{
    "message_name": "transmitter_battery_reset_answer",
    "message_version": 0
}
```

Listing 15: JSON structure for **transmitter_battery_reset_answer** message

Binary Encoding and Description

Table 4.11: Binary encoding for **transmitter_battery_reset_answer** message sent on FPort 14

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 3.
message_version	uint4	4..7	Message version: 0.

4.6 Factory Reset

The transmitter can be reset to factory configuration using the **transmitter_factory_reset_request** downlink.

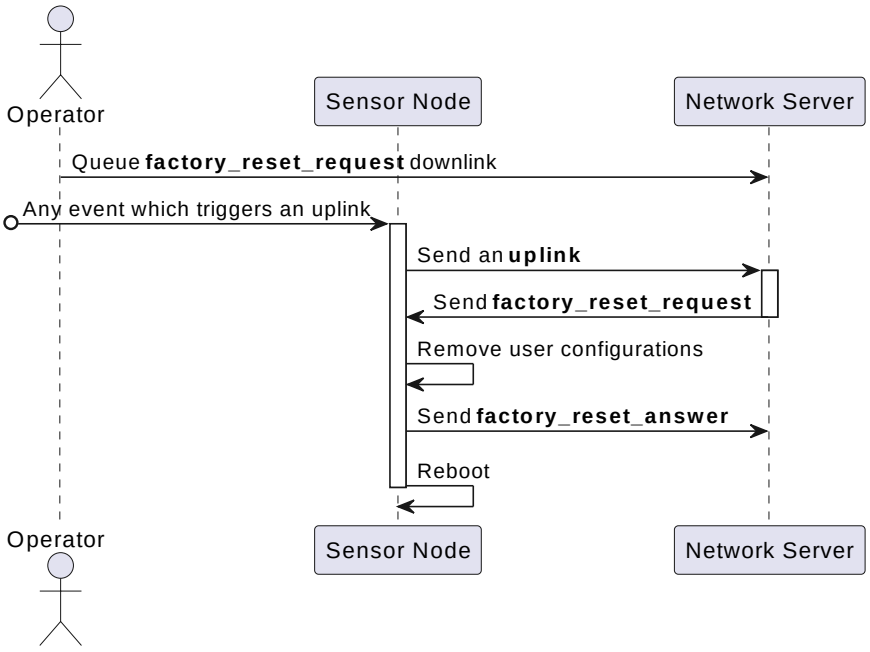

Figure 4.3: Factory Reset

Factory reset removes all user configurations and schedules. After the request is processed, the device sends a **transmitter_factory_reset_answer** indicating a successful factory reset and reboots (see Figure 4.3).

4.6.1 factory_reset_request message

Message Structure

```
{
    "message_name": "factory_reset_request",
    "message_version": 0,
    "magic_value": <number>
}
```

Listing 16: JSON structure for **factory_reset_request** message

Binary Encoding and Description

Table 4.12: Binary encoding for **factory_reset_request** message sent on FPort 14

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 2.
message_version	uint4	4..7	Message version: 0.
magic_value	uint16	8..23	A fixed value used to verify the request. Must be 39763.

The hex representation of the encoded request message is 0x209b53.

4.6.2 factory_reset_answer message

Message Structure

```
{
    "message_name": "factory_reset_answer",
    "message_version": 0
}
```

Listing 17: JSON structure for **factory_reset_answer** message

Binary Encoding and Description

Table 4.13: Binary encoding for **factory_reset_answer** message sent on FPort 14

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 2.
message_version	uint4	4..7	Message version: 0.

4.7 Configuration

The transmitter's non-sensor related behavior can be adjusted using the **transmitter** configuration.

4.7.1 transmitter configuration

Configuration Payload

```
{
    "type": "transmitter",
    "version": 0,
    "allow_deactivation": <boolean>,
    "require_sensor_pairing": <boolean>,
    "enable_class_b": <boolean>,
    "time_synchronization_interval_days": <number>,
    "fragmented_uplink_redundancy_percent": <number>
}
```

Listing 18: JSON structure for **transmitter** configuration payload

Binary Encoding and Description

Table 4.14: Binary encoding for **transmitter** configuration payload

Field ID	Type	Bit Index	Description
`type`	`uint12`	40..51	Configuration payload type for **transmitter** (encoded value: 1).
`version`	`uint4`	52..55	Configuration payload version: 0.
`allow_deactivation`	`bool`	56	Allow local deactivation procedure to be executed for either disabling the sensor or replacing it with a new sensor.
`require_sensor_pairing`	`bool`	57	If `true`, the transmitter will only communicate with the sensor attached during the activation. Changing the attached sensor requires pairing through deactivation and activation.
`enable_class_b`	`bool`	58	If `true`, Class B is enabled; otherwise, the device operates in Class A. Note that Class B increases energy consumption depending on the ping-slot periodicity configured by the LNS. This is an experimental feature.
`time_synchronization_interval_days`	`uint3`	59..61	Interval in days at which the device uses the **DeviceTimeReq** MAC command to synchronize time with the server. Setting the interval to 0 disables time synchronization.
`fragmented_uplink_redundancy_percent`	`uint8`	62..69	Not applicable to this use case. This value will be ignored.
Reserved for Future Use	`uint2`	70..71	

Default

```
{
    "message_name": "configuration_update_request",
    "message_version": 0,
    "tag": "0x573889d9",
    "payload": {
        "type": "transmitter",
        "version": 0,
        "allow_deactivation": true,
        "require_sensor_pairing": false,
        "enable_class_b": false,
        "time_synchronization_interval_days": 1,
        "fragmented_uplink_redundancy_percent": 10
    }
}
```

Listing 19: Example of the JSON structure for the default transmitter configuration.

5 Sensor

The NEON Steam trap Sensor allows for condition monitoring of steam traps. The sensor measures the temperature of the condensate line, the ambient temperature and the ultrasonic activity of the steam trap. It can determine the state of the steam trap based on these measurements, and alert the user when the steam trap is in a faulty state.

Since there are many different types of steam traps, which can work under a wide variety of process conditions, the sensor has a built-in calibration mode, this mode allows the sensor to learn the normal operating conditions of the steam trap, and determine the thresholds for the alerts based on these normal operating conditions. This calibration mode takes up the duration of 1 full day. The mode is started automatically on activation of the device. When done, the sensor will transition to normal operation by itself, without any need for user interaction.

The calibration mode negates the need for the user to input the type of steam trap and the process conditions like upstream and downstream pressure and temperature. The following sections will describe the different messages that can be sent by the NEON Steam trap Sensor, and how to schedule them. For the scheduling of the messages, an online configuration tool is available at https://neon-configurator.twtg.io/neon/sts/v1.0/.

5.1 Steam Trap Measurement

The NEON Steam Trap Sensor is designed to monitor the condition of steam traps in industrial systems. It provides measurements that help in identifying steam trap performance and failures. It has the capability to report the status of the steam trap as well as key parameters related to its operation, such as temperatures and ultrasound levels.

5.1.1 sts_measurement task

Task Settings

```
{
    "type": "sts_measurement",
    "version": 0,
    "f_min": <number>,
    "f_max": <number>,
    "blocked_state_holdoff_hours": <number>,
    "enable_confirmed_message": <boolean>
}
```

Listing 20: JSON structure for **sts_measurement** task settings

Binary Encoding and Description

Table 5.1: Binary encoding for **sts_measurement** Task settings

Field ID	Type	Bit Index	Description
`type`	`uint12`	112..123	Task type, for **sts_measurement**: 25
`version`	`uint4`	124..127	Task version: 0.
`f_min`	`float32`	128..159	Lower bound of the frequency range sent in Hz. The recommended value for f_min is 25 kHz.
`f_max`	`float32`	160..191	Upper bound of the frequency range sent in Hz. The recommended value for f_max is 65 kHz.
`blocked_state_holdoff_hours`	`uint10`	192..201	Number of hours that the device will wait before reporting the blocked state. This is used to avoid reporting blocked state during startup or maintenance when the steam trap is not expected to operate.
`enable_confirmed_message`	`bool`	202	Enable confirmed messages.

5.1.2 sts_measurement message

The **sts_measurement** message contains the overall values used to assess the condition of a steam trap, measured according to the scheduled `"sts_measurement"` task.

Message Structure

```
{
    "message_name": "sts_measurement",
    "message_version": 0,
    "timestamp": <string>,
    "sts_status": <string>,
    "sts_spl_db": <number>,
    "sts_condensate_temperature": <number>,
    "sts_ambient_temperature": <number>
}
```

Listing 21: JSON structure for **sts_measurement** message

Binary Encoding and Description

Table 5.2: Binary encoding for **sts_measurement** message sent on FPort 17

Field ID	Type	Bit Index	Description
`id`	`uint4`	0..3	Message ID: 9.
`message_version`	`uint4`	4..7	Message version: 0.
`timestamp`	`short_timestamp`	8..23	UTC ISO timestamp of the measurement. Detailed information about the binary encoding is provided in Section 1.3.1.
`sts_status`	`uint3`	24..26	Status of the steam trap. For value mapping see the table below.
`sts_spl_db`	`float16`	27..42	Measured sound pressure level in dB.
`sts_condensate_temperature`	`float16`	43..58	Measured condensate temperature in °C [°F].
`sts_ambient_temperature`	`float16`	59..74	Measured ambient temperature in °C [°F].
Reserved for Future Use	`uint5`	75..79	

Enum Value Mappings

Table 5.3: Value mapping of **sts_status_v0**

Value mapping	Value
`"ok"`	0
`"leaking"`	1
`"blocked"`	2
`"steam_on_condensate_side"`	3
`"calibrating"`	4

5.2 Steam Trap Calibration

The NEON Steam trap sensor aims to be as easy to use as possible. Therefore, the user is not required to input any information about the steam trap or the process conditions. Instead, the sensor has a built-in calibration mode. The calibration mode allows the sensor to learn the normal operating conditions of the steam trap, and determine the thresholds for the alerts based on these normal operating conditions.

The calibration mode is started automatically on activation of the device, and takes up the duration of 1 full day. During the calibration mode, the sensor will respond to the same messages as in normal operation, but with all zeros for the values, and `CALIBRATING` for the state. After the calibration mode is completed, the sensor will start normal operation automatically.

Note that the advanced user has the option to restart or overwrite the calibration values at any time by sending a **sts_calibration** configuration. However, this influences the detection of the steam trap state and should therefore be handled with care and based on a thorough understanding of the process conditions.

5.2.1 sts_calibration configuration

The calibration settings are configured using the **sts_calibration** configuration.

Configuration Payload

```
{
    "type": "sts_calibration",
    "version": 0,
    "command": <string>,
    "spl_db_leaking_threshold": <number>,
    "temperature_difference_threshold_blocked": <number>,
    "f_min": <number>,
    "f_max": <number>
}
```

Listing 22: JSON structure for **sts_calibration** configuration payload

Binary Encoding and Description

Table 5.4: Binary encoding for **sts_calibration** configuration payload

Field ID	Type	Bit Index	Description
`type`	`uint12`	40..51	Configuration payload type for **sts_calibration** (encoded value: 15).
`version`	`uint4`	52..55	Configuration payload version: 0.

Continues on next page

Table 5.4 Continued: Binary encoding for **sts_calibration** configuration payload

Field ID	Type	Bit Index	Description
`command`	`uint8`	56..63	Command for the steam trap configuration. **calibrate** Start a new calibration process. The process will take 24 hours. **write_values** Set the sound pressure level threshold used for detecting a leaking steam trap in dB. For value mapping see the table below.
`spl_db_leaking_threshold`	`float16`	64..79	Sound pressure level threshold used for detecting a leaking steam trap in dB. **Note** that the device will not simply check if the measured `sts_spl_db` is above this threshold to determine if the steam trap is leaking, but it will use this threshold in combination with the other measurements and the internal statistical logic of the device to determine the state of the steam trap.
`temperature_difference_threshold_blocked`	`uint8`	80..87	Temperature difference threshold between the condensate and ambient temperature used for detecting a blocked steam trap in °C [°F].
`f_min`	`uint7`	88..94	Lower bound of the frequency range in kHz used for the calibration of the steam trap measurement. The recommended value for f_min is 25 kHz.
`f_max`	`uint7`	95..101	Upper bound of the frequency range in kHz used for the calibration of the steam trap measurement. The recommended value for f_max is 65 kHz.

Enum Value Mappings

Table 5.5: Value mapping of **sts_calibration_command_v0**

Value mapping	Value
`"calibrate"`	0
`"write_values"`	1

A Code Examples

A.1 Encoding and Decoding of Binary Data

```
var ReadCursor = (function () {
  /**
   * ReadCursor class constructor
   *
   * @param bytes The byte array to read from.
   */
  function ReadCursor(bytes) {
    this.bytes = bytes;
    // Initial conditions which ensure that a read from an empty bytes array
    // results in an error
    this.byteIndex = -1;
    this.unreadBitsInByte = 0;
  }

  /**
   * Extracts a specified number of bits
   *
   * @param count - The number of bits to extract.
   * @returns The extracted bits as an integer.
   */
  ReadCursor.prototype.readBits = function (count) {
    var result = 0;
    if (this.remainingBits() < count) {
      throw new Error('Payload is too short');
    }
    while (count > 0) {
      if (this.unreadBitsInByte <= 0) {
        this.byteIndex += 1;
        this.unreadBitsInByte = 8;
      }
      var bitsToRead = Math.min(this.unreadBitsInByte, count);
      var bitShift = this.unreadBitsInByte - bitsToRead;
      var bitMask = ((1 << bitsToRead) - 1) << bitShift;
      var bits = (this.bytes[this.byteIndex] & bitMask) >> bitShift;
      // NOTE: Math.pow is used because JS shift operations work only on 32bit integers
      result = result * Math.pow(2, bitsToRead) + bits;
      this.unreadBitsInByte -= bitsToRead;
      count -= bitsToRead;
    }
    return result;
  };

  return ReadCursor;
})();
```

Listing 23: Code example for reading bit from a byte array in JavaScript

```
var WriteCursor = (function () {
  /**
   * WriteCursor constructor, starts with an empty byte array.
   */
  function WriteCursor() {
    this.bytes = [];
    this.writtenBitsInByte = 8;
  }

  /**
   * Writes value within the specified number of bits
   */
  WriteCursor.prototype.writeBits = function (value, count) {
    // NOTE: Math.pow is used because JS shift operations work only on 32bit integers
    if (value < 0 || value >= Math.pow(2, count)) {
      throw new Error(
        'Value '
          .concat(value, ' cannot be represented with ')
          .concat(count, ' bits.')
      );
    }
    while (count > 0) {
      if (this.writtenBitsInByte === 8) {
        // Start a new byte if the current one is full
        this.bytes.push(0);
        this.writtenBitsInByte = 0;
      }
      var bitsToWrite = Math.min(8 - this.writtenBitsInByte, count);
      var bitMask = (1 << bitsToWrite) - 1;
      var bits = (value >> (count - bitsToWrite)) & bitMask;
      this.bytes[this.bytes.length - 1] |=
        bits << (8 - (this.writtenBitsInByte + bitsToWrite));
      this.writtenBitsInByte += bitsToWrite;
      count -= bitsToWrite;
    }
  };

  return WriteCursor;
})();
```

Listing 24: Code example for writing bit to a byte array in JavaScript

A.2 Encoding and Decoding of float32

```
/**
 * Converts a float value into a IEEE 754 32 bit number
 *
 * @param {WriteCursor} cursor Instance of WriteCursor to write the message
 * @param {Number} value the float to encode
 * @returns the 32 bit representing the IEEE 754 float
 */
function encodeFloat32(cursor, value) {
  if (typeof value !== 'number') {
    throw new Error(
      'Invalid float32 value: '.concat(value, ', must be a number')
    );
  }
  var encoded = 0;
  if (isNaN(value)) {
    encoded = 0x7fc00000;
  } else if (value === Number.POSITIVE_INFINITY) {
    encoded = 0x7f800000;
  } else if (value === Number.NEGATIVE_INFINITY) {
    encoded = 0xff800000;
  } else if (value === 0 && 1 / value === -Infinity) {
    // Above is equivalent of Object.is(value, -0) which is not supported in ES5
    encoded = 0x80000000;
  } else if (value === 0) {
    encoded = 0x00000000;
  } else {
    var sign = value < 0 ? 0x80000000 : 0;
    value = Math.abs(value);
    if (value > 3.4028234663852886e38) {
      encoded = sign + 0x7f80000; // Overflow -> Infinity
    } else if (value < 1.401298464324817e-45) {
      encoded = sign + 0x00000001; // Underflow -> smallest subnormal
    } else if (value <= 1.1754942106924411e-38) {
      // subnormal
      var mantissa = Math.round(value * Math.pow(2, 149));
      encoded = sign + (mantissa & 0x7fffff);
    } else {
      // normal
      var valueLog = log2(value);
      var exponent = Math.min(Math.floor(valueLog), 127);
      var mantissa = Math.min(
        Math.round((Math.pow(2, valueLog - exponent) - 1) * 0x800000),
        0x7fffff
      );
      encoded = sign + ((exponent + 127) << 23) + mantissa;
    }
  }
  cursor.writeBits(encoded, 32);
}
```

Listing 25: Code example for encoding a float32 in JavaScript

```
/**
 * Decodes a 32-bit floating point number from number
 *
 * @param {ReadCursor} cursor Instance of ReadCursor containing the received message
 * @returns a Number
 */
function decodeFloat32(cursor) {
  var encoded = cursor.readBits(32);
  var sign = encoded & 0x80000000 ? -1 : 1;
  var exp = (encoded & 0x7f800000) >> 23;
  var fraction = encoded & 0x7fffff;
  if (exp === 0) {
    return sign * 1.401298464324817e-45 * fraction; // 2^-149
  }
  if (exp === 255) {
    return fraction ? NaN : sign * Infinity;
  }
  return sign * Math.pow(2, exp - 127) * (1 + fraction * 1.1920928955078125e-7); // 2^-23
}
```

Listing 26: Code example for decoding a float32 in JavaScript

A.3 Encoding and Decoding of float16 and pfloat15

```
/**
 * Converts a float value into a IEEE 754 16 bit number
 *
 * @param value the float to encode
 * @returns the 16 bit representing the IEEE 754 float
 */
function numberToFloat16(value) {
  var encoded = 0;
  if (isNaN(value)) {
    encoded = 0x7e00;
  } else if (value === Number.POSITIVE_INFINITY) {
    encoded = 0x7c00;
  } else if (value === Number.NEGATIVE_INFINITY) {
    encoded = 0xfc00;
  } else if (value === 0 && 1 / value === -Infinity) {
    // Above is equivalent of Object.is(value, -0) which is not supported in ES5
    encoded = 0x8000;
  } else if (value === 0) {
    encoded = 0x0000;
  } else {
    var sign = value < 0 ? 0x8000 : 0x0000;
    value = Math.abs(value);
    if (value > 65504) {
      encoded = sign | 0x7cff; // Overflow
    } else if (value < 5.960464477539063e-8) {
      encoded = sign | 0x0001; // Underflow
    } else if (value <= 0.00006097555160522461) {
      // subnormal
      var mantissa = Math.round(value * Math.pow(2, 24));
      encoded = sign | (mantissa & 0x3ff);
    } else {
      // normal
      var valueLog = log2(value);
      var exponent = Math.min(Math.floor(valueLog), 15);
      var mantissa = Math.min(
        Math.round((Math.pow(2, valueLog - exponent) - 1) * 0x400),
        0x3ff
      );
      encoded = sign | ((exponent + 15) << 10) | mantissa;
    }
  }
  return encoded;
}

/**
 * Writes a 16 bit floating point number to the message
 *
 * @param {WriteCursor} cursor Instance of WriteCursor to write the message
 * @param {Number} value The float to write
 */
function encodeFloat16(cursor, value) {
  if (typeof value !== 'number') {
    throw new Error(
      'Invalid float16 value: '.concat(value, ', must be a number')
    );
  }
  var encoded = numberToFloat16(value);
  cursor.writeBits(encoded, 16);
}
```

```
/**
 * Writes a 15 bit floating point number to the message
 *
 * @param {WriteCursor} cursor Instance of WriteCursor to write the message
 * @param {Number} value The float to write (positive only)
 */
function encodePFloat15(cursor, value) {
  if (typeof value !== 'number') {
    throw new Error(
      'Invalid pfloat15 value: '.concat(value, ', must be a number')
    );
  }
  var encoded = numberToFloat16(value);
  cursor.writeBits(encoded, 15);
}
```

Listing 27: Code example for encoding a float16 and pfloat15 in JavaScript

```
/**
 * Decodes a 16-bit floating point number from number
 *
 * @param {Number} encoded a number with the binary representation of a 16-bit floating point number
 */
function float16ToNumber(encoded) {
  var sign = encoded & 0x8000 ? -1 : 1;
  var exp = (encoded & 0x7c00) >> 10;
  var fraction = encoded & 0x03ff;
  if (exp === 0) {
    return sign * 5.960464477539063e-8 * fraction; // 2^-24
  }
  if (exp === 31) {
    return fraction ? NaN : sign * Infinity;
  }
  return sign * Math.pow(2, exp - 15) * (1 + fraction * 0.0009765625); // 2^-10
}

/**
 * Reads a 16-bit floating point number from the message
 *
 * @param {ReadCursor} cursor Instance of ReadCursor containing the received message
 * @returns a Number
 */
function decodeFloat16(cursor) {
  var encoded = cursor.readBits(16);
  return float16ToNumber(encoded);
}

/**
 * Reads a 15-bit floating point number (without the sign bit) from the message
 * 15 bit float can only represent positive numbers
 *
 * @param {ReadCursor} cursor Instance of ReadCursor containing the received message
 * @returns a Number
 */
function decodePFloat15(cursor) {
  var encoded = cursor.readBits(15); // the 16th bit is the sign bit, which is kept zero
  return float16ToNumber(encoded);
}
```

Listing 28: Code example for decoding a float16 and pfloat15 in JavaScript

A.4 Decoding of short_timestamp

```
/**
 * Reads a short timestamp from the message and converts it to a full timestamp
 *
 * @param {ReadCursor} readCursor Instance of ReadCursor containing the received message
 * @param {Date} recvTime Date object when the message was received
 * @returns an ISO formatted timestamp
 */
function decodeShortTimestamp(cursor, recvTime) {
  var shortTimestamp = cursor.readBits(16);
  var minutesWrap = 0xffff; // 16 bit minus 1, the last value is used to indicate no valid time
  var recvMin = Math.floor(recvTime.getTime() / (1000 * 60));
  if (shortTimestamp === minutesWrap) {
    // no valid time, use the receive time instead
    return recvTime.toISOString();
  }
  var wrapBase = Math.floor(recvMin / minutesWrap) * minutesWrap;
  var wrapOffset = recvMin - wrapBase;
  var gap = wrapOffset - shortTimestamp;
  if (gap < 0) {
    // in the future, go wrap lower
    wrapBase -= minutesWrap;
    gap += minutesWrap;
  }
  if (gap > 30240) {
    // > 21 days in minutes
    throw new Error('Timestamp gap too big: '.concat(gap, ' minutes'));
  }
  var msgMinutes = wrapBase + shortTimestamp;
  return new Date(msgMinutes * 60 * 1000).toISOString();
}
```

Listing 29: Code example for converting a short_timestamp to an absolute timestamp in JavaScript

A.5 Encoding of cron expression

```
function parseCronExpression(cron) {
  var isNumber = /^(\d+)$/;
  var isNumberOrStar = /^(\d+|\*)$/;
  var start = '';
  var end = '';
  var step = '';
  var rangestr = '';
  var parts = cron.split('/');
  if (parts.length === 1) {
    rangestr = parts[0];
  } else if (parts.length === 2 && isNumber.test(parts[1])) {
    rangestr = parts[0];
    step = parts[1];
  }
  var range_parts = rangestr.split('-');
  if (range_parts.length === 1 && isNumberOrStar.test(range_parts[0])) {
    start = range_parts[0];
  } else if (
    range_parts.length === 2 &&
    isNumber.test(range_parts[0]) &&
    isNumber.test(range_parts[1])
  ) {
    start = range_parts[0];
    end = range_parts[1];
  }
  return { start: start, end: end, step: step };
}

function fillArray(arr, value) {
  for (var i = 0; i < arr.length; i++) {
    arr[i] = value;
  }
}

/**
 * Converts a cron string value for the week days into a number where bit x corresponds to day x of
    the week
 *
 * @param {WriteCursor} cursor Instance of WriteCursor to write the message
 * @param {string} value cron string eg `3`, `3-5`, `*`, `1-4/2`
 */
function encodeCronWeekDays(cursor, value) {
  var matches = value.split(/,/);
  var encodedValue = Array(7);
  fillArray(encodedValue, 0);
  for (var _i = 0, matches_1 = matches; _i < matches_1.length; _i++) {
    var match = matches_1[_i];
    var _a = parseCronExpression(match),
      start = _a.start,
      end = _a.end,
      step = _a.step;
    if (!start) {
      throw new Error('Invalid cron week days format: ' + match);
    } else if (start === '*') {
      if (step) {
        var stepValue = parseInt(step, 10);
        if (stepValue < 1 || stepValue > 6) {
          throw new Error('Invalid cron week days format: ' + match);
        }
        for (var i = 0; i < 7; i += stepValue) {
```

```
          encodedValue[i] = 1;
        }
      } else {
        fillArray(encodedValue, 1);
      }
    } else {
      var startDay = parseInt(start || '0', 10);
      var endDay = end === '*' ? 6 : parseInt(end || start || '0', 10);
      var stepValue = parseInt(step || '1', 10);
      for (var i = startDay; i <= endDay; i += stepValue) {
        if (i >= 0 && i < 7) {
          encodedValue[i] = 1;
        } else {
          throw new Error('Invalid cron week days format: ' + match);
        }
      }
    }
  }
  // Convert the array of bits to an 7-bit number.
  var encodedNumber = parseInt(encodedValue.reverse().join(''), 2);
  cursor.writeBits(encodedNumber, 7);
}

/**
 * Converts a cron string value for the week days into a number where bit x corresponds to day x of
    the week
 *
 * @param {WriteCursor} cursor Instance of WriteCursor to write the message
 * @param {string} value cron string eg `3`, `3-5`, `*`, `1-4/2`
 */
function encodeCronHours(cursor, value) {
  var matches = value.split(/,/);
  var encodedValue = Array(24);
  fillArray(encodedValue, 0);
  for (var _i = 0, matches_2 = matches; _i < matches_2.length; _i++) {
    var match = matches_2[_i];
    var _a = parseCronExpression(match),
      start = _a.start,
      end = _a.end,
      step = _a.step;
    if (!start) {
      throw new Error('Invalid cron hours format: ' + match);
    } else if (start === '*') {
      if (step) {
        var stepValue = parseInt(step, 10);
        if (stepValue < 1 || stepValue > 23) {
          throw new Error('Invalid cron hours format: ' + match);
        }
        for (var i = 0; i < 24; i += stepValue) {
          encodedValue[i] = 1;
        }
      } else {
        fillArray(encodedValue, 1);
      }
    } else {
      var startHour = parseInt(start || '0', 10);
      var endHour = end === '*' ? 23 : parseInt(end || start || '0', 10);
      var stepValue = parseInt(step || '1', 10);
      for (var i = startHour; i <= endHour; i += stepValue) {
        if (i >= 0 && i < 24) {
          encodedValue[i] = 1;
```

```
        } else {
          throw new Error('Invalid cron hours format: ' + match);
        }
      }
    }
  }
  // Convert the array of bits to a single 24-bit number.
  var encodedNumber = parseInt(encodedValue.reverse().join(''), 2);
  cursor.writeBits(encodedNumber, 24);
}

/**
 * Encodes a cron string representing minutes into an array of numbers where each bit represents one
    minute
 *
 * @param {WriteCursor} cursor Instance of WriteCursor to write the message
 * @param {string} value cron string such as '3', '4-23', '2-33/3', '*'
 */
function encodeCronMinutes(cursor, value) {
  var matches = value.split(/,/);
  var encodedValue = Array(60);
  for (var _i = 0, matches_3 = matches; _i < matches_3.length; _i++) {
    var match = matches_3[_i];
    var _a = parseCronExpression(match),
      start = _a.start,
      end = _a.end,
      step = _a.step;
    if (!start) {
      throw new Error('Invalid cron minutes format: ' + match);
    } else if (start === '*') {
      if (step) {
        var stepValue = parseInt(step, 10);
        if (stepValue < 1 || stepValue > 59) {
          throw new Error('Invalid cron minutes format: ' + match);
        }
        for (var i = 0; i < 60; i += stepValue) {
          encodedValue[i] = 1;
        }
      } else {
        fillArray(encodedValue, 1);
      }
    } else {
      var startMinute = parseInt(start || '0', 10);
      var endMinute = end === '*' ? 59 : parseInt(end || start || '0', 10);
      var stepValue = parseInt(step || '1', 10);
      for (var i = startMinute; i <= endMinute; i += stepValue) {
        if (i >= 0 && i < 60) {
          encodedValue[i] = 1;
        } else {
          throw new Error('Invalid cron minutes format: ' + match);
        }
      }
    }
  }
  for (var i = 59; i >= 0; i--) {
    cursor.writeBits(encodedValue[i], 1);
  }
}

/**
 * Encodes a cron string representing minutes, hours, and week days.
```

```
 *
 * @param {string} value a cron string such as: '0 1 * * 2'
 */
function encodeCron(cursor, value) {
  if (typeof value !== 'string') {
    throw new Error('Cron expression must be a string');
  }
  var matches = value.split(/ /);
  if (matches.length !== 5) {
    throw new Error('Cron expression must consist of 5 fields');
  }
  if (matches[2] !== '*') {
    throw new Error('Cron day of the month field must be "*"');
  }
  if (matches[3] !== '*') {
    throw new Error('Cron month field must be "*"');
  }
  encodeCronMinutes(cursor, matches[0]);
  encodeCronHours(cursor, matches[1]);
  encodeCronWeekDays(cursor, matches[4]);
}
```

Listing 30: Code example for encoding a cron expression in JavaScript

A.6 Encoding of timing

```
/**
 * Writes a 92 bit timing value to the message
 *
 * @param {WriteCursor} cursor Instance of WriteCursor to write the message
 * @param {Number} value The timing to write, either a cron expression or a number
 */
function encodeTiming(cursor, value) {
  if (
    typeof value === 'number' ||
    (typeof value === 'string' && !isNaN(value))
  ) {
    value = Number(value);
    if (!isFinite(value) || Math.floor(value) !== value) {
      throw new Error(
        'Invalid period value: '.concat(value, ', must be an integer')
      );
    }
    cursor.writeBits(0, 1);
    cursor.writeBits(value, 15);
    cursor.writeBits(0, 32);
    cursor.writeBits(0, 32);
    cursor.writeBits(0, 12);
  } else if (typeof value === 'string') {
    cursor.writeBits(1, 1);
    encodeCron(cursor, value);
  } else {
    throw new Error(
      'Invalid timing value: '.concat(value, ', must be a string or number')
    );
  }
}
```

Listing 31: Code example for encoding timing in JavaScript

Process Automation

IMI plc
Lakeside, Solihull Parkway
Birmingham Business Park
Birmingham
B37 7XZ
United Kingdom