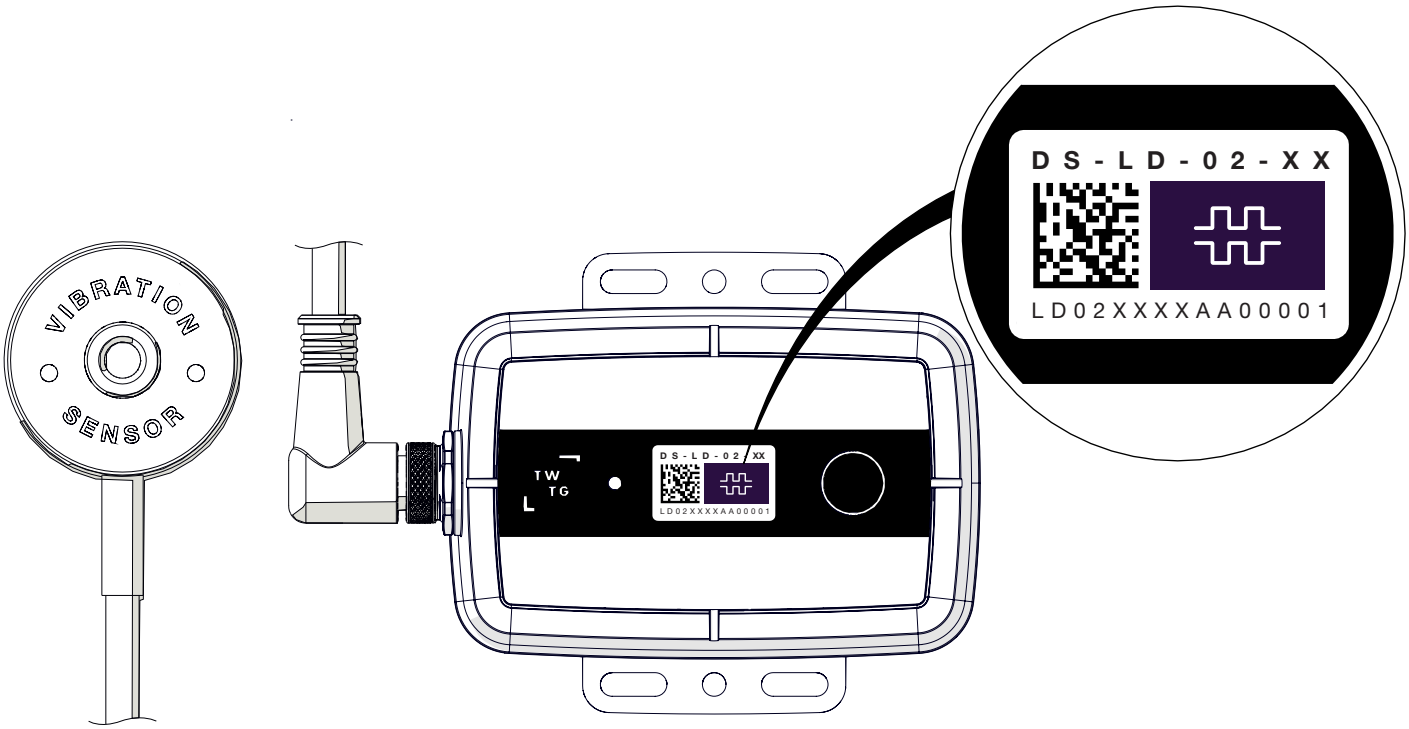
NEON Vibration Sensor

Communication Protocol v4

This document applies to:

- **DS-LD-02-00** with **DS-VB-02-00** from production batch **AF**
- **DS-LD-xx-xx** with **DS-VB-xx-xx** after firmware update



Revision	A2
Date	2024-09-05
Document	NEON_Vibration-Sensor_Communication-Protocol-v4_DS-VB-xx-xx_4003_4_A2

Contents

Revision History

Revision	Date	Description
A1	2024-04-12	VB v4 protocol beta.
A2	2024-09-05	VB v4 protocol final.

1 Introduction

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

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.

The binary encoding and usage of each message is described in the relevant sections. A JavaScript codec implementation of this protocol is available on the product support page. If it is used, the binary encoding information can be ignored.

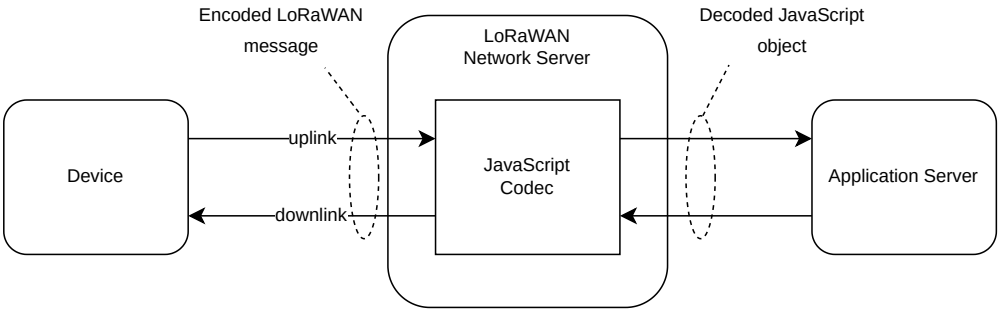


Figure 1.1: Encoding and decoding of LoRaWAN messages

The device and codec implement the following LoRaWAN specifications:

- LoRaWAN© L2 1.0.4 (TS001-1.0.4)
- RP002-1.0.3 LoRaWAN© Regional Parameters
- LoRaWAN© Payload Codec API Specification TS013-1.0.0

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.

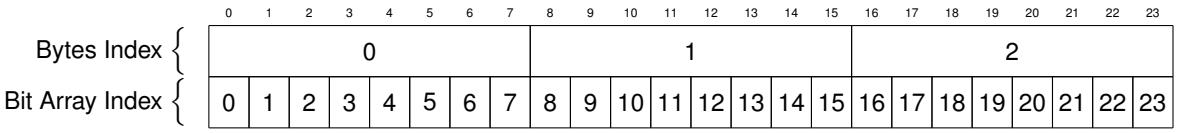


Figure 1.2: Mapping of Byte Index to Bit Index

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 1.2.

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 1.3 shows how bit groups of varying sizes are read in succession.

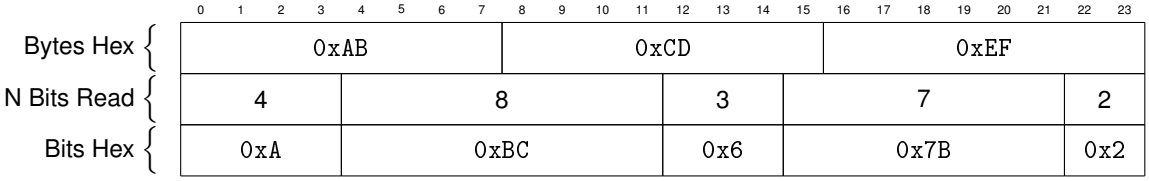


Figure 1.3: Bit Reading Example

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 follow a similar structure. The first 8 bits of the message payload is the header. It consists of the **Message ID** and **Message Version**. They are used together with the FPort to identify a message uniquely. The **Message Version** is locked within the overall **Communication Protocol Version**, which is the version of this document (v4). The remainder of the message are the data fields.

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

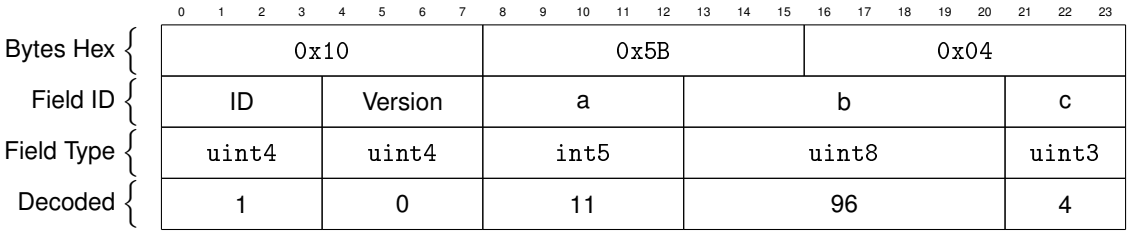


Figure 1.4: Message Decoding Example

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 1.5. As specified in TS013-1.0.0, the receive time of the message (`recvTime`) must be provided for successful decoding.

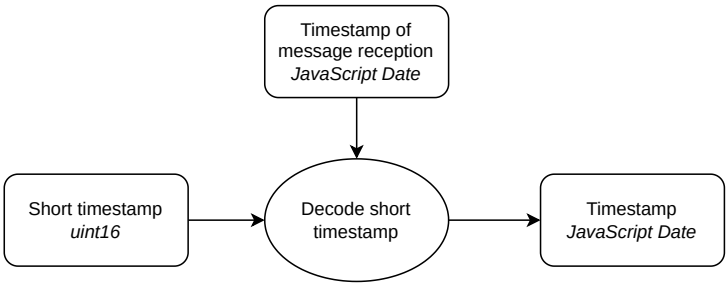


Figure 1.5: Using receive time to decode short timestamp

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 45.
  - 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 Cron Expression

Actions that the device should perform can be configured via schedules. This may be configured using the `cron` expression.

`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 schedules.

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/.

Implementation restrictions

The device does not support the `<day-of-month>` and `<month>` fields, those should be always set to "*".

Examples

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

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

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

Binary encoding

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

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

The cron expression can be encoded following the code as listed in Appendix A.5.

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

1.3.3 Timing

Device actions can be configured through schedules, with the `timing` type determining when each schedule is executed.

Figure 1.6: 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 1.6). 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.

The `timing` type can be encoded following the code as listed in Appendix A.6.

Examples

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

Example 2 `"0x800100020004000 81ff803e"` Cron trigger of `"*/15 8-17 * * 1-5"` (every 15 minutes between 8:00 AM and 5:00 PM on weekdays)

1.3.4 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 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 1.2: Overview of the device messages and configurations

Section	Message	Configuration
Fragmented Uplinks	fragmented_uplink_start fragmented_uplink_data fragmented_uplink_stop	
Configuration	configuration_update_request configuration_update_answer	
Scheduling		schedule
Transmitter	transmitter_deactivated transmitter_boot transmitter_status transmitter_battery transmitter_battery_reset_request transmitter_battery_reset_answer factory_reset_request	transmitter
Sensor	measurement past_measurement_request statistics spectrum machine_fault_indicator alert sensor_boot	vb_asset vb_alert vb_spectrum_alert

2 Fragmented Uplinks

LoRaWAN restricts the maximum uplink size, ranging from 11 to 242 bytes, depending on the region and data rate. To accommodate this, most uplinks in this document are limited to 11 bytes. Larger uplinks are split into multiple fragments and transmitted using the Fragmented Uplinks layer, as illustrated in Figure 2.1.

Features of the fragmented uplink are:

- Designed to transport large uplinks over the LoRaWAN network transparently.
- Complies with the actual LoRaWAN message size restrictions, depending on the active data rate.
- Fragments are sent over a dedicated fragmented uplink FPort.
- The reconstructed uplink includes the original FPort.
- Forward error correction is used to ensure that the complete uplink can be reconstructed, even if some of the fragments are lost.
- It can handle up to 32 kbyte uplinks.
- Integrity check of the reconstructed uplink is performed using CRC32.

Note that sending large uplinks causes more airtime and thus energy usage from the battery. Therefore, large uplinks should be used with care.

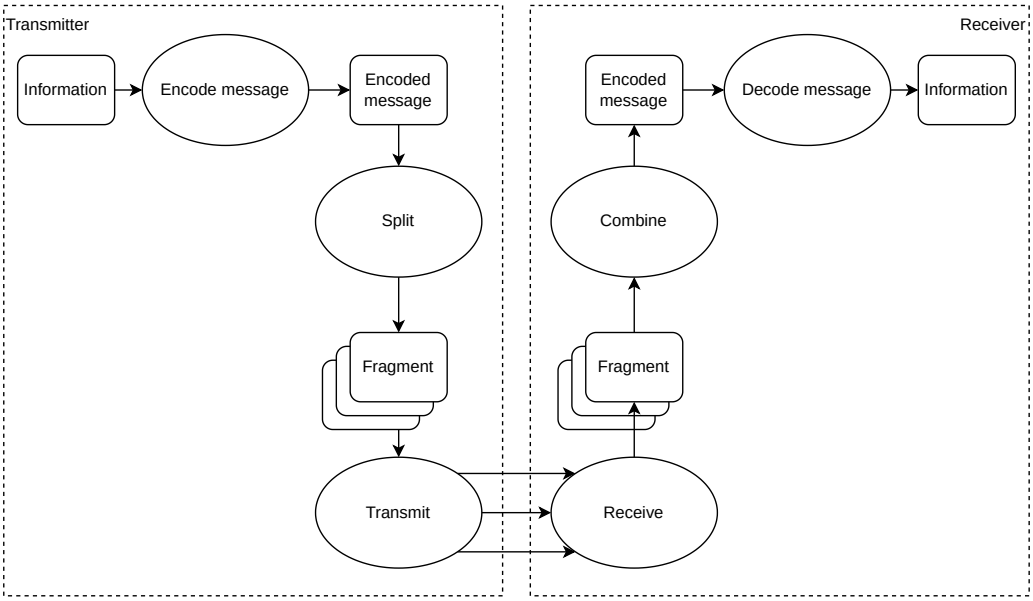


Figure 2.1: Illustration of the Fragmented Uplinks Layer Concept

2.1 Fragmented Uplink Session

On the *Transmitter* side, the *fragmentation handler transmitter* splits up the large uplinks into smaller fragments and sends them to the network server as depicted in Figure 2.2. On the *Receiver* side, the fragments must be reconstructed by a *fragmentation handler receiver*. When the large uplink is reconstructed the *fragmentation handler receiver* injects the large uplink into the application server as a regular uplink on the original FPort.

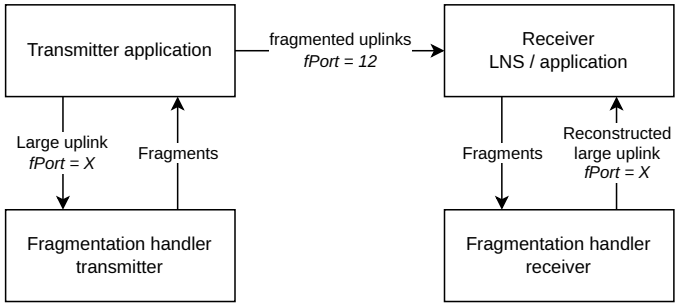


Figure 2.2: Fragmented uplink data flow

The sequence of a fragmented uplink session is shown in Figure 2.3. The *Transmitter* starts by sending a **fragmented_uplink_start** uplink to the *Receiver*. It contains information for the *fragmentation handler receiver* to start a fragmentation session: the FPort of the large uplink, its size, the size of a single fragment, and the CRC of the original message. The **fragmented_uplink_start** uplink is sent confirmed, with retries, to guarantee that it has been received.

When the **fragmented_uplink_start** uplink is acknowledged by the *Receiver*, the *Transmitter* sends the large uplink fragments with the **fragmented_uplink_data** uplinks. A single **fragmented_uplink_data** may contain multiple fragments. The fragments are sent in order, starting with the plain fragments followed by the redundancy fragments.

When the *fragmentation handler receiver* receives enough fragments to reconstruct the large uplink, it may send a **fragmented_uplink_stop** downlink to the *Transmitter*. Upon reception, the *Transmitter* stops sending the remaining **fragmented_uplink_data** uplinks for this session. If all redundancy fragments are sent, but the *fragmentation handler receiver* did not receive enough fragments, the reconstruction will fail. The amount of redundancy fragments is configurable in the **transmitter** configuration, as described in Section 5.7.

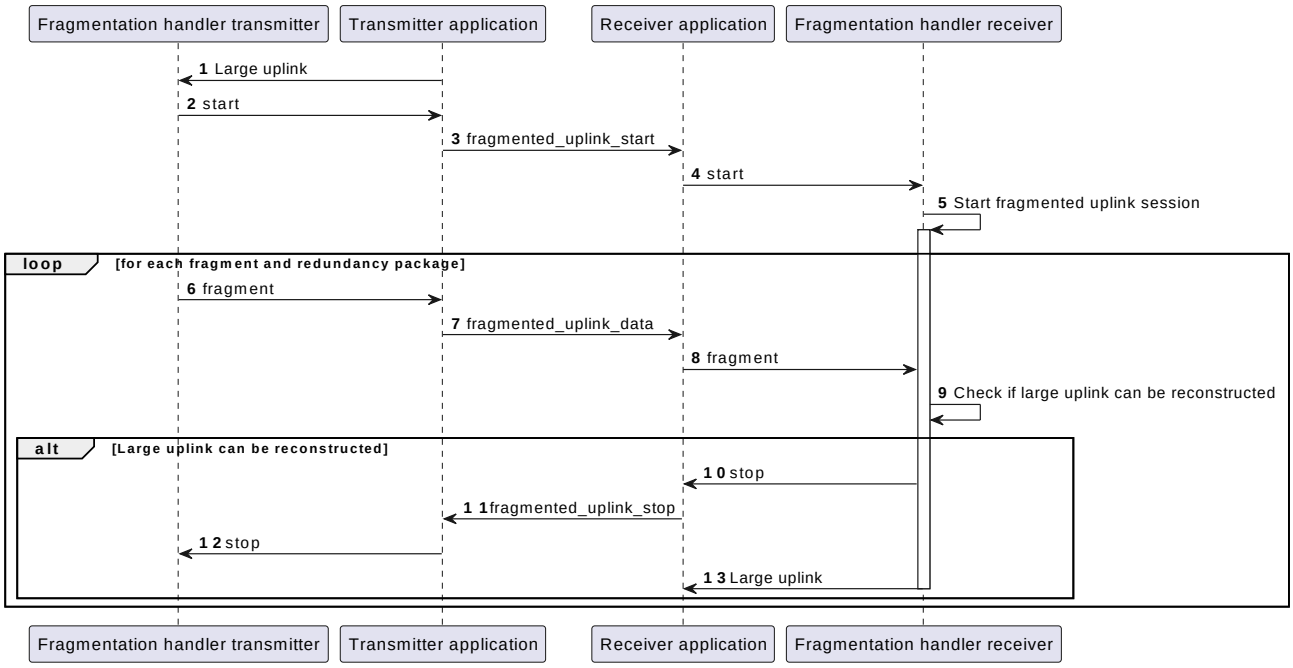


Figure 2.3: Fragmented uplink sequence

2.2 Uplink Reconstruction

The *Fragmented Uplinks* layer uses the same fragmentation algorithm as LoRaWAN© Fragmented Data Block Transport Specification TS004-2.0.0. The specification document explains the algorithm in its appendix: **Appendix: Data Block Fragmentation Forward Error Correction Code**. Note that only the fragmentation algorithm is used, the messages for the large uplink fragmentation are defined in this document.

The sending of fragments can be split into two stages (shown in Figure 2.4). First, the plain uplink fragments are sent. These fragments contain the whole original uplink. In the second stage, the redundancy fragments are sent.

In case all plain fragments are received, the receiver can reconstruct the original message directly by concatenating the plain fragments and cutting off the padding (based on the uplink size received in **fragmented_uplink_start**).

Plain fragments												Redundancy fragments						
1	2	3	4	5	6	7	8	9	10	11	12	R1	R2	R3	R4	R5	R6	R7

Figure 2.4: Concatenation of fragments

In case some of the plain fragments are lost, the receiver can use the consecutive redundancy fragments to perform forward error correction (FEC) to recover the lost fragments. When all lost fragments are recovered, it can reconstruct the original message as in the first case.

2.2.1 Integrity check of reconstructed message

If the fragmentation handling is correctly implemented, the reconstructed message should always be the same as the original message. Due to implementation differences or mixing up fragmentation sessions (e.g. when the service is temporarily unavailable), the reconstructed message might be different. The receiver can check the integrity of the reconstructed message by comparing the CRC of the reconstructed message with the CRC received in the `fragmentation_start` message.

The CRC used is the CRC-32-IEEE, which is the default in Typescript/Javascript (`node:crc`) and Python (`binascii/zlib`) libraries. In Listing 1 CRC calculation and verification of the reconstructed message is shown. An example of custom CRC calculation in Javascript is shown in Appendix A.7.

```
var crc = require('crc'); // npm install crc, or use custom implementation

var crc32Calculated = crc(recontructedMessage);
var crc32Received = fragmentation_start.crc;

if (crc32Calculated === crc32Received) {
  console.log('The reconstructed message is correct');
} else {
  console.log('The reconstructed message is incorrect and should be discarded');
}
```

Listing 1: CRC calculation and verification of the reconstructed message

2.3 fragmented_uplink_start message

Message Structure

```
{
    "fragmented_uplink_start": {
        "version": 0,
        "fport": <number>,
        "uplink_size": <number>,
        "fragment_size": <number>,
        "crc": <number>
    }
}
```

Listing 2: JSON structure for **fragmented_uplink_start** message

Binary Encoding and Description

Table 2.1: Binary encoding for **fragmented_uplink_start** message sent on FPort 12

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID for **fragmented_uplink_start** (encoded value: 0).
version	uint4	4..7	Message version: 0.
fport	uint8	8..15	FPort of the message being fragmented. It is used to label the original FPort to the reconstructed message.
uplink_size	uint16	16..31	Size of the original uplink message that has been fragmented. The reconstructed message will have the same size.
fragment_size	uint8	32..39	Payload size of each fragment in this fragmented uplink session.
crc	uint32	40..71	CRC of the original uplink message that has been fragmented, the reconstructed message will have the same CRC. It is used to verify the integrity of the reconstructed message.

2.4 fragmented_uplink_data message

Message Structure

```
{
    "fragmented_uplink_data": {
        "version": 0,
        "index": <number>,
        "data": <array of numbers>
    }
}
```

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

Binary Encoding and Description

Table 2.2: Binary encoding for **fragmented_uplink_data** message sent on FPort 12

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **fragmented_uplink_data** (encoded value: 1).
version	`uint4`	4..7	Message version: 0.
index	`uint16`	8..23	Index of the first fragment in the `data` array.
data	`uint8[]`	24..	Payload containing one or more fragments. Each fragment is of size `fragment_size` as reported in **fragmented_uplink_start**. The size of this array is a multiple of `fragment_size`. The number of fragments in a message depends on the network condition of the transmitter. The receiver's fragmentation handler needs to iterate over this array for each fragment.

2.5 fragmented_uplink_stop message

Message Structure

```
{
    "fragmented_uplink_stop": {
        "version": 0
    }
}
```

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

Binary Encoding and Description

Table 2.3: Binary encoding for **fragmented_uplink_stop** message sent on FPort 12

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **fragmented_uplink_stop** (encoded value: 2).
version	`uint4`	4..7	Message version: 0.

3 Configuration

3.1 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.

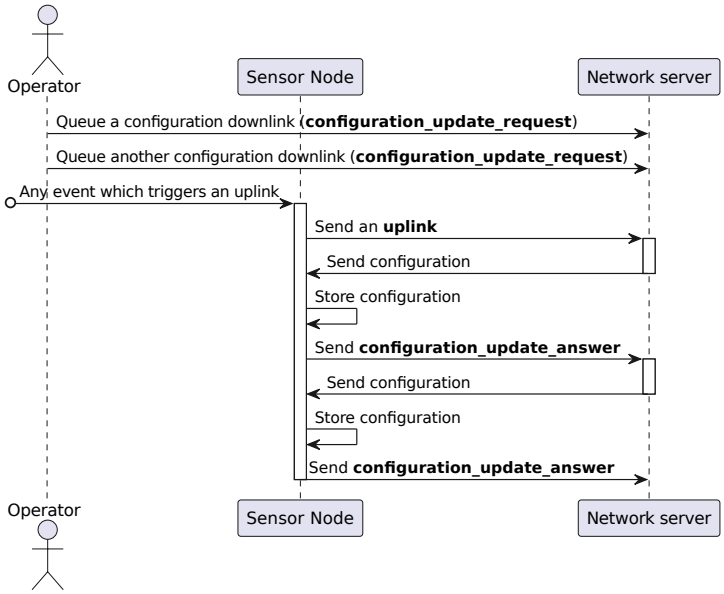


Figure 3.1: Update of Multiple Configurations

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 3.1. 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/

3.1.1 configuration_update_request message

Message Structure

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

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

Binary Encoding and Description

Table 3.1: Binary encoding for **configuration_update_request** message sent on FPort 11

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID for **configuration_update_request** (encoded value: 0).
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.

3.1.2 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

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

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

Binary Encoding and Description

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

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **configuration_update_answer** (encoded value: 0).
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**. Value mapping: `"transmitter"` → 1 `"schedule"` → 2 `"vb_alert"` → 3 `"vb_spectrum_alert"` → 4 `"vb_asset"` → 5
status	`uint4`	52..55	Status of the configuration update. Value mapping: `"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 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.

4.1 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.

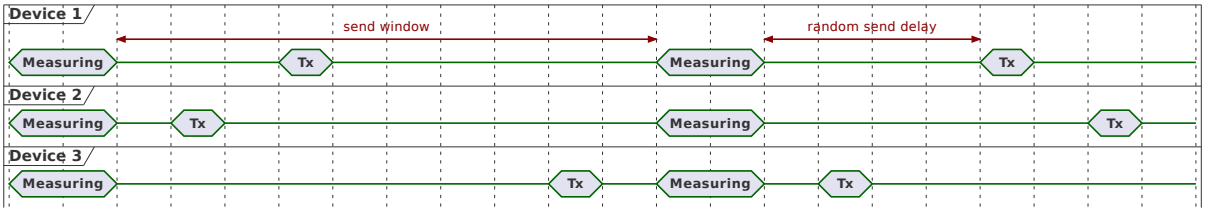


Figure 4.1: Synchronized Measurement

Figure 4.1 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.

4.2 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.

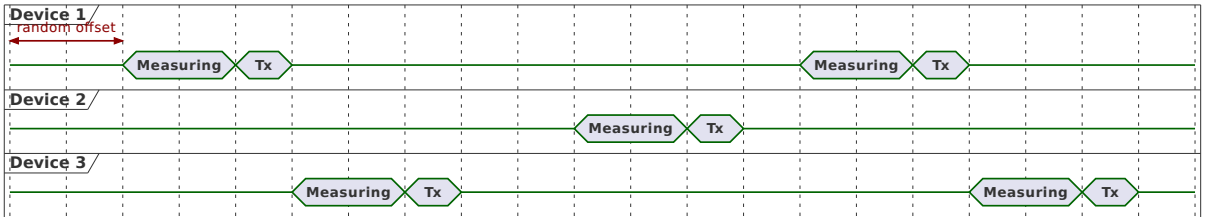


Figure 4.2: Unsynchronized Measurement

Figure 4.2 shows three devices configured with the same unsynchronized measurement schedule. The measurements occur at a fixed interval, with a randomized offset.

4.3 schedule configuration

Device task schedules are set using the **schedule** configuration (see **configuration_update_request** in Section 3.1). This configuration defines the task to be executed and its execution interval. The scheduled task is specified in the `settings` field, which at a minimum consists of the settings type and version. The settings type determines the specific task to be executed.

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.

The maximum number of supported schedules varies per settings type, as listed in Table 4.1. Additionally, there is a limit of 18 schedules that can be executed simultaneously (aligned execution).

Table 4.1: Schedule types

Schedule Type	Maximum number of schedules	Encoded Value
transmitter_status	1	0
transmitter_battery	1	15
vb_measurement	10	1
vb_machine_fault_indicator	10	2
vb_spectrum	10	3
vb_statistics_x_rms_velocity	1	5
vb_statistics_x_rms_acceleration	1	6
vb_statistics_x_peak_acceleration	1	7
vb_statistics_y_rms_velocity	1	8
vb_statistics_y_rms_acceleration	1	9
vb_statistics_y_peak_acceleration	1	10
vb_statistics_z_rms_velocity	1	11
vb_statistics_z_rms_acceleration	1	12
vb_statistics_z_peak_acceleration	1	13
vb_statistics_temperature	1	14

Configuration Payload

```
{
    "type": "schedule",
    "version": 0,
    "command": <string>,
    "timing": <string or number>,
    "triggered_on_button_press": <boolean>,
    "send": <boolean>,
    "settings": <object>
}
```

Listing 7: JSON structure for **schedule** configuration payload

Binary Encoding and Description

Table 4.2: Binary encoding for **schedule** configuration payload

Field ID	Type	Bit Index	Description
type	`uint12`	40..51	Configuration payload type for **schedule** (encoded value: 2).
version	`uint4`	52..55	Configuration payload version: 0.
command	`uint4`	56..59	Instructs the device how the schedule should be handled. **set** Sets the schedule. Overwrites any existing schedule with the same settings type and configuration tag. If the limit of schedules for the settings type is reached, configuration fails with `"rejected_schedule_type_limit"` status. **replace** Replaces all schedules with the same settings type with this one. **execute** Executes the schedule immediately without storing. Only the `settings` field is considered. **reset** Removes all schedules of the specified settings type, ignoring all other fields. **remove** Removes the schedule of the same settings type and configuration tag. Value mapping: `"set"` → 0 `"replace"` → 1 `"execute"` → 2 `"reset"` → 3 `"remove"` → 4
timing		60..151	Specifies when the schedule is executed, which can be defined either as a cron expression (see Section 1.3.2) or as a period in minutes (1–32767). Detailed information about the binary encoding is provided in Section 1.3.3.
triggered_on_button_press	`bool`	152	If `true`, the scheduled task is also triggered on a short button press by the user.
send	`bool`	153	If `true`, the scheduled task triggers a transmission.
Reserved for Future Use	`uint6`	154..159	
settings		160..	Scheduled task and its parameters. The definition varies by the settings type, as explained in the relevant sections. All tasks share a common header structure, including 12 bits for the settings type and 4 bits for the settings version. The supported task types are listed in Table 4.1.

Note that the bit index for the first field in Table 4.2 is 40 as the previous fields are defined by **configuration_update_request** (see binary encoding in Table 3.1).

5 Transmitter

5.1 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 5.1.

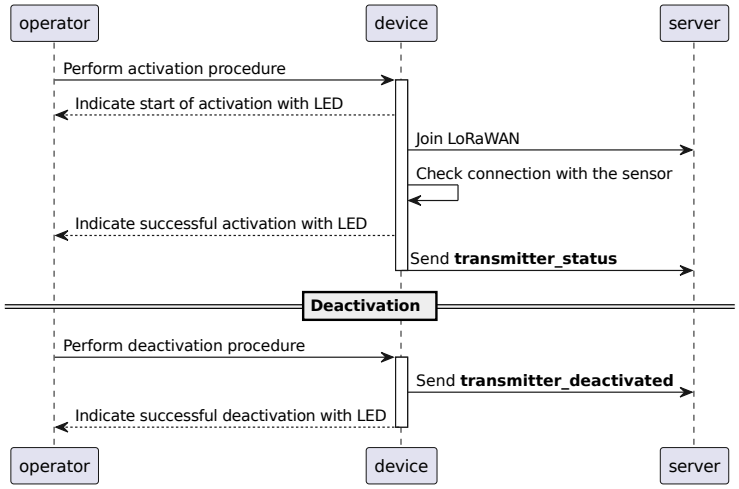


Figure 5.1: Transmitter Activation and Deactivation Sequence

The activation consists of the following steps:

1. The transmitter joins the LoRaWAN Network Server
2. The transmitter verifies connection with the attached sensor. If the verification fails, a **transmitter_deactivated** uplink is sent with the reason: `"activation_sensor_comm_fail"`.

In case one of the above steps fails, the procedure is aborted.

5.1.1 transmitter_deactivated message

Message Structure

```
{
    "transmitter_deactivated": {
        "version": 0,
        "reason": <string>
    }
}
```

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

Binary Encoding and Description

Table 5.1: Binary encoding for **transmitter_deactivated** message sent on FPort 16

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID for **transmitter_deactivated** (encoded value: 3).
version	uint4	4..7	Message version: 0.
reason	uint8	8..15	Reason for the device's deactivation. Value mapping: "user_triggered" → 0 "activation_sensor_comm_fail" → 1

5.2 Button Press

While the device is activated, pressing the button triggers the transmission of a **transmitter_status** message (see Section 5.4). 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 4.3).

The device response to a button press is shown in Figure 5.2.

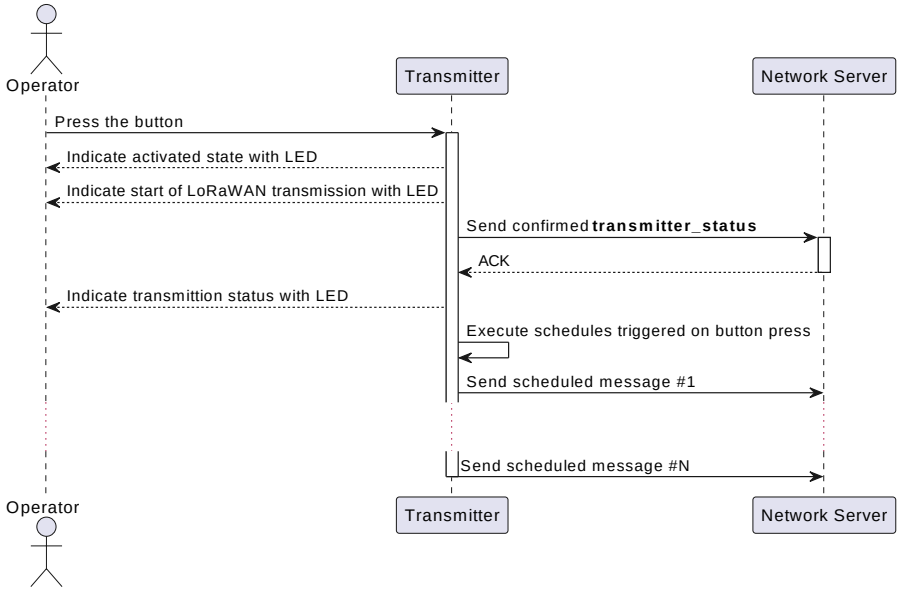


Figure 5.2: Transmitter Button Press Sequence

5.3 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.

5.3.1 transmitter_boot message

Message Structure

```
{
    "transmitter_boot": {
        "version": 0,
        "reboot_reason": <string>
    }
}
```

Listing 9: JSON structure for **transmitter_boot** message

Binary Encoding and Description

Table 5.2: Binary encoding for **transmitter_boot** message sent on FPort 16

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID for **transmitter_boot** (encoded value: 0).
version	uint4	4..7	Message version: 0.
reboot_reason	uint16	8..23	Reason for the device's reboot. Value mapping: "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

5.4 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.

5.4.1 transmitter_status schedule

transmitter_status uplinks are configured using the *Schedule Configuration* (as described in Section 4.3), with the *Schedule Configuration Settings* described below. Note that the index for the first bit is 160 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 1.

Schedule Configuration Settings

```
{
    "type": "transmitter_status",
    "version": 0
}
```

Listing 10: JSON structure for **transmitter_status** schedule configuration settings

Binary Encoding and Description

Table 5.3: Binary encoding for **transmitter_status** schedule configuration settings

Field ID	Type	Bit Index	Description
type	`uint12`	160..171	Schedule settings type for **transmitter_status** (encoded value: 0).
version	`uint4`	172..175	Schedule settings version: 0.

5.4.2 transmitter_status message

Message Structure

```
{
    "transmitter_status": {
        "version": 0,
        "temperature": <number>,
        "rssi": <number>,
        "lora_tx_counter": <number>,
        "bist": {
            "power_supply": <boolean>,
            "configuration": <boolean>,
            "sensor_connection": <boolean>,
            "sensor_paired": <boolean>,
            "flash_memory": <boolean>,
            "internal_temperature_sensor": <boolean>,
            "time_synchronized": <boolean>
        }
    }
}
```

Listing 11: JSON structure for **transmitter_status** message

Binary Encoding and Description

Table 5.4: Binary encoding for **transmitter_status** message sent on FPort 16

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **transmitter_status** (encoded value: 1).
version	`uint4`	4..7	Message version: 0.
temperature	`int8`	8..15	Measured transmitter temperature in °C [°F].
rssi	`int8`	16..23	RSSI (Received Signal Strength Indicator) of the latest downlink.
lora_tx_counter	`uint16`	24..39	Number of LoRa transmissions since the last **transmitter_status** message.
bist.power_supply	`bool`	40	Passed if the power supply module is operational.
bist.configuration	`bool`	41	Passed if the device has all mandatory configurations.
bist.sensor_connection	`bool`	42	Passed if the sensor is connected to the transmitter.
bist.sensor_paired	`bool`	43	Passed if the sensor is paired to the transmitter. See `require_sensor_pairing` in Section 5.7.1.
bist.flash_memory	`bool`	44	Passed if the flash memory is operational.
bist.internal_temperature_sensor	`bool`	45	Passed if the temperature of the microcontroller unit is within the acceptable range.

Continues on next page

Table 5.4 Continued: Binary encoding for **transmitter_status** message sent on FPort 16

Field ID	Type	Bit Index	Description
bist.time_synchronized	`bool`	46	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.
Reserved for Future Use	`uint1`	47	

5.4.3 Example

To send a **transmitter_status** uplink every day. This is the default configuration.

```
{
    "configuration_update_request": {
        "version": 0,
        "tag": "0xfe2192c9",
        "payload": {
            "type": "schedule",
            "version": 0,
            "command": "replace",
            "timing": 1440,
            "triggered_on_button_press": true,
            "send": true,
            "settings": {
                "type": "transmitter_status",
                "version": 0
            }
        }
    }
}
```

Listing 12: Example of the JSON structure for scheduling a **transmitter_status** uplink.

5.5 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.

5.5.1 transmitter_battery schedule

transmitter_battery uplinks are configured using the *Schedule Configuration* (as described in Section 4.3), with the *Schedule Configuration Settings* described below. Note that the index for the first bit is 160 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 1.

Schedule Configuration Settings

```
{
    "type": "transmitter_battery",
    "version": 0
}
```

Listing 13: JSON structure for **transmitter_battery** schedule configuration settings

Binary Encoding and Description

Table 5.5: Binary encoding for **transmitter_battery** schedule configuration settings

Field ID	Type	Bit Index	Description
type	`uint12`	160..171	Schedule settings type for **transmitter_battery** (encoded value: 15).
version	`uint4`	172..175	Schedule settings version: 0.

5.5.2 transmitter_battery message

Message Structure

```
{
    "transmitter_battery": {
        "version": 0,
        "transmitter_charge_used": <number>,
        "sensor_charge_used": <number>,
        "average_temperature": <number>,
        "battery_level": <number>
    }
}
```

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

Binary Encoding and Description

Table 5.6: Binary encoding for **transmitter_battery** message sent on FPort 14

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **transmitter_battery** (encoded value: 1).
version	`uint4`	4..7	Message version: 0.
transmitter_charge_used	`pfloat15`	8..22	Amount of charge used by the transmitter in mAh.
sensor_charge_used	`pfloat15`	23..37	Amount of charge used by the sensor in mAh.
average_temperature	`float16`	38..53	Average temperature of the device since the previous message, measured in ℃ [℉].
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
Reserved for Future Use	`uint2`	62..63	

5.5.3 transmitter_battery_reset_request message

Message Structure

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

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

Binary Encoding and Description

Table 5.7: Binary encoding for **transmitter_battery_reset_request** message sent on FPort 14

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

5.5.4 transmitter_battery_reset_answer message

Message Structure

```
{
    "transmitter_battery_reset_answer": {
        "version": 0
    }
}
```

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

Binary Encoding and Description

Table 5.8: Binary encoding for **transmitter_battery_reset_answer** message sent on FPort 14

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID for **transmitter_battery_reset_answer** (encoded value: 3).
version	uint4	4..7	Message version: 0.

5.6 Factory Reset

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

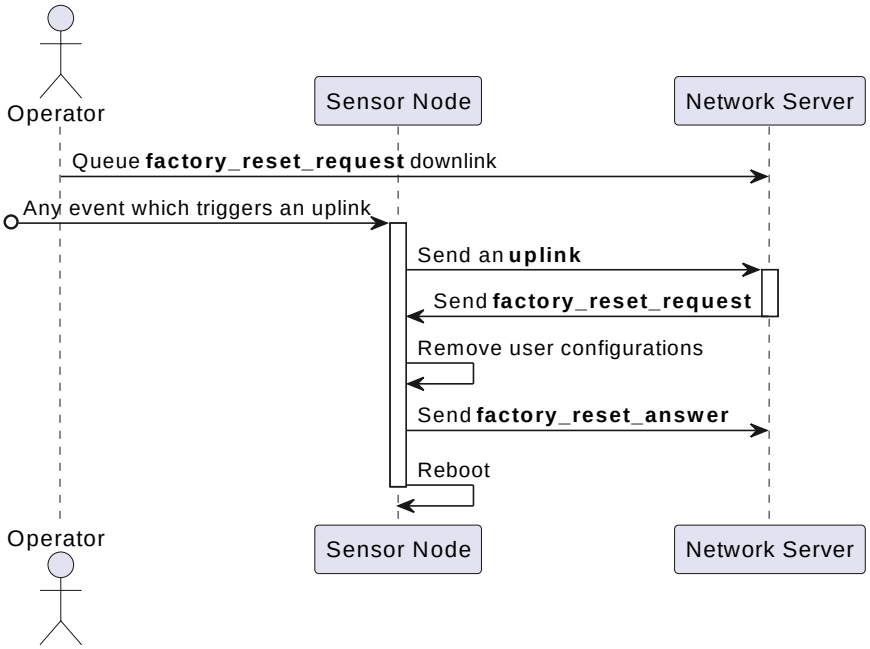


Figure 5.3: Factory Reset

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 5.3).

5.6.1 factory_reset_request message

Message Structure

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

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

Binary Encoding and Description

Table 5.9: Binary encoding for **factory_reset_request** message sent on FPort 14

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

5.6.2 factory_reset_answer message

Message Structure

```
{
    "factory_reset_answer": {
        "version": 0
    }
}
```

Listing 18: JSON structure for **factory_reset_answer** message

Binary Encoding and Description

Table 5.10: Binary encoding for **factory_reset_answer** message sent on FPort 14

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID for **factory_reset_answer** (encoded value: 2).
version	uint4	4..7	Message version: 0.

5.7 Configuration

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

5.7.1 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 19: JSON structure for **transmitter** configuration payload

Binary Encoding and Description

Table 5.11: 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	Redundancy percentage of fragmented uplinks.
Reserved for Future Use	`uint2`	70..71	

Default

```
{
    "configuration_update_request": {
        "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 20: Example of the JSON structure for the default transmitter configuration.

6 Sensor

The NEON Vibration Sensor allows for condition-based monitoring of an asset.

Scheduling of Sensor Tasks

When scheduling tasks for the Vibration Sensor, the following settings are common:

- `sample_speed_divider`: Used to control the sampling rate of the sensor. It divides the maximum rate of 26,667 samples per second by an integer. The vibration sensor always acquires the same amount of vibration data points, so a higher `sample_speed_divider` results in a longer measurement. A measurement takes approximately (sample_speed_divider $\times$ 0.61) seconds. It is advisable to capture at least 10 periods of the lowest frequency of interest in the measurement. Calculate the minimal `sample_speed_divider` using the equation: sample_speed_divider = $\frac{16.3}{f}$ where f is the frequency of interest and sample_speed_divider is rounded up to the nearest integer.
  Example: The rotating speed of the asset is 3 Hz (180 RPM). The user is not interested in any frequencies below the rotating speed. The frequency of interest f = 3, leading to $\frac{16.3}{3}$ = 5.4. The `sample_speed_divider` should be set to 6 or higher.
- `f_min`: Sets the lower end of the sensor's bandwidth. The minimum value is restricted by the greater of $\frac{5}{\text{sample\_speed\_divider}}$ or 0.5 Hz. If a lower value is set, it will be restricted by the sensor.
- `f_max`: Sets the upper limit of the frequency range and cannot exceed 0.5 $\times$ sample_speed. If set to a higher value, the sensor will reject the schedule.
- `axis`: The axis the vibration will be measured for. For more details on the axes of the NEON Vibration Sensor, please see the User Manual.
- `range`: Acceleration's scale range. A lower range provides a higher magnitude resolution. It is recommended to use the scale just above the highest expected magnitude to maximize resolution.

These settings define the parameters used for the vibration measurement and processing. When these settings are equal for overlapping schedules, the vibration measurement is only done once. If these settings differ, the vibration measurement is done as many times as necessary to perform the overlapping tasks.

6.1 Measurement

The NEON Vibration Sensor reports the overall measurement values with the **measurement** uplink.

6.1.1 vb_measurement schedule

measurement uplinks are configured using the *Schedule Configuration* (as described in Section 4.3), with the *Schedule Configuration Settings* described below. Note that the index for the first bit is 160 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 10.

Schedule Configuration Settings

```
{
    "type": "vb_measurement",
    "version": 0,
    "axis": <string>,
    "range": <string>,
    "sample_speed_divider": <number>,
    "f_min": <number>,
    "f_max": <number>,
    "enable_confirmed_message": <boolean>,
    "send_condition": {
        "value_type": <string>,
        "threshold": <number>
    }
}
```

Listing 21: JSON structure for **vb_measurement** schedule configuration settings

Binary Encoding and Description

Table 6.1: Binary encoding for **vb_measurement** schedule configuration settings

Field ID	Type	Bit Index	Description
type	`uint12`	160..171	Schedule settings type for **vb_measurement** (encoded value: 1).
version	`uint4`	172..175	Schedule settings version: 0.
axis	`uint2`	176..177	Axis for which the vibration is measured. If `"all"` is selected, the schedule will result in one message for each of the three axes. Value mapping: `"x"` → 0 `"y"` → 1 `"z"` → 2 `"all"` → 3
range	`uint2`	178..179	Acceleration's scale range. A lower range provides a higher magnitude resolution. It is recommended to use the scale just above the highest expected magnitude to maximize resolution. Value mapping: `"gscale_2"` → 0 `"gscale_4"` → 1 `"gscale_8"` → 2 `"gscale_16"` → 3
sample_speed_divider	`uint8`	180..187	Adjusts the sensor's sampling rate by dividing the maximum speed of 26,667 samples per second by a factor between 1 and 255. Calculate the advised minimum value using sample_speed_divider = $\frac{16.3}{f}$ where f is the lowest frequency of interest and the result is rounded up to the nearest integer.
f_min	`pfloat15`	188..202	Lower bound of the frequency range to be analyzed and/or sent in Hz.
f_max	`pfloat15`	203..217	Upper bound of the frequency range to be analyzed and/or sent in Hz. Must be lower than the sampling rate divided by 2.
enable_confirmed_message	`bool`	218	Enable confirmed messages.
send_condition.value_type	`uint4`	219..222	Type of the *send condition* for the schedule. The device only transmits the scheduled uplink if it is satisfied. If set to `"always"`, the scheduled messages are sent unconditionally. Value mapping: `"always"` → 0 `"peak_acceleration_above"` → 1 `"rms_acceleration_above"` → 2 `"rms_velocity_above"` → 3 `"temperature_above"` → 4 `"temperature_below"` → 5
send_condition.threshold	`float16`	223..238	Value that needs to be reached for the send condition. To be used in combination with `send_condition.value_type`.
Reserved for Future Use	`uint1`	239	

6.1.2 measurement message

The **measurement** message contains the overall values for a vibration signal measured according to the scheduled `"vb_measurement"` task.

Message Structure

```
{
    "measurement": {
        "version": 0,
        "timestamp": <string>,
        "axis": <string>,
        "temperature": <number>,
        "peak_acceleration": <number>,
        "rms_acceleration": <number>,
        "rms_velocity": <number>
    }
}
```

Listing 22: JSON structure for **measurement** message

Binary Encoding and Description

Table 6.2: Binary encoding for **measurement** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **measurement** (encoded value: 1).
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.
axis	`uint2`	24..25	Axis for which the vibration is measured. Value mapping: `"x"` → 0 `"y"` → 1 `"z"` → 2
temperature	`int8`	26..33	Measured sensor temperature in °C [°F].
peak_acceleration	`float16`	34..49	Measured peak acceleration in g.
rms_acceleration	`float16`	50..65	Measured RMS acceleration in g.
rms_velocity	`float16`	66..81	Measured RMS velocity in mms^{-1} [ins^{-1}].
Reserved for Future Use	`uint6`	82..87	

6.1.3 Example

To send a **measurement** uplink for the x, y, and Z axis every hour.

```
{
    "configuration_update_request": {
        "version": 0,
        "tag": "0x3633c816",
        "payload": {
            "type": "schedule",
            "version": 0,
            "command": "set",
            "timing": 60,
            "triggered_on_button_press": false,
            "send": true,
            "settings": {
                "type": "vb_measurement",
                "version": 0,
                "axis": "all",
                "range": "gscale_16",
                "sample_speed_divider": 1,
                "f_min": 5.0,
                "f_max": 6300.0,
                "enable_confirmed_message": false,
                "send_condition": {
                    "value_type": "always",
                    "threshold": 0.0
                }
            }
        }
    }
}
```

Listing 23: Example of the JSON structure for scheduling a **measurement** uplink.

6.2 Past Measurement Request

The sensor stores a **measurement** uplink associated with each vibration measurement taken. These uplinks can be retrieved using the **past_measurement_request**.

When the request is received, the device will send all **measurement** uplinks associated with a vibration measurement closest to the `timestamp` provided.

The following restrictions apply:

- If the storage is full, the oldest measurement is discarded.
- It is not possible to request a measurement older than 21 days (due to `short_timestamp` type restrictions, see Section 1.3.1).

6.2.1 past_measurement_request message

Message Structure

```
{
    "past_measurement_request": {
        "version": 0,
        "timestamp": <string>
    }
}
```

Listing 24: JSON structure for **past_measurement_request** message

Binary Encoding and Description

Table 6.3: Binary encoding for **past_measurement_request** message sent on FPort 14

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **past_measurement_request** (encoded value: 0).
version	`uint4`	4..7	Message version: 0.
timestamp	`timestamp`	8..39	UTC ISO timestamp of the measurement.

6.3 Statistics

The NEON Vibration Sensor can keep track of the average and extremes of all overall sensor values described in Section 6.1. In order to keep track of the statistics, a **statistics** schedule must be active. There can be only one active statistics schedule per sensor value per axis. When a **statistics** schedule is active, the device will keep track of all sensor values that result from vibration measurements with the same common settings as those of the **statistics** schedule. This includes any measurement except for the **low sample speed envelope** measurement. The statistics can be taken over a time anywhere between 1 minute and 3 weeks. The statistics are reported with the **statistics** uplinks. Whenever a **statistics** uplink is sent, the statistics are reset.

6.3.1 vb_statistics schedules

statistics uplinks are configured using the *Schedule Configuration* (as described in Section 4.3), with the *Schedule Configuration Settings* described below. Note that the index for the first bit is 160 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 1 per statistics type.

Schedule Configuration Settings

```
{
    "type": "vb_statistics...",
    "version": 0,
    "range": <string>,
    "sample_speed_divider": <number>,
    "f_min": <number>,
    "f_max": <number>
}
```

Listing 25: JSON structure for **vb_statistics...** schedule configuration settings

Binary Encoding and Description

Table 6.4: Binary encoding for **vb_statistics...** schedule configuration settings

Field ID	Type	Bit Index	Description
type	uint12	160..171	Schedule settings type for the desired statisitcs. Value mapping: "vb_statistics_x_rms_velocity" → 5 "vb_statistics_x_rms_acceleration" → 6 "vb_statistics_x_peak_acceleration" → 7 "vb_statistics_y_rms_velocity" → 8 "vb_statistics_y_rms_acceleration" → 9 "vb_statistics_y_peak_acceleration" → 10 "vb_statistics_z_rms_velocity" → 11 "vb_statistics_z_rms_acceleration" → 12 "vb_statistics_z_peak_acceleration" → 13 "vb_statistics_temperature" → 14
version	uint4	172..175	Schedule settings version: 0.

Continues on next page

Table 6.4 Continued: Binary encoding for **vb_statistics...** schedule configuration settings

Field ID	Type	Bit Index	Description
range	`uint2`	176..177	Acceleration's scale range. A lower range provides a higher magnitude resolution. It is recommended to use the scale just above the highest expected magnitude to maximize resolution. Value mapping: `"gscale_2"` → 0 `"gscale_4"` → 1 `"gscale_8"` → 2 `"gscale_16"` → 3
sample_speed_divider	`uint8`	178..185	Adjusts the sensor's sampling rate by dividing the maximum speed of 26,667 samples per second by a factor between 1 and 255. Calculate the advised minimum value using sample_speed_divider = $\frac{16.3}{f}$ where f is the lowest frequency of interest and the result is rounded up to the nearest integer.
f_min	`pfloat15`	186..200	Lower bound of the frequency range to be analyzed and/or sent in Hz.
f_max	`pfloat15`	201..215	Upper bound of the frequency range to be analyzed and/or sent in Hz. Must be lower than the sampling rate divided by 2.

6.3.2 statistics message

Message Structure

```
{
    "statistics": {
        "version": 0,
        "selection": <string>,
        "min": <number>,
        "max": <number>,
        "avg": <number>,
        "max_timestamp": <string>
    }
}
```

Listing 26: JSON structure for **statistics** message

Binary Encoding and Description

Table 6.5: Binary encoding for **statistics** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **statistics** (encoded value: 4).
version	`uint4`	4..7	Message version: 0.
selection	`uint4`	8..11	Type of the statistics. Value mapping: `"x_rms_velocity"` → 0 `"x_rms_acceleration"` → 1 `"x_peak_acceleration"` → 2 `"y_rms_velocity"` → 3 `"y_rms_acceleration"` → 4 `"y_peak_acceleration"` → 5 `"z_rms_velocity"` → 6 `"z_rms_acceleration"` → 7 `"z_peak_acceleration"` → 8 `"temperature"` → 9
min	`float16`	12..27	Minimum value measured during the scheduled period in g, mms^{-1} [ins^{-1}] or °C [°F], depending on selection.
max	`float16`	28..43	Maximum value measured during the scheduled period in g, mms^{-1} [ins^{-1}] or °C [°F], depending on selection.
avg	`float16`	44..59	Average value measured during the scheduled period in g, mms^{-1} [ins^{-1}] or °C [°F], depending on selection.
max_timestamp	`short_timestamp`	60..75	UTC ISO timestamp of the maximum measurement. Detailed information about the binary encoding is provided in Section 1.3.1.
Reserved for Future Use	`uint4`	76..79	

6.3.3 Example

To send a **statistics** uplink for the RMS on the z axis every day.

```
{
    "configuration_update_request": {
        "version": 0,
        "tag": "0xc2eedfc8",
        "payload": {
            "version": 0,
            "type": "schedule",
            "command": "replace",
            "triggered_on_button_press": false,
            "timing": 1440,
            "send": true,
            "settings": {
                "version": 0,
                "type": "vb_statistics_z_rms_velocity",
                "range": "gscale_16",
                "sample_speed_divider": 1,
                "f_min": 5.0,
                "f_max": 6300.0
            }
        }
    }
}
```

Listing 27: Example of the JSON structure for scheduling a **statistics** uplink for Z RMS Velocity.

6.4 Spectrum

The NEON Vibration Sensor can be configured to send three distinct types of spectrum: acceleration, velocity, and (acceleration) envelope. The spectrums are reported using the **spectrum** uplink.

Important remarks regarding spectrum measurements:

- The **spectrum** is sent using the Fragmented Uplink transport layer (See Section 2).
- The sensor can only store two spectrums.
- At sample speeds below the maximum (`"sample_speed_divider" > 1`), the envelope spectrum requires its own measurement. This measurement is not used for other purposes.

6.4.1 Resolution and Frequency Range

The `sample_speed_divider` determines the frequency resolution and the frequency range that can be accurately displayed in the spectrum. The frequency resolution of the output spectrum is approximately given by $\frac{1.63}{\text{sample\_speed\_divider}}$. The -3dB points, which define the bandwidth of the sensor, are set using `f_min` and `f_max`. `f_min` determines the lowest frequency that can be accurately measured. The minimum value of `f_min` is restricted by the greater of $\frac{5}{\text{sample\_speed\_divider}}$ or 0.5 Hz. If the `f_min` is set to a lower value, the sensor will restrict it to the previously mentioned value. `f_max` sets the upper limit of the frequency range and cannot exceed 0.5 $\times$ sample_speed. If the `f_max` is set to a higher value, the sensor will reject the schedule. Only the part of the spectrum between `f_min` and `f_max` will be sent in the **spectrum** uplink. The sensor has multiple measurement modes for spectrums. The modes are controlled by the settings mentioned above. There is only one mode for the acceleration spectrum, regardless of the settings.

There are two modes for the velocity spectrum:

- Normal mode (f_min $\geq \frac{10}{\text{sample\_speed\_divider}}$): The `f_max` is subject to the above limitations.
- Low frequency mode (f_min $< \frac{10}{\text{sample\_speed\_divider}}$): The `f_max` will be limited to $\frac{1000}{\text{sample\_speed\_divider}}$ by the sensor.

There are two modes for the envelope spectrum:

- Normal mode (sample_speed_divider == 1): The `f_min` and `f_max` are subject to normal limits as described above.
- High resolution mode (sample_speed_divider $>$ 1): This mode is mainly meant for use with low RPM machines. The `f_min` and `f_max` will not affect the frequency limits of the measurement itself, but only the part of the spectrum that is sent in the **spectrum** uplink. This means the envelope signal will be measured over the highest possible bandwidth of the sensor. The `f_max` and frequency resolution are still subject to the limits mentioned above. This is a special type of vibration measurement, making it unable to be combined with other overlapping measurements.

6.4.2 vb_spectrum schedule

spectrum uplinks are configured using the *Schedule Configuration* (as described in Section 4.3), with the *Schedule Configuration Settings* described below. Note that the index for the first bit is 160 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 10.

Schedule Configuration Settings

```
{
    "type": "vb_spectrum",
    "version": 0,
    "axis": <string>,
    "range": <string>,
    "sample_speed_divider": <number>,
    "f_min": <number>,
    "f_max": <number>,
    "spectrum_type": <string>,
    "averaging": <number>,
    "time_to_transmit_min": <number>,
    "send_condition": {
        "value_type": <string>,
        "threshold": <number>
    }
}
```

Listing 28: JSON structure for **vb_spectrum** schedule configuration settings

Binary Encoding and Description

Table 6.6: Binary encoding for **vb_spectrum** schedule configuration settings

Field ID	Type	Bit Index	Description
type	uint12	160..171	Schedule settings type for **vb_spectrum** (encoded value: 3).
version	uint4	172..175	Schedule settings version: 0.
axis	uint2	176..177	Axis for which the vibration is measured. Value mapping: "x" → 0 "y" → 1 "z" → 2
range	uint2	178..179	Acceleration's scale range. A lower range provides a higher magnitude resolution. It is recommended to use the scale just above the highest expected magnitude to maximize resolution. Value mapping: "gscale_2" → 0 "gscale_4" → 1 "gscale_8" → 2 "gscale_16" → 3
sample_speed_divider	uint8	180..187	Adjusts the sensor's sampling rate by dividing the maximum speed of 26,667 samples per second by a factor between 1 and 255. Calculate the advised minimum value using sample_speed_divider = $\frac{16.3}{f}$ where f is the lowest frequency of interest and the result is rounded up to the nearest integer.

Continues on next page

Table 6.6 Continued: Binary encoding for **vb_spectrum** schedule configuration settings

Field ID	Type	Bit Index	Description
f_min	`pfloat15`	188..202	Lower bound of the frequency range to be analyzed and/or sent in Hz.
f_max	`pfloat15`	203..217	Upper bound of the frequency range to be analyzed and/or sent in Hz. Must be lower than the sampling rate divided by 2.
spectrum_type	`uint2`	218..219	Type of frequency spectrum. Value mapping: `"acceleration"` → 0 `"velocity"` → 1 `"envelope"` → 2
averaging	`uint3`	220..222	Amount of spectrums to average over. Used to reduce noise in the spectrums.
time_to_transmit_min	`pfloat15`	223..237	Specifies the duration of time used to transmit the fragmented uplink in min.
send_condition.value_type	`uint4`	238..241	Type of the *send condition* for the schedule. The device only transmits the scheduled uplink if it is satisfied. If set to `"always"`, the scheduled messages are sent unconditionally. Value mapping: `"always"` → 0 `"peak_acceleration_above"` → 1 `"rms_acceleration_above"` → 2 `"rms_velocity_above"` → 3 `"temperature_above"` → 4 `"temperature_below"` → 5
send_condition.threshold	`float16`	242..257	Value that needs to be reached for the send condition. To be used in combination with `send_condition.value_type`.
Reserved for Future Use	`uint4`	258..261	

6.4.3 spectrum message

The **spectrum** message contains the spectrum data calculated by the device. The vibration data used depends on the scheduled `"vb_spectrum"` settings.

If multiple `"vb_spectrum"` schedules overlap, the resulting **spectrum** messages are sent one after the other. Up to two pending **spectrum** messages are stored on the device. In case of a third, it will overwrite one with the same or lower priority. The priority levels are, listed from highest to lowest priority:

1. User request (button press or `"execute"` schedule command)
2. Spectrum Alert trigger
3. Scheduled trigger

Message Structure

```
{
    "spectrum": {
        "version": 0,
        "timestamp": <string>,
        "axis": <string>,
        "spectrum_type": <string>,
        "temperature": <number>,
        "f_min": <number>,
        "peak_acceleration": <number>,
        "rms_acceleration": <number>,
        "rms_velocity": <number>,
        "rpm": <number>,
        "frequencies": <array of numbers>,
        "magnitudes": <array of numbers>
    }
}
```

Listing 29: JSON structure for **spectrum** message

Binary Encoding and Description

Table 6.7: Binary encoding for **spectrum** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **spectrum** (encoded value: 5).
version	`uint4`	4..7	Message version: 0.
timestamp	`timestamp`	8..39	UTC ISO timestamp of the measurement.
axis	`uint2`	40..41	Axis for which the vibration is measured. Value mapping: `"x"` → 0 `"y"` → 1 `"z"` → 2
spectrum_type	`uint2`	42..43	Type of frequency spectrum. Value mapping: `"acceleration"` → 0 `"velocity"` → 1 `"envelope"` → 2
temperature	`int8`	44..51	Measured sensor temperature in °C [°F].
df	`float16`	52..67	Frequency bin size in Hz.

Continues on next page

Table 6.7 Continued: Binary encoding for **spectrum** message sent on FPort 17

Field ID	Type	Bit Index	Description
f_min	`float16`	68..83	Lower bound of the frequency range sent in Hz.
peak_acceleration	`float16`	84..99	Measured peak acceleration in g.
rms_acceleration	`float16`	100..115	Measured RMS acceleration in g.
rms_velocity	`float16`	116..131	Measured RMS velocity in mms^{-1} [ins^{-1}].
rpm	`float32`	132..163	Estimated rotating speed in rpm. Bound by `rpm_min` and `rpm_max` set in the **vb_asset** configuration (see Section 6.8.1).
magnitudes_scaling	`pfloat15`	164..178	Scaling value used to convert `magnitude_values` to Hz, based on the maximum magnitude measured.
Reserved for Future Use	`uint5`	179..183	
magnitude_values	`uint10[]`	184..	Array with the scaled values of the spectrum calculated.
frequencies			Array of spectrum frequencies: $\texttt{frequencies[i]} = \texttt{f\_min} + \texttt{df} \times \texttt{i}$ where `i` is the index of `magnitude_values` (starting from 0).
magnitudes			Array of spectrum magnitudes: $\texttt{magnitudes[i]} = \texttt{magnitude\_values[i]} \times \texttt{magnitudes\_scaling}$ where `i` is the index of `magnitude_values` (starting from 0).

6.4.4 Example

To send a **spectrum** uplink for the z axis every week. The fragmented message is sent within one week.

```
{
    "configuration_update_request": {
        "version": 0,
        "tag": "0x003b3f10",
        "payload": {
            "type": "schedule",
            "version": 0,
            "command": "replace",
            "timing": 10080,
            "triggered_on_button_press": false,
            "send": true,
            "settings": {
                "type": "vb_spectrum",
                "version": 0,
                "axis": "z",
                "range": "gscale_16",
                "sample_speed_divider": 1,
                "f_min": 5.0,
                "f_max": 6300.0,
                "spectrum_type": "acceleration",
                "averaging": 0,
                "time_to_transmit_min": 10080,
                "send_condition": {
                    "value_type": "always",
                    "threshold": 0.0
                }
            }
        }
    }
}
```

Listing 30: Example of the JSON structure for scheduling a **spectrum** uplink.

6.5 Machine Fault Indicator

The NEON Vibration Sensor can be configured to diagnose two primary types of faults: common faults and bearing faults.

Common Faults

- When set to detect common faults, the sensor focuses on diagnosing anomalies in the velocity spectrum that are synchronous with the rotating frequency of the asset.
- This configuration is effective in identifying issues such as unbalance and looseness, which can lead to inefficiency and wear if left unaddressed.

Bearing Faults

- For bearing faults, the sensor is configured to analyze anomalies in the envelope spectrum.
- This allows the sensor to detect early signs of bearing wear or damage.
- Important remark: At any time, the sensor can only handle one measurement in high resolution mode (sample_speed_divider $>$ 1, fault_type = "bearing_fault", see Section 6.4.1). Selecting axis = "all" will result in three separate measurements, and will therefore be rejected. For sample_speed_divider = 1, normal conditions apply.

Fault Categorization and Proof

- Upon detecting an unhealthy condition, the sensor categorizes the fault, providing an indication of the type of issue identified.
- For example, if a bearing fault is detected, the sensor will specifically indicate this category, enabling targeted maintenance actions.
- The sensor will always provide proof for the diagnosed fault in the form of harmonics. It will send the frequency and amplitude of the first 10 harmonics that indicate there is a fault.
- When the sensor diagnoses the asset to be healthy, it will send the most significant set of harmonics in the spectrum.
- The sensor might send a set of 4 instead of 10 harmonics, depending on a combination of the region and data rate.

6.5.1 vb_machine_fault_indicator schedule

machine_fault_indicator uplinks are configured using the *Schedule Configuration* (as described in Section 4.3), with the *Schedule Configuration Settings* described below. Note that the index for the first bit is 160 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 10.

Schedule Configuration Settings

```
{
    "type": "vb_machine_fault_indicator",
    "version": 0,
    "axis": <string>,
    "range": <string>,
    "sample_speed_divider": <number>,
    "f_min": <number>,
    "f_max": <number>,
    "fault_type": <string>,
    "send_condition": {
        "value_type": <string>,
        "threshold": <number>
    }
}
```

Listing 31: JSON structure for **vb_machine_fault_indicator** schedule configuration settings

Binary Encoding and Description

Table 6.8: Binary encoding for **vb_machine_fault_indicator** schedule configuration settings

Field ID	Type	Bit Index	Description
type	`uint12`	160..171	Schedule settings type for **vb_machine_fault_indicator** (encoded value: 2).
version	`uint4`	172..175	Schedule settings version: 0.
axis	`uint2`	176..177	Axis for which the vibration is measured. If `"all"` is selected, the schedule will result in one message for each of the three axes. Value mapping: `"x"` → 0 `"y"` → 1 `"z"` → 2 `"all"` → 3
range	`uint2`	178..179	Acceleration's scale range. A lower range provides a higher magnitude resolution. It is recommended to use the scale just above the highest expected magnitude to maximize resolution. Value mapping: `"gscale_2"` → 0 `"gscale_4"` → 1 `"gscale_8"` → 2 `"gscale_16"` → 3
sample_speed_divider	`uint8`	180..187	Adjusts the sensor's sampling rate by dividing the maximum speed of 26,667 samples per second by a factor between 1 and 255. Calculate the advised minimum value using sample_speed_divider = $\frac{16.3}{f}$ where f is the lowest frequency of interest and the result is rounded up to the nearest integer.
f_min	`pfloat15`	188..202	Lower bound of the frequency range to be analyzed and/or sent in Hz.
f_max	`pfloat15`	203..217	Upper bound of the frequency range to be analyzed and/or sent in Hz. Must be lower than the sampling rate divided by 2.
fault_type	`uint2`	218..219	Machine fault type. Value mapping: `"common_fault"` → 0 `"bearing_fault"` → 1
send_condition.value_type	`uint4`	220..223	Type of the *send condition* for the schedule. The device only transmits the scheduled uplink if it is satisfied. If set to `"always"`, the scheduled messages are sent unconditionally. Value mapping: `"always"` → 0 `"peak_acceleration_above"` → 1 `"rms_acceleration_above"` → 2 `"rms_velocity_above"` → 3 `"temperature_above"` → 4 `"temperature_below"` → 5
send_condition.threshold	`float16`	224..239	Value that needs to be reached for the send condition. To be used in combination with `send_condition.value_type`.

6.5.2 machine_fault_indicator message

The sensor sends the **machine_fault_indicator** uplink on each execution of the **vb_machine_fault_indicator** schedule that satisfies the configured `send_condition`.

Message Structure

```
{
    "machine_fault_indicator": {
        "version": 0,
        "timestamp": <string>,
        "axis": <string>,
        "fault_type": <string>,
        "fault_category": <string>,
        "harmonic_frequencies": <array of numbers>,
        "harmonic_amplitudes": <array of numbers>
    }
}
```

Listing 32: JSON structure for **machine_fault_indicator** message

Binary Encoding and Description

Table 6.9: Binary encoding for **machine_fault_indicator** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **machine_fault_indicator** (encoded value: 3).
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.
axis	`uint2`	24..25	Axis for which the vibration is measured. Value mapping: `"x"` → 0 `"y"` → 1 `"z"` → 2
fault_type	`uint2`	26..27	Machine fault type. Value mapping: `"common_fault"` → 0 `"bearing_fault"` → 1
fault_category	`uint6`	28..33	Machine fault category. Value mapping: `"none"` → 0 `"one_x"` → 1 `"two_x"` → 2 `"n_x"` → 3 `"bearing"` → 4
frequency_first_harmonic	`pfloat15`	34..48	Ground frequency of the harmonics.
amplitude_first_harmonic	`pfloat15`	49..63	Amplitude in the spectrum at the ground frequency of the harmonics.

Continues on next page

Table 6.9 Continued: Binary encoding for **machine_fault_indicator** message sent on FPort 17

Field ID	Type	Bit Index	Description
relative_nth_harmonic_amplitudes	`uint8[]`	64..	Sensor scales the harmonic amplitudes to be a multiple of the amplitude of the first harmonic between 0 and 2.5.
harmonic_frequencies			Array of harmonic frequencies: $\texttt{frequency\_first\_harmonic} \times \texttt{i}$ where `i` is 1, 2, .., number of harmonics.
harmonic_amplitudes			Array of harmonic amplitudes. The first harmonic is `frequency_first_harmonic` and the remaining are calculated: $\frac{\texttt{first}}{102} \times \texttt{relative\_nth\_harmonic\_amplitudes[i]}$ where `i` is the index (starting from 0) and `first` is the first harmonic frequency.

6.6 Alerts

The sensor supports monitoring of specific asset conditions with configurable alerts. When a measurement results in an alert status change (both `false` to `true` and `true` to `false`), an **alert** uplink is sent.

The NEON Vibration Sensor supports up to 8 sensor alerts and 5 spectrum alerts. The sensor alerts are based on the overall measurement values (e.g. Z peak acceleration), while the spectrum alerts are based on values within a specific frequency range.

6.6.1 vb_alert configuration

The sensor alerts are configured using the **vb_alert** configuration.

Configuration Payload

```
{
    "type": "vb_alert",
    "version": 0,
    "enable_confirmed_alert": <boolean>,
    "enable_spectrum_on_alert": <boolean>,
    "spectrum_type": <string>,
    "time_to_transmit_min": <number>,
    "hold_off_hours": <number>,
    "alert_0": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>
    },
    "alert_1": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>
    },
    "alert_2": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>
    },
    "alert_3": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>
    },
    "alert_4": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>
    },
    "alert_5": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>
    },
    "alert_6": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>
    },
    "alert_7": {
        "selection": <string>,
        "threshold": <number>,
```

```
        "hysteresis": <number>
    }
}
```

Listing 33: JSON structure for **vb_alert** configuration payload

Binary Encoding and Description

Table 6.10: Binary encoding for **vb_alert** configuration payload

Field ID	Type	Bit Index	Description
type	uint12	40..51	Configuration payload type for **vb_alert** (encoded value: 3).
version	uint4	52..55	Configuration payload version: 0.
enable_confirmed_alert	bool	56	If true, the **alert** uplinks are sent confirmed.
enable_spectrum_on_alert	bool	57	If true, a **spectrum** uplink is sent upon alert status change from false to true.
spectrum_type	uint2	58..59	Type of frequency spectrum. Value mapping: "acceleration" → 0 "velocity" → 1 "envelope" → 2
time_to_transmit_min	pfloat15	60..74	Specifies the duration of time used to transmit the fragmented uplink in min.
hold_off_hours	uint10	75..84	Time to wait before sending a new spectrum message triggered by an alert in h. Each alert has its own hold off timer.
alert_0.selection	uint4	85..88	Mode for sensor_alert_0. Similarly, alert_N.selection defines the mode for sensor_alert_N. Value mapping: "off" → 0 "x_rms_velocity_above" → 1 "x_rms_acceleration_above" → 2 "x_peak_acceleration_above" → 3 "y_rms_velocity_above" → 4 "y_rms_acceleration_above" → 5 "y_peak_acceleration_above" → 6 "z_rms_velocity_above" → 7 "z_rms_acceleration_above" → 8 "z_peak_acceleration_above" → 9 "temperature_above" → 10 "temperature_below" → 11
alert_0.threshold	float16	89..104	Threshold for sensor_alert_0. Similarly, alert_N.threshold defines the threshold for sensor_alert_N.
alert_0.hysteresis	pfloat15	105..119	Hysteresis of sensor_alert_0. It is the lagging added to prevent unwanted multiple alerts from rapid switching or noise. Similarly, alert_N.hysteresis defines the hysteresis of the sensor_alert_N.
alert_1.selection	uint4	120..123	
alert_1.threshold	float16	124..139	

Continues on next page

Table 6.10 Continued: Binary encoding for **vb_alert** configuration payload

Field ID	Type	Bit Index	Description
alert_1.hysteresis	`pfloat15`	140..154	
alert_2.selection	`uint4`	155..158	
alert_2.threshold	`float16`	159..174	
alert_2.hysteresis	`pfloat15`	175..189	
alert_3.selection	`uint4`	190..193	
alert_3.threshold	`float16`	194..209	
alert_3.hysteresis	`pfloat15`	210..224	
alert_4.selection	`uint4`	225..228	
alert_4.threshold	`float16`	229..244	
alert_4.hysteresis	`pfloat15`	245..259	
alert_5.selection	`uint4`	260..263	
alert_5.threshold	`float16`	264..279	
alert_5.hysteresis	`pfloat15`	280..294	
alert_6.selection	`uint4`	295..298	
alert_6.threshold	`float16`	299..314	
alert_6.hysteresis	`pfloat15`	315..329	
alert_7.selection	`uint4`	330..333	
alert_7.threshold	`float16`	334..349	
alert_7.hysteresis	`pfloat15`	350..364	
Reserved for Future Use	`uint6`	365..370	

6.6.2 vb_spectrum_alert configuration

The spectrum alerts are configured using the **vb_spectrum_alert** configuration. Limits for the resolution and frequency as discussed in Section 6.4.1 apply.

Configuration Payload

```
{
    "type": "vb_spectrum_alert",
    "version": 0,
    "enable_confirmed_alert": <boolean>,
    "enable_spectrum_on_alert": <boolean>,
    "time_to_transmit_min": <number>,
    "hold_off_hours": <number>,
    "alert_0": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>,
        "f_min": <number>,
        "f_max": <number>
    },
    "alert_1": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>,
        "f_min": <number>,
        "f_max": <number>
    },
    "alert_2": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>,
        "f_min": <number>,
        "f_max": <number>
    },
    "alert_3": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>,
        "f_min": <number>,
        "f_max": <number>
    },
    "alert_4": {
        "selection": <string>,
        "threshold": <number>,
        "hysteresis": <number>,
        "f_min": <number>,
        "f_max": <number>
    }
}
```

Listing 34: JSON structure for **vb_spectrum_alert** configuration payload

Binary Encoding and Description

Table 6.11: Binary encoding for **vb_spectrum_alert** configuration payload

Field ID	Type	Bit Index	Description
type	`uint12`	40..51	Configuration payload type for **vb_spectrum_alert** (encoded value: 4).
version	`uint4`	52..55	Configuration payload version: 0.
enable_confirmed_alert	`bool`	56	If `true`, the **alert** uplinks are sent confirmed.
enable_spectrum_on_alert	`bool`	57	If `true`, a **spectrum** uplink is sent upon alert status change from `false` to `true`.
time_to_transmit_min	`pfloat15`	58..72	Specifies the duration of time used to transmit the fragmented uplink in min.
hold_off_hours	`uint10`	73..82	Time to wait before sending a new spectrum message triggered by an alert in h. Each alert has its own hold off timer.
alert_0.selection	`uint5`	83..87	Mode for `spectrum_alert_0`. Similarly, `alert_N.selection` defines the mode for the `spectrum_alert_N`. Value mapping: `"off"` → 0 `"peak_velocity_x"` → 1 `"peak_velocity_y"` → 2 `"peak_velocity_z"` → 3 `"peak_acceleration_x"` → 4 `"peak_acceleration_y"` → 5 `"peak_acceleration_z"` → 6 `"peak_envelope_x"` → 7 `"peak_envelope_y"` → 8 `"peak_envelope_z"` → 9 `"rms_velocity_x"` → 10 `"rms_velocity_y"` → 11 `"rms_velocity_z"` → 12 `"rms_acceleration_x"` → 13 `"rms_acceleration_y"` → 14 `"rms_acceleration_z"` → 15 `"rms_envelope_x"` → 16 `"rms_envelope_y"` → 17 `"rms_envelope_z"` → 18 `"machine_fault_1x"` → 19 `"machine_fault_2x"` → 20 `"machine_fault_nx"` → 21 `"machine_fault_bearing"` → 22 `"machine_fault_any"` → 23
alert_0.threshold	`pfloat15`	88..102	Threshold for `spectrum_alert_0`. Similarly, `alert_N.threshold` defines the threshold for the `spectrum_alert_N`.
alert_0.hysteresis	`pfloat15`	103..117	Hysteresis of `spectrum_alert_0`. It is the lagging added to prevent unwanted multiple alerts from rapid switching or noise. Similarly, `alert_N.hysteresis` defines the hysteresis of the `spectrum_alert_N`.

Continues on next page

Table 6.11 Continued: Binary encoding for **vb_spectrum_alert** configuration payload

Field ID	Type	Bit Index	Description
alert_0.f_min	pfloat15	118..132	Starting frequency of the spectrum_alert_0 range. Similarly, alert_N.f_min defines the starting frequency of the spectrum_alert_N range.
alert_0.f_max	pfloat15	133..147	Ending frequency of the alert_0 range. Similarly, alert_N.f_max defines the ending frequency of the spectrum_alert_N range.
alert_1.selection	uint5	148..152	
alert_1.threshold	pfloat15	153..167	
alert_1.hysteresis	pfloat15	168..182	
alert_1.f_min	pfloat15	183..197	
alert_1.f_max	pfloat15	198..212	
alert_2.selection	uint5	213..217	
alert_2.threshold	pfloat15	218..232	
alert_2.hysteresis	pfloat15	233..247	
alert_2.f_min	pfloat15	248..262	
alert_2.f_max	pfloat15	263..277	
alert_3.selection	uint5	278..282	
alert_3.threshold	pfloat15	283..297	
alert_3.hysteresis	pfloat15	298..312	
alert_3.f_min	pfloat15	313..327	
alert_3.f_max	pfloat15	328..342	
alert_4.selection	uint5	343..347	
alert_4.threshold	pfloat15	348..362	
alert_4.hysteresis	pfloat15	363..377	
alert_4.f_min	pfloat15	378..392	
alert_4.f_max	pfloat15	393..407	

6.6.3 alert message

Message Structure

```
{
    "alert": {
        "version": 0,
        "timestamp": <string>,
        "sensor_alert_0": <boolean>,
        "sensor_alert_1": <boolean>,
        "sensor_alert_2": <boolean>,
        "sensor_alert_3": <boolean>,
        "sensor_alert_4": <boolean>,
        "sensor_alert_5": <boolean>,
        "sensor_alert_6": <boolean>,
        "sensor_alert_7": <boolean>,
        "spectrum_alert_0": <boolean>,
        "spectrum_alert_1": <boolean>,
        "spectrum_alert_2": <boolean>,
        "spectrum_alert_3": <boolean>,
        "spectrum_alert_4": <boolean>
    }
}
```

Listing 35: JSON structure for **alert** message

Binary Encoding and Description

Table 6.12: Binary encoding for **alert** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	`uint4`	0..3	Message ID for **alert** (encoded value: 2).
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.
sensor_alert_0	`bool`	24	If `true`, `alert_0` defined in **vb_alert** configuration is triggered. Similarly, `sensor_alert_N` indicates if `alert_N` is triggered.
sensor_alert_1	`bool`	25	
sensor_alert_2	`bool`	26	
sensor_alert_3	`bool`	27	
sensor_alert_4	`bool`	28	
sensor_alert_5	`bool`	29	
sensor_alert_6	`bool`	30	
sensor_alert_7	`bool`	31	
spectrum_alert_0	`bool`	32	If `true`, `alert_0` defined in **vb_spectrum_alert** configuration is triggered. Similarly, `spectrum_alert_N` indicates if `alert_N` is triggered.
spectrum_alert_1	`bool`	33	

Continues on next page

Table 6.12 Continued: Binary encoding for **alert** message sent on FPort 17

Field ID	Type	Bit Index	Description
spectrum_alert_2	`bool`	34	
spectrum_alert_3	`bool`	35	
spectrum_alert_4	`bool`	36	
Reserved for Future Use	`uint3`	37..39	

6.7 Boot

After every sensor reboot, the sensor reports the reboot reason with the **sensor_boot** uplink.

Reboots might occur intentionally in case of activation or firmware update. Otherwise, the sensor might reboot due to a system error as a matter of recovery.

6.7.1 sensor_boot message

Message Structure

```
{
    "sensor_boot": {
        "version": 0,
        "reboot_reason": <string>
    }
}
```

Listing 36: JSON structure for **sensor_boot** message

Binary Encoding and Description

Table 6.13: Binary encoding for **sensor_boot** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID for **sensor_boot** (encoded value: 0).
version	uint4	4..7	Message version: 0.
reboot_reason	uint16	8..23	Reason for the device's reboot. Value mapping: "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

6.8 Configuration

The sensor behavior can be adjusted using the **vb_asset** configuration.

6.8.1 vb_asset configuration

Configuration Payload

```
{
    "type": "vb_asset",
    "version": 0,
    "rpm_min": <number>,
    "rpm_max": <number>
}
```

Listing 37: JSON structure for **vb_asset** configuration payload

Binary Encoding and Description

Table 6.14: Binary encoding for **vb_asset** configuration payload

Field ID	Type	Bit Index	Description
type	uint12	40..51	Configuration payload type for **vb_asset** (encoded value: 5).
version	uint4	52..55	Configuration payload version: 0.
rpm_min	uint16	56..71	Lower bound of the RPM estimation in rpm.
rpm_max	uint16	72..87	Upper bound of the RPM estimation in rpm.

Default

```
{
    "configuration_update_request": {
        "version": 0,
        "tag": "0x6ddbffb9",
        "payload": {
            "version": 0,
            "type": "vb_asset",
            "rpm_min": 1000,
            "rpm_max": 4000
        }
    }
}
```

Listing 38: Example of the JSON structure for the default vb_asset configuration.

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 39: 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 40: 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 41: 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 42: 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 43: 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 44: 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 45: 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 46: 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 47: Code example for encoding timing in JavaScript

A.7 Custom CRC32 Calculation

```
var crc = (function () {
  var table = new Uint32Array(256);

  for (var i = 0; i < 256; i++) {
    var c = i;
    for (var j = 0; j < 8; j++) {
      if (c & 1) {
        c = 0xedb88320 ^ (c >>> 1);
      } else {
        c = c >>> 1;
      }
    }
    table[i] = c;
  }

  return function (str) {
    var crc = 0 ^ -1;
    for (var i = 0; i < str.length; i++) {
      crc = (crc >>> 8) ^ table[(crc ^ str.charCodeAt(i)) & 0xff];
    }
    return (crc ^ -1) >>> 0;
  };
})();
```

Listing 48: Code example for calculating a CRC32 in JavaScript