Process Automation

Our product brands:
IMI TWTG

# NEON Steam Trap Sensor v1

## Communication Protocol

Breakthrough engineering for a better world

**Revision** A1
**Date** 2026-06-11

# Contents

**1 Introduction** **3**
1.1 Firmware and Hardware Versions . . . 4
1.1.1 Example . . . 4
1.2 Document Content . . . 5

**2 Scheduling** **6**
2.1 Synchronized Schedules . . . 6
2.2 Unsynchronized Schedules . . . 6
2.3 Schedule Update . . . 7
2.3.1 schedule_update_request message . . . 8
2.3.2 schedule_update_answer message . . . 10

**3 Transmitter** **12**
3.1 Activation and Deactivation . . . 12
3.1.1 transmitter_deactivated message . . . 13
3.2 Button Press . . . 14
3.3 Boot . . . 15
3.3.1 transmitter_boot message . . . 15
3.4 Status . . . 16
3.4.1 transmitter_status task . . . 16
3.4.2 transmitter_status message . . . 17
3.4.3 Example . . . 18
3.5 Battery . . . 19
3.5.1 transmitter_battery task . . . 19
3.5.2 transmitter_battery message . . . 20

**4 Sensor** **21**
4.1 Steam Trap Measurement . . . 22
4.1.1 sts_measurement task . . . 22
4.1.2 sts_measurement message . . . 23

## 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 use with the JavaScript codec provided on the product support page. It will therefore not include detailed information on the binary encoding of messages. For users who wish to implement their own codec, the advanced version of this document contains the complete technical specification, including all advanced configurations and internal protocol details. It can be found on the product support page, at the same location as this document.

The JavaScript codec implements the LoRaWAN® Payload Codec API Specification TS013-1.0.0.

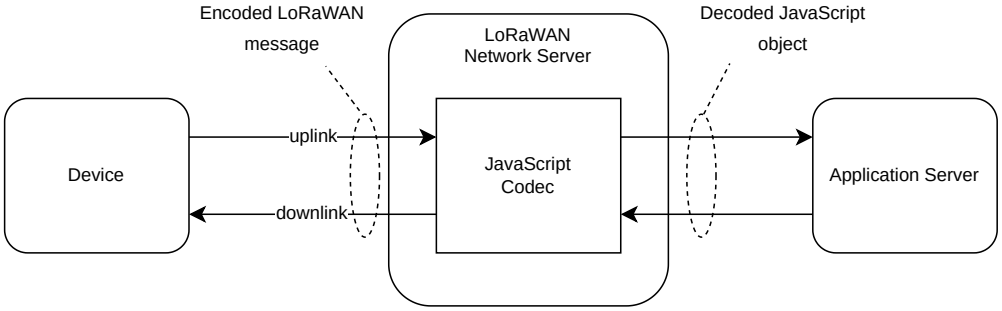


**Figure 1.1:** Encoding and decoding of LoRaWAN messages

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 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 1.1 and Table 1.2.

**Table 1.1:** Sensor Firmware Versions

| Version | Protocol Revision |
|---|---|
| 91d2 | A |

**Table 1.2:** 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 1.3:** Sensor Hardware Versions

| Version | Sensor |
|---|---|
| 0007 | DS-SX-02-00 |

**Table 1.4:** Transmitter Hardware Versions

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

### 1.1.1 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.

## 1.2 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 1.5:** Overview of the device messages and configurations

| Section | Message | Configuration |
|---|---|---|
| **Scheduling** | **schedule_update_request**<br>**schedule_update_answer** | |
| **Transmitter** | **transmitter_deactivated**<br>**transmitter_boot**<br>**transmitter_status**<br>**transmitter_battery** | |
| **Sensor** | **sts_measurement** | |

# 2 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.

## 2.1 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.

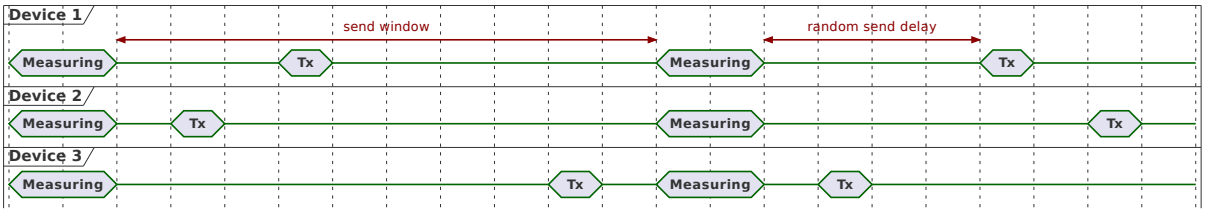


**Figure 2.1:** Synchronized Measurement

Figure 2.1 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.

## 2.2 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.

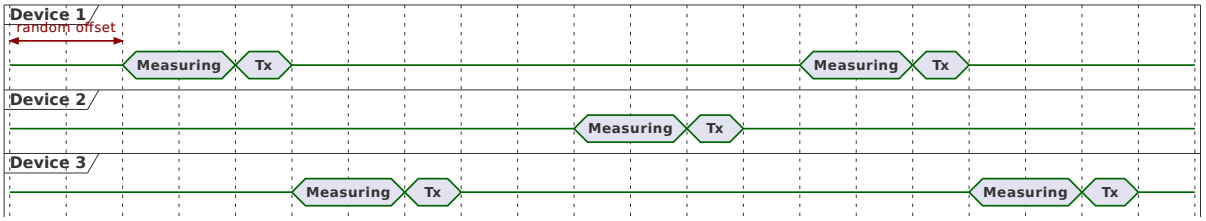


**Figure 2.2:** Unsynchronized Measurement

Figure 2.2 shows three devices configured with the same unsynchronized measurement schedule. The measurements occur at a fixed interval, with a randomized offset.

## 2.3 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 2.1.

**Table 2.1:** 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.

### 2.3.1 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 3:** JSON structure for **schedule_update_request** message

#### Binary Encoding and Description

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

| Field ID | Description |
|---|---|
| `id` | Message ID: 2. |
| `message_version` | Message version: 0. |
| `slot` | The slot used for the request. |
| `command` | Instructs the device how the schedule should be handled.<br>**set** Sets the schedule in the specified slot. Overwrites any existing schedule in the specified slot.<br>**replace** Removes all schedules of the same task type and sets this one in the specified slot.<br>**execute** Executes the schedule immediately without storing. Only the `task` field is considered.<br>**reset** Removes all schedules of the specified type, ignoring all other fields.<br>**remove** Removes the schedule from the specified slot.<br>For value mapping see the table below. |
| `timing` | Specifies when the schedule is executed, which can be defined either as a cron expression or as a period in minutes (1–32767). |
| `triggered_on_button_press` | 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` | If `true`, the scheduled task triggers a transmission. |
| *Reserved for Future Use* | |
| `task` | 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 2.1. |

### Enum Value Mappings

**Table 2.3:** Value mapping of **schedule_set_command_v0**

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

### 2.3.2 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 4:** JSON structure for **schedule_update_answer** message

#### Binary Encoding and Description

**Table 2.4:** Binary encoding for **schedule_update_answer** message sent on FPort 11

| Field ID | Description |
|---|---|
| `id` | Message ID: 2. |
| `message_version` | Message version: 0. |
| `slot` | The slot that was used in the request. |
| `task_type` | Type of the schedule sent through the **schedule_update_request**. For value mapping see the table below. |
| `status` | Status of the schedule update. For value mapping see the table below. |
| `number_of_slots` | Total number of slots available for this task. See Table 2.1. |
| `slot_0_available` | 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` | See `slot_0_available`. |
| `slot_2_available` | See `slot_0_available`. |
| `slot_3_available` | See `slot_0_available`. |
| `slot_4_available` | See `slot_0_available`. |
| `slot_5_available` | See `slot_0_available`. |
| `slot_6_available` | See `slot_0_available`. |
| `slot_7_available` | See `slot_0_available`. |

Continues on next page

**Table 2.4 Continued:** Binary encoding for **schedule_update_answer** message sent on FPort 11

| Field ID | Description |
|---|---|
| `slot_8_available` | See `slot_0_available`. |
| `slot_9_available` | See `slot_0_available`. |
| *Reserved for Future Use* | |

### Enum Value Mappings

**Table 2.5:** Value mapping of **schedule_type_v0**

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

**Table 2.6:** 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 |

# 3 Transmitter

## 3.1 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 3.1.

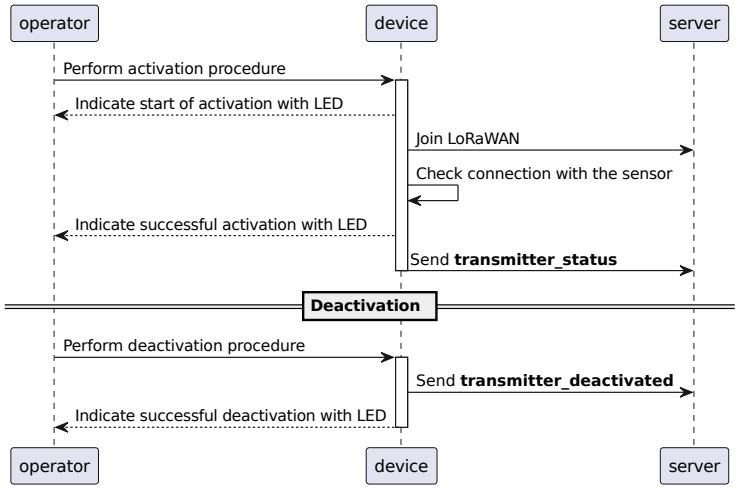


**Figure 3.1:** Transmitter Activation and Deactivation Sequence

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"`.

### 3.1.1 transmitter_deactivated message

#### Message Structure

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

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

#### Binary Encoding and Description

**Table 3.1:** Binary encoding for **transmitter_deactivated** message sent on FPort 16

| Field ID | Description |
|---|---|
| `id` | Message ID: 3. |
| `message_version` | Message version: 0. |
| `deactivation_reason` | Reason for the device's deactivation. For value mapping see the table below. |

#### Enum Value Mappings

**Table 3.2:** Value mapping of **deactivation_reason_v0**

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

## 3.2 Button Press

While the device is activated, pressing the button triggers the transmission of a **transmitter_status** message (see Section 3.4). 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 2.3.1).

The device response to a button press is shown in Figure 3.2.

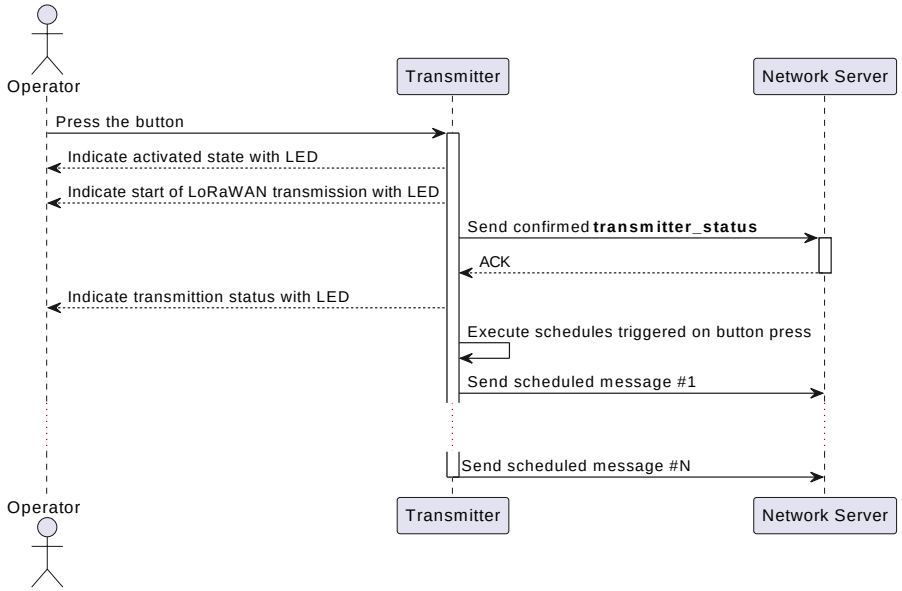


**Figure 3.2:** Transmitter Button Press Sequence

## 3.3 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.

### 3.3.1 transmitter_boot message

#### Message Structure

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

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

#### Binary Encoding and Description

**Table 3.3:** Binary encoding for **transmitter_boot** message sent on FPort 16

| Field ID | Description |
|---|---|
| `id` | Message ID: 0. |
| `message_version` | Message version: 0. |
| `transmitter_reboot_reason` | Reason for the device's reboot. For value mapping see the table below. |

#### Enum Value Mappings

**Table 3.4:** 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 |

## 3.4 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.

### 3.4.1 transmitter_status task

**transmitter_status** uplinks are scheduled using the `schedule_update_request` (as described in Section 2.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 2.2. The number of supported schedules of this type is 1.

#### Task Settings

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

**Listing 7:** JSON structure for **transmitter_status** task settings

#### Binary Encoding and Description

**Table 3.5:** Binary encoding for **transmitter_status** Task settings

| Field ID | Description |
|---|---|
| `type` | Task type, for **transmitter_status**: 0 |
| `version` | Task version: 1. |

### 3.4.2 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 8:** JSON structure for **transmitter_status** message

#### Binary Encoding and Description

**Table 3.6:** Binary encoding for **transmitter_status** message sent on FPort 16

| Field ID | Description |
|---|---|
| id | Message ID: 1. |
| message_version | Message version: 1. |
| transmitter_temperature | Measured transmitter temperature in °C [°F]. |
| rssi | RSSI (Received Signal Strength Indicator) of the latest downlink. |
| bist_power_supply | Passed if the power supply module is operational. |
| bist_configuration | Passed if the device has all mandatory configurations. |
| bist_sensor_connection | Passed if the sensor is connected to the transmitter. |
| bist_sensor_paired | Passed if the sensor is paired to the transmitter. |
| bist_flash_memory | Passed if the flash memory is operational. |
| bist_internal_temperature_sensor | Passed if the temperature of the microcontroller unit is within the acceptable range. |
| bist_time_synchronized | 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. |

Continues on next page

**Table 3.6 Continued:** Binary encoding for **transmitter_status** message sent on FPort 16

| Field ID | Description |
|---|---|
| *Reserved for Future Use* | |
| use_case | Use case of the device. For value mapping see the table below. |
| protocol_major | 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 | 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 3.7:** Value mapping of **use_case_v0**

| Value mapping | Value |
|---|---|
| "STS" | 2 |

### 3.4.3 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 9:** Example of the JSON structure for scheduling a **transmitter_status** uplink.

## 3.5 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.

### 3.5.1 transmitter_battery task

**transmitter_battery** uplinks are scheduled using the `schedule_update_request` (as described in Section 2.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 2.2. The number of supported schedules of this type is 1.

#### Task Settings

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

**Listing 10:** JSON structure for **transmitter_battery** task settings

#### Binary Encoding and Description

**Table 3.8:** Binary encoding for **transmitter_battery** Task settings

| Field ID | Description |
|---|---|
| `type` | Task type, for **transmitter_battery**: 15 |
| `version` | Task version: 1. |

### 3.5.2 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 11:** JSON structure for **transmitter_battery** message

#### Binary Encoding and Description

**Table 3.9:** Binary encoding for **transmitter_battery** message sent on FPort 14

| Field ID | Description |
|---|---|
| `id` | Message ID: 1. |
| `message_version` | Message version: 1. |
| `transmitter_charge_used` | Amount of charge used by the transmitter in mWh. |
| `sensor_charge_used` | Amount of charge used by the sensor in mWh. |
| `transmitter_average_temperature` | Average temperature of the device since the previous message, measured in °C [°F]. |
| `battery_level` | Reported battery level at the current temperature. The binary encoding is the same as `Battery` in **DevStatusAns** defined in LoRaWAN® L2 1.0.4:<br>0 External power source<br>1..254 Remaining battery level<br>255 Battery level is not available |
| `battery_voltage` | Measured battery voltage in V. |
| *Reserved for Future Use* | |

# 4 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/.

## 4.1 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.

### 4.1.1 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 12:** JSON structure for **sts_measurement** task settings

#### Binary Encoding and Description

**Table 4.1:** Binary encoding for **sts_measurement** Task settings

| Field ID | Description |
|---|---|
| `type` | Task type, for **sts_measurement**: 25 |
| `version` | Task version: 0. |
| `f_min` | Lower bound of the frequency range sent in Hz. The recommended value for f_min is 25 kHz. |
| `f_max` | Upper bound of the frequency range sent in Hz. The recommended value for f_max is 65 kHz. |
| `blocked_state_holdoff_hours` | 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` | Enable confirmed messages. |

### 4.1.2 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 13:** JSON structure for **sts_measurement** message

#### Binary Encoding and Description

**Table 4.2:** Binary encoding for **sts_measurement** message sent on FPort 17

| Field ID | Description |
|---|---|
| `id` | Message ID: 9. |
| `message_version` | Message version: 0. |
| `timestamp` | UTC ISO timestamp of the measurement. |
| `sts_status` | Status of the steam trap. For value mapping see the table below. |
| `sts_spl_db` | Measured sound pressure level in dB. |
| `sts_condensate_temperature` | Measured condensate temperature in °C [°F]. |
| `sts_ambient_temperature` | Measured ambient temperature in °C [°F]. |
| *Reserved for Future Use* | |

#### Enum Value Mappings

**Table 4.3:** Value mapping of **sts_status_v0**

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

# Process Automation



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