

NEON Vibration Sensor

Communication Protocol v4



Contents

1	Introduction	4
1.1	Binary Data Encoding	5
1.2	Message Structure	6
1.3	Special Types Encoding	7
1.3.1	Short Timestamp	7
1.3.2	Timing	8
1.3.3	Reserved Fields	9
1.4	Firmware and Hardware Versions	10
1.4.1	Example	10
1.5	Document Content	11
2	Fragmented Uplinks	12
2.1	Fragmented Uplink Session	13
2.2	Uplink Reconstruction	15
2.2.1	Integrity check of reconstructed message	15
2.3	fragmented_uplink_start message	16
2.4	fragmented_uplink_data message	17
2.5	fragmented_uplink_stop message	18
3	Configuration	19
3.1	Update	19
3.1.1	configuration_update_request message	20
3.1.2	configuration_update_answer message	21
4	Scheduling	22
4.1	Synchronized Schedules	22
4.2	Unsynchronized Schedules	22
4.3	Schedule Update	23
4.3.1	schedule_update_request message	24
4.3.2	schedule_update_answer message	26
5	Transmitter	28
5.1	Activation and Deactivation	28
5.1.1	transmitter_deactivated message	29
5.2	Button Press	30
5.3	Boot	31
5.3.1	transmitter_boot message	31
5.4	Status	33
5.4.1	transmitter_status task	33
5.4.2	transmitter_status message	34
5.4.3	Example	36
5.5	Battery	37
5.5.1	transmitter_battery task	37
5.5.2	transmitter_battery message	38
5.5.3	transmitter_battery_reset_request message	39
5.5.4	transmitter_battery_reset_answer message	39
5.6	Factory Reset	40
5.6.1	factory_reset_request message	41
5.6.2	factory_reset_answer message	41
5.7	Configuration	42
5.7.1	transmitter configuration	42
6	Sensor	44
6.1	Envelope Detection	44
6.2	Schedule Slots	45

6.2.1	Example	45
6.3	Triaxial Measurement	46
6.3.1	vb_triaxial_measurement task	46
6.3.2	triaxial-measurement message	49
6.3.3	Example	51
6.4	Measurement	52
6.4.1	vb_measurement task	52
6.4.2	measurement message	55
6.4.3	Example	56
6.5	Measurement Envelope	57
6.5.1	vb_measurement_envelope task	57
6.5.2	measurement_envelope message	60
6.5.3	Example	61
6.6	Past Measurement Request	62
6.6.1	past_measurement_request message	62
6.7	Statistics	63
6.7.1	vb_statistics task	63
6.7.2	statistics message	67
6.7.3	Example	69
6.8	Spectrum	70
6.8.1	Resolution and Frequency Range	70
6.8.2	vb_spectrum task	71
6.8.3	spectrum message	74
6.8.4	Example	76
6.9	Machine Fault Indicator	77
6.9.1	vb_machine_fault_indicator task	77
6.9.2	machine_fault_indicator message	80
6.10	Alerts	82
6.10.1	vb_alert configuration	82
6.10.2	vb_spectrum_alert configuration	86
6.10.3	alert message	90
6.11	Boot	92
6.11.1	sensor_boot message	92
6.12	Configuration	94
6.12.1	vb_asset configuration	94
A	Code Examples	95
A.1	Encoding and Decoding of Binary Data	95
A.2	Encoding and Decoding of float32	97
A.3	Encoding and Decoding of float16 and pfloat15	99
A.4	Decoding of short_timestamp	102
A.5	Encoding of cron expression	103
A.6	Encoding of timing	107
A.7	Custom CRC32 Calculation	108

Revision History

Revision	Date	Description
A1	2024-04-12	VB v4 protocol beta.
A2	2024-09-05	VB v4 protocol final.
		VB v4.1 protocol:
		• Added schedule_update_request and schedule_update_answer messages for handling of schedules. • Introduced measurement message v1. • Introduced vb_statistics task v0. • Introduced machine_fault_indicator v1. • Introduced spectrum v1. • Introduced vb_alert v1. • Introduced measurement_envelope v0. • Introduced triaxial_measurement v0.
B1	2026-03-09	• Removed the sample_speed_divider from all tasks. It is now calculated automatically based on other settings. • Improved high resolution envelope measurements. Now alerts and statistics can apply for these measurements. • Added missing restrictions for RPM min and max in the vb_asset configuration. • Added Firmware and Hardware Versions chapter. • Refactored the JSON structures to improve data association and clarity. • Corrected the unit of charge used in transmitter_battery. • Applied IMI branding.

Versioning scheme

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

1 Introduction

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

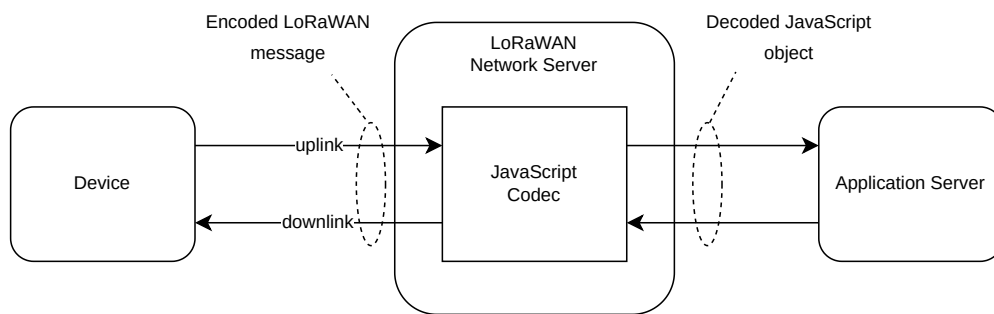


Figure 1.1: Encoding and decoding of LoRaWAN messages

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
- LoRaWAN® Firmware Management Protocol Specification TS006-1.0.0.

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

1.1 Binary Data Encoding

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

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Bytes Index	0								1								2							
Bit Array Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

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.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Bytes Hex	0xAB								0xCD								0xEF							
N Bits Read	4		8						3			7					2							
Bits Hex	0xA		0xBC						0x6			0x7B					0x2							

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`.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Bytes Hex	0x10								0x5B								0x04							
Field ID	ID				Version				a				b				c							
Field Type	uint4				uint4				int5				uint8				uint3							
Decoded	1				0				11				96				4							

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.

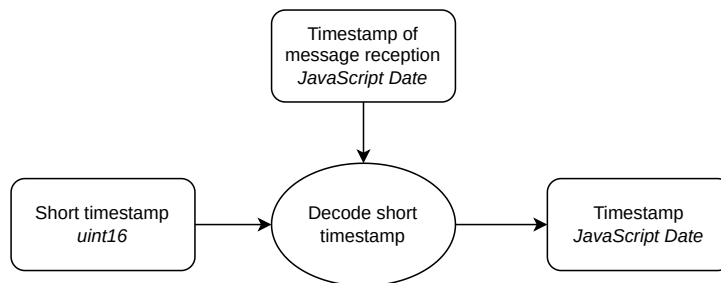


Figure 1.5: Using receive time to decode short timestamp

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 57.
 - $\text{recvTime} = 2023-08-10T11:31:00.000Z = 1691667060000$
 - $\text{recvMin} = \text{recvTime} / 60000 = 28194451$
 - $\text{wrapOffset} = \text{recvMin} \% 65535 = 14401$
 - $\text{wrapBase} = \text{recvMin} - \text{wrapOffset} = 28194451 - 14401 = 28180050$
 - $\text{gap} = \text{wrapOffset} - \text{shortTimestamp} = 14401 - 14400 = 1$
 - gap is positive and smaller than 21 days
 - $\text{msgMinutes} = \text{wrapBase} + \text{shortTimestamp} = 28180050 + 14400 = 28194450$
 - $\text{msgTime} = \text{msgMinutes} * 60000 = 1691667000000 = 2023-08-10T11:30:00.000Z$

1.3.2 Timing

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

Periodic	0	period	0
Cron	1	cron	

Figure 1.6: 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.

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

The cron expression consists of 5 fields separated by spaces:

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

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

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

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

Cron binary encoding

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

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

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

Examples

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

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

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

Timing Example 1 "0x00f000000000000000000000" Periodic trigger every 4 hours

¹<https://en.wikipedia.org/wiki/Cron>

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

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

1.3.3 Reserved Fields

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

1.4 Firmware and Hardware Versions

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

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

Table 1.2: Sensor Firmware Versions

Version	Protocol Revision
72fd	A
cfd2	A
3243	B

Table 1.3: Transmitter Firmware Versions

Version	Protocol Revision
bf7c	A
7590	A
f738	A
c583	B

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

Table 1.4: Sensor Hardware Versions

Version	Sensor
0000	DS-VB-xx-xx
0001	DS-VB-xx-xx

Table 1.5: Transmitter Hardware Versions

Version	Transmitter
0001	DS-LD-01-xx
0003	DS-LD-02-xx
0006	XT-01-xx

1.4.1 Example

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

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

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

```
{
  "message_name": "dev_version_ans",
  "FW_version": "0x3243c583",
  "HW_version": "0x00000003"
}
```

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

1.5 Document Content

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

Table 1.6: Overview of the device messages and configurations

Section	Message	Configuration
Fragmented Uplinks	fragmented_uplink_start fragmented_uplink_data fragmented_uplink_stop	
Configuration	configuration_update_request configuration_update_answer	
Scheduling	schedule_update_request schedule_update_answer	
Transmitter	transmitter_deactivated transmitter_boot transmitter_status transmitter_battery transmitter_battery_reset_request transmitter_battery_reset_answer factory_reset_request	transmitter
Sensor	triaxial_measurement measurement measurement_envelope 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.

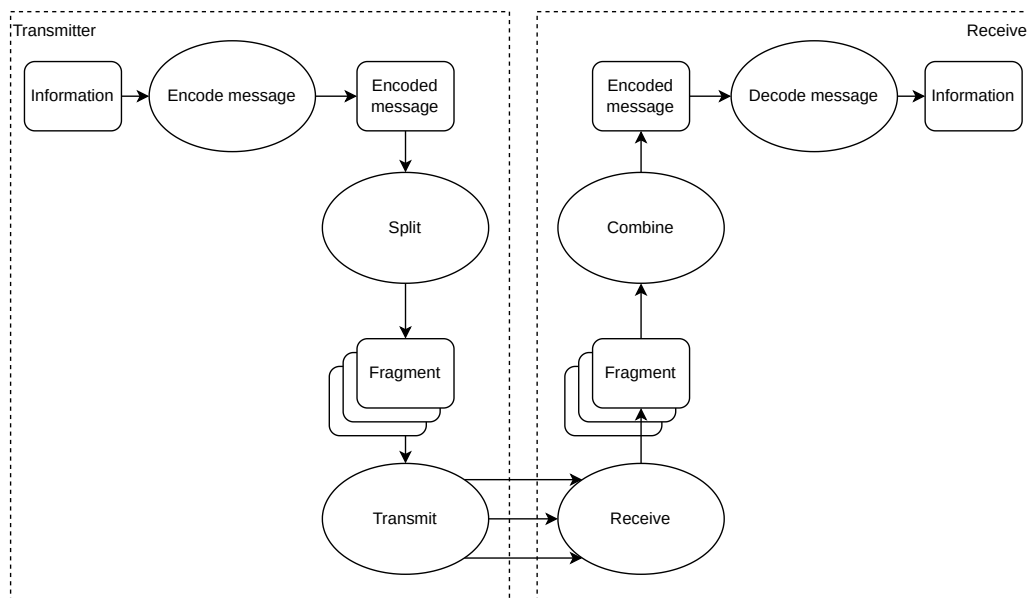


Figure 2.1: Illustration of the Fragmented Uplinks Layer Concept

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.

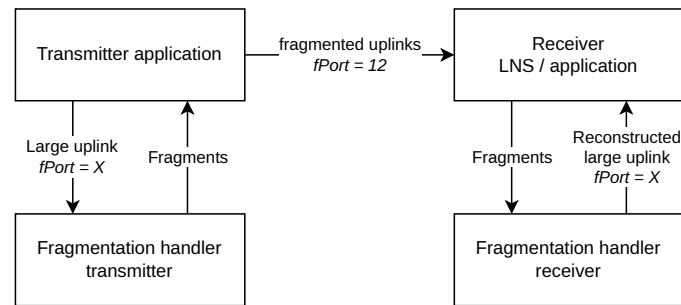


Figure 2.2: Fragmented uplink data flow

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.

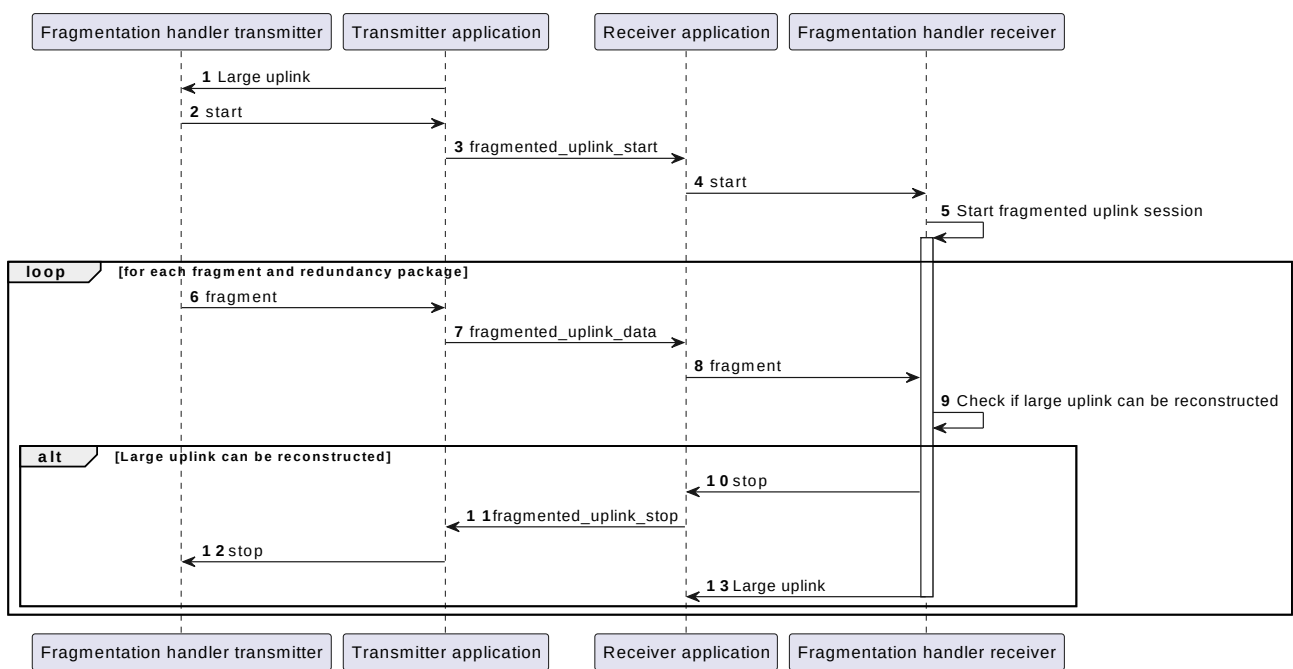


Figure 2.3: Fragmented uplink sequence

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 3 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(reconstructedMessage);
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 3: CRC calculation and verification of the reconstructed message

2.3 fragmented_uplink_start message

Message Structure

```
{
  "message_name": "fragmented_uplink_start",
  "message_version": 0,
  "fragmented_uplink_fport": <number>,
  "fragmented_uplink_size": <number>,
  "fragmented_uplink_fragment_size": <number>,
  "fragmented_uplink_crc": <number>
}
```

Listing 4: 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: 0.
message_version	uint4	4..7	Message version: 0.
fragmented_uplink_fport	uint8	8..15	FPort of the message being fragmented. It is used to label the original FPort to the reconstructed message.
fragmented_uplink_size	uint16	16..31	Size of the original uplink message that has been fragmented. The reconstructed message will have the same size.
fragmented_uplink_fragment_size	uint8	32..39	Payload size of each fragment in this fragmented uplink session.
fragmented_uplink_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

```
{
  "message_name": "fragmented_uplink_data",
  "message_version": 0,
  "fragmented_uplink_index": <number>,
  "fragmented_uplink_data": <array of numbers>
}
```

Listing 5: 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: 1.
message_version	uint4	4..7	Message version: 0.
fragmented_uplink_index	uint16	8..23	Index of the first fragment in the data array.
fragmented_uplink_data	uint8[]	24..	Payload containing one or more fragments. Each fragment is of size <code>fragment_size</code> as reported in fragmented_uplink_start . The size of this array is a multiple of <code>fragment_size</code> . 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

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

Listing 6: 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: 2.
message_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.

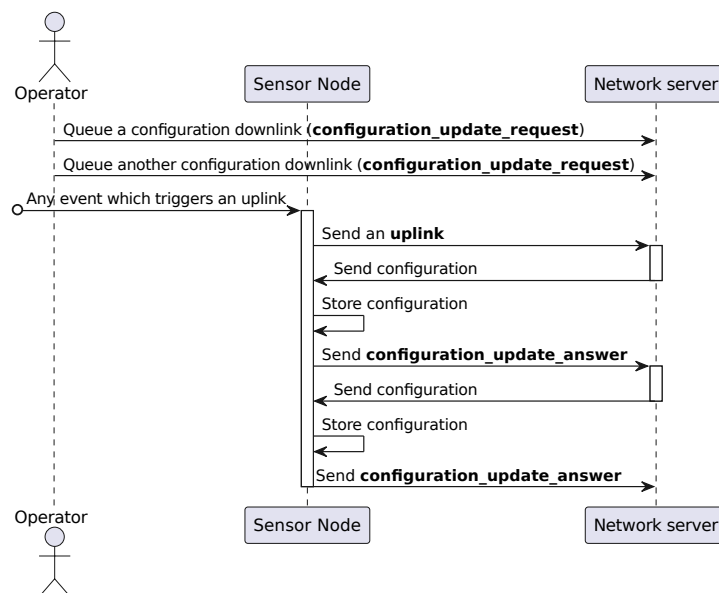


Figure 3.1: Update of Multiple Configurations

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

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

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

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

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

Listing 8: 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: 0.
message_version	uint4	4..7	Message version: 0.
tag	uint32	8..39	Value used to identify the configuration.
type	uint12	40..51	Type of the configuration sent through the configuration_update_request . For value mapping see the table below.
status	uint4	52..55	Status of the configuration update. For value mapping see the table below.

Enum Value Mappings

Table 3.3: Value mapping of **device_configuration_type_v0**

Value mapping	Value
"transmitter"	1
"vb_alert"	3
"vb_spectrum_alert"	4
"vb_asset"	5

Table 3.4: Value mapping of **configuration_update_status_v0**

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

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.

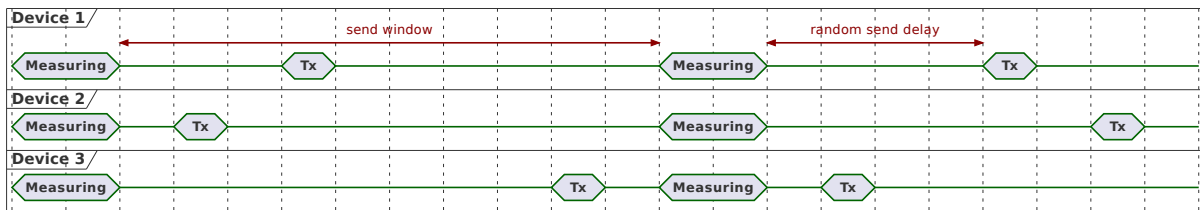


Figure 4.1: Synchronized Measurement

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.

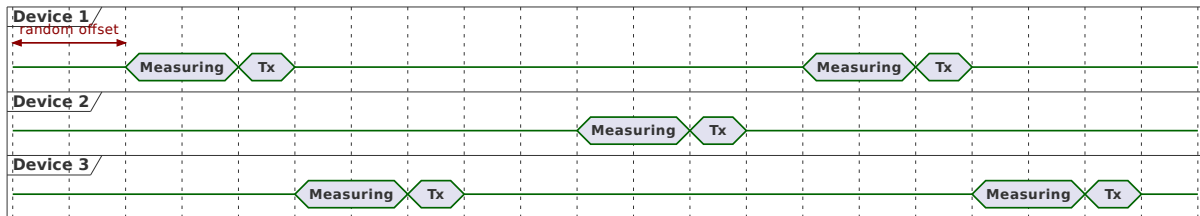


Figure 4.2: Unsynchronized Measurement

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 Update

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

The **schedule_update_request** downlink is used to change the schedules of the device. It puts a schedule in a specific slot on the device. The maximum number of available slots varies per task type, as listed in Table 4.1.

Table 4.1: Supported Tasks

Task	Number of schedule slots	Encoded Value
transmitter_status	1	0
transmitter_battery	1	15
vb_triaxial_measurement	6	23
vb_measurement	6	1
vb_measurement_envelope	6	22
vb_machine_fault_indicator	2	2
vb_spectrum	2	3
vb_statistics	1	24

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.

4.3.1 schedule_update_request message

Message Structure

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

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

Binary Encoding and Description

Table 4.2: Binary encoding for **schedule_update_request** message sent on FPort 11

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

Continues on next page

Table 4.2 Continued: Binary encoding for **schedule_update_request** message sent on FPort 11

Field ID	Type	Bit Index	Description
task		112..	Scheduled task and its parameters. The definition varies by the task type, as explained in the relevant sections. All tasks share a common header structure, including 12 bits for the task type and 4 bits for the task version. The supported tasks are listed in Table 4.1.

Enum Value Mappings

Table 4.3: Value mapping of **schedule_set_command_v0**

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

4.3.2 schedule_update_answer message

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

Message Structure

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

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

Binary Encoding and Description

Table 4.4: Binary encoding for **schedule_update_answer** message sent on FPort 11

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 2.
message_version	uint4	4..7	Message version: 0.
slot	uint4	8..11	The slot that was used in the request.
task_type	uint12	12..23	Type of the schedule sent through the schedule_update_request . For value mapping see the table below.
status	uint4	24..27	Status of the schedule update. For value mapping see the table below.
number_of_slots	uint4	28..31	Total number of slots available for this task. See Table 4.1.
slot_0_available	bool	32	If true , the slot 0 is free. No schedule is saved in this slot. Similarly, slot_N_available indicates if the slot N is free. Only supported slots are included in the decoded object, based on number_of_slots .
slot_1_available	bool	33	See slot_0_available.
slot_2_available	bool	34	See slot_0_available.
slot_3_available	bool	35	See slot_0_available.
slot_4_available	bool	36	See slot_0_available.
slot_5_available	bool	37	See slot_0_available.

Continues on next page

Table 4.4 Continued: Binary encoding for **schedule_update_answer** message sent on FPort 11

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

Enum Value Mappings

Table 4.5: Value mapping of **schedule_type_v0**

Value mapping	Value
"transmitter_status"	0
"transmitter_battery"	15
"vb_triaxial_measurement"	23
"vb_measurement"	1
"vb_measurement_envelope"	22
"vb_machine_fault_indicator"	2
"vb_statistics"	24
"vb_spectrum"	3

Table 4.6: Value mapping of **configuration_update_status_v0**

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

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.

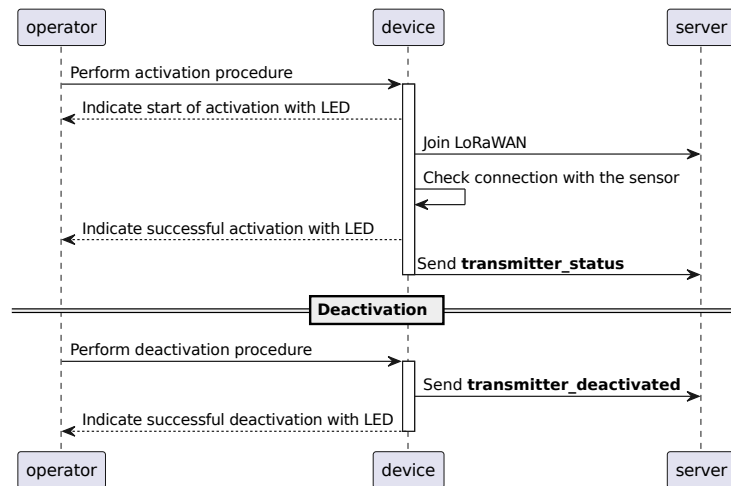


Figure 5.1: Transmitter Activation and Deactivation Sequence

During activation, the transmitter attempts to join the LoRaWAN Network Server. If the join procedure fails, the device will not be activated, and the process is aborted. Additionally, during the activation, the transmitter verifies communication with the connected sensor. If this verification fails, activation is aborted, and a **transmitter_deactivated** uplink is sent with the reason: `"activation_sensor_comm_fail"`.

5.1.1 transmitter_deactivated message

Message Structure

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

Listing 11: 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: 3.
message_version	uint4	4..7	Message version: 0.
deactivation_reason	uint8	8..15	Reason for the device's deactivation. For value mapping see the table below.

Enum Value Mappings

Table 5.2: Value mapping of **deactivation_reason_v0**

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

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 4.3.1).

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

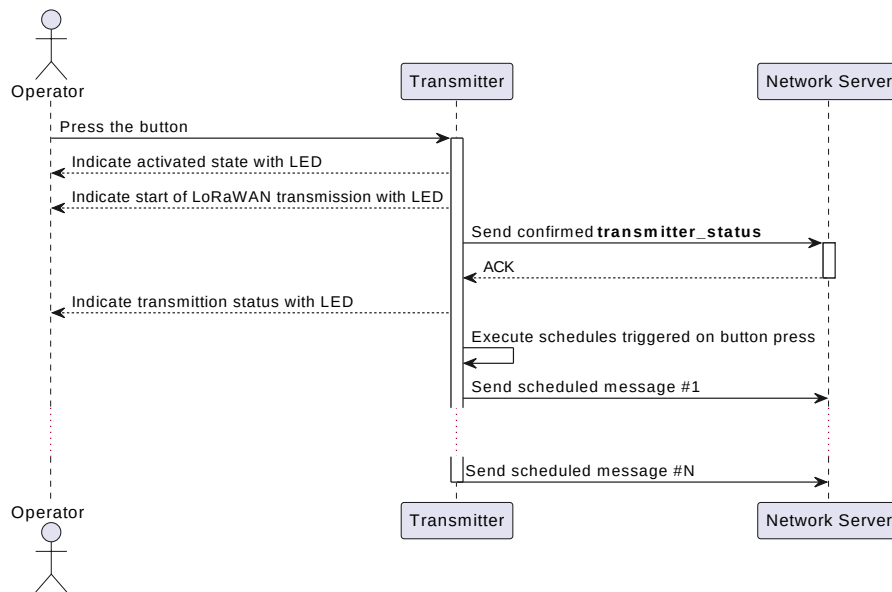


Figure 5.2: Transmitter Button Press Sequence

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

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

Listing 12: JSON structure for **transmitter_boot** message

Binary Encoding and Description

Table 5.3: Binary encoding for **transmitter_boot** message sent on FPort 16

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

Enum Value Mappings

Table 5.4: Value mapping of **reboot_reason_v0**

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

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 task

transmitter_status uplinks are scheduled using the `schedule_update_request` (as described in Section 4.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 1.

Task Settings

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

Listing 13: JSON structure for **transmitter_status** task settings

Binary Encoding and Description

Table 5.5: Binary encoding for **transmitter_status** Task settings

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

5.4.2 transmitter_status message

Message Structure

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

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

Binary Encoding and Description

Table 5.6: Binary encoding for **transmitter_status** message sent on FPort 16

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 1.
message_version	uint4	4..7	Message version: 1.
transmitter_temperature	int8	8..15	Measured transmitter temperature in °C [°F].
rssi	int8	16..23	RSSI (Received Signal Strength Indicator) of the latest downlink.
bist_power_supply	bool	24	Passed if the power supply module is operational.
bist_configuration	bool	25	Passed if the device has all mandatory configurations.
bist_sensor_connection	bool	26	Passed if the sensor is connected to the transmitter.
bist_sensor_paired	bool	27	Passed if the sensor is paired to the transmitter. See <code>require_sensor_pairing</code> in Section 5.7.1.
bist_flash_memory	bool	28	Passed if the flash memory is operational.
bist_internal_temperature_sensor	bool	29	Passed if the temperature of the microcontroller unit is within the acceptable range.

Continues on next page

Table 5.6 Continued: Binary encoding for **transmitter_status** message sent on FPort 16

Field ID	Type	Bit Index	Description
bist_time_synchronized	bool	30	Passed if the device clock is considered to have a valid time. The time is considered valid if it has been synchronized with the server at least once in the last three time synchronization intervals. After a reboot the time is considered invalid until the first time synchronization. If the device time is invalid, all tasks are scheduled with a randomized offset, and uplinks with a timestamp will indicate that the device time is invalid. In that case the server can use the message reception time as the timestamp.
bist_passed			Passed if all other BIST passed.
<i>Reserved for Future Use</i>	uint1	31	
use_case	uint8	32..39	Use case of the device. For value mapping see the table below.
protocol_major	uint4	40..43	Major version of the communication protocol implemented by the device. For the current revision of the communication protocol the major version is 4.
protocol_minor	uint4	44..47	Minor version of the communication protocol implemented by the device. For the current revision of the communication protocol the minor version is 1.

Enum Value Mappings

Table 5.7: Value mapping of **use_case_v0**

Value mapping	Value
"VB"	1

5.4.3 Example

The device is configured by default to send a `transmitter_status` uplink once per day. This configuration is applied during production. If no other schedule is set, the device will follow this default schedule.

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

Listing 15: 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 task

transmitter_battery uplinks are scheduled using the `schedule_update_request` (as described in Section 4.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 1.

Task Settings

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

Listing 16: JSON structure for **transmitter_battery** task settings

Binary Encoding and Description

Table 5.8: Binary encoding for **transmitter_battery** Task settings

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

5.5.2 transmitter_battery message

Message Structure

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

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

Binary Encoding and Description

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

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

5.5.3 transmitter_battery_reset_request message

Message Structure

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

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

Binary Encoding and Description

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

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

5.5.4 transmitter_battery_reset_answer message

Message Structure

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

Listing 19: JSON structure for **transmitter_battery_reset_answer** message

Binary Encoding and Description

Table 5.11: Binary encoding for **transmitter_battery_reset_answer** message sent on FPort 14

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

5.6 Factory Reset

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

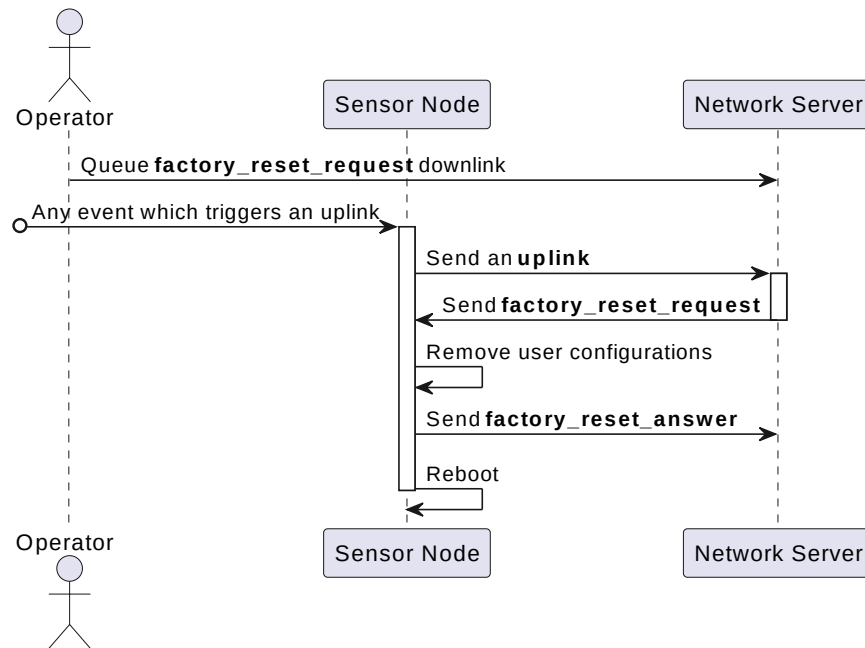


Figure 5.3: Factory Reset

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

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

Listing 20: JSON structure for **factory_reset_request** message

Binary Encoding and Description

Table 5.12: Binary encoding for **factory_reset_request** message sent on FPort 14

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

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

5.6.2 factory_reset_answer message

Message Structure

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

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

Binary Encoding and Description

Table 5.13: Binary encoding for **factory_reset_answer** message sent on FPort 14

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

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 22: JSON structure for **transmitter** configuration payload

Binary Encoding and Description

Table 5.14: Binary encoding for **transmitter** configuration payload

Field ID	Type	Bit Index	Description
type	uint12	40..51	Configuration payload type for transmitter (encoded value: 1).
version	uint4	52..55	Configuration payload version: 0.
allow_deactivation	bool	56	Allow local deactivation procedure to be executed for either disabling the sensor or replacing it with a new sensor.
require_sensor_pairing	bool	57	If true , the transmitter will only communicate with the sensor attached during the activation. Changing the attached sensor requires pairing through deactivation and activation.
enable_class_b	bool	58	If true , Class B is enabled; otherwise, the device operates in Class A. Note that Class B increases energy consumption depending on the ping-slot periodicity configured by the LNS. This is an experimental feature.
time_synchronization_interval_days	uint3	59..61	Interval in days at which the device uses the DeviceTimeReq MAC command to synchronize time with the server. Setting the interval to 0 disables time synchronization.
fragmented_uplink_redundancy_percent	uint8	62..69	Redundancy percentage of fragmented uplinks.
<i>Reserved for Future Use</i>	uint2	70..71	

Default

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

Listing 23: Example of the JSON structure for the default transmitter configuration.

6 Sensor

The NEON Vibration Sensor allows for condition-based monitoring of an asset. The sensor has functionalities ranging from basic vibration measurement to advanced fault detection.

It can:

- Measure the overall values of the vibration signal in the acceleration, (acceleration) envelope and velocity domains. [Section 6.3, Section 6.4, Section 6.5]
- Measure the temperature of the sensor. [Section 6.3, Section 6.4, Section 6.5, Section 6.8, Section 6.8]
- Keep track of statistics of the overall values of the vibration signal and the temperature of the sensor. [Section 6.7]
- Calculate and send an acceleration, envelope and velocity spectrum. [Section 6.8]
- Detect the rpm of the asset. [Section 6.8]
- Detect common fault patterns such as imbalance, misalignment, looseness etc. [Section 6.9]
- Detect bearing faults. [Section 6.9]
- Alert the user when the overall values of the vibration signal exceeds a certain threshold. [Section 6.10]
- Alert the user when the vibration levels in specific frequency bands exceed a certain threshold. [Section 6.10]

The following sections will describe the different messages that can be sent by the NEON Vibration Sensor, and how to schedule them. For the scheduling of the messages, an online configuration tool is available at <https://neon-configurator.twtg.io/neon/vb/v4/>.

6.1 Envelope Detection

Envelope detection is a signal processing technique used to extract the amplitude modulation of a vibration signal, which is especially useful for identifying repetitive impacts such as those caused by bearing faults or gear defects.

These faults cause periodic impacts every time the defect comes in contact with another part of the machine. These impacts generate high frequency vibrations on top of the healthy vibration signal. The envelope detection process extracts these high frequency vibrations from the healthy signal, so they can be analyzed separately. The envelope detection process consists of two main steps:

1. Bandpass filtering: The vibration signal is passed through a bandpass filter to isolate the frequency range of interest. These are set using `f_min_envelope_filter` and `f_max_envelope_filter`.
2. Envelope extraction: The frequencies of interest are extracted from the filtered signal.

6.2 Schedule Slots

Any scheduled task is put in a slot. The number of available slots depends on the type of the task, and is shown in Table 4.1. If multiple slots are supported for a given task type, the schedule slot is reported in the message. This way, different messages can be linked to the schedules that triggered them.

An example of this is the **triaxial_measurement** task, which has a total of 6 slots available. Since different schedules will probably have different settings, their results should be distinguished.

6.2.1 Example

Get the `rms_velocity` for all axes, in the low $[0.5 - 10]$ Hz, medium $[10 - 100]$ Hz and high $[100 - 1000]$ Hz frequency range, every hour using the **triaxial_measurement** schedule. The low frequency range will be in slot 0, the medium frequency range in slot 1 and the high frequency range in slot 2. The low frequency range will have a higher `rms_velocity` value than medium range, which will have a higher value than the high frequency range. This is expected, since the low frequency range will usually have more energy than the medium and high frequency ranges. However, this means that for proper analysis, the results should be distinguished, for instance in different graphs. The flow is visualized below.

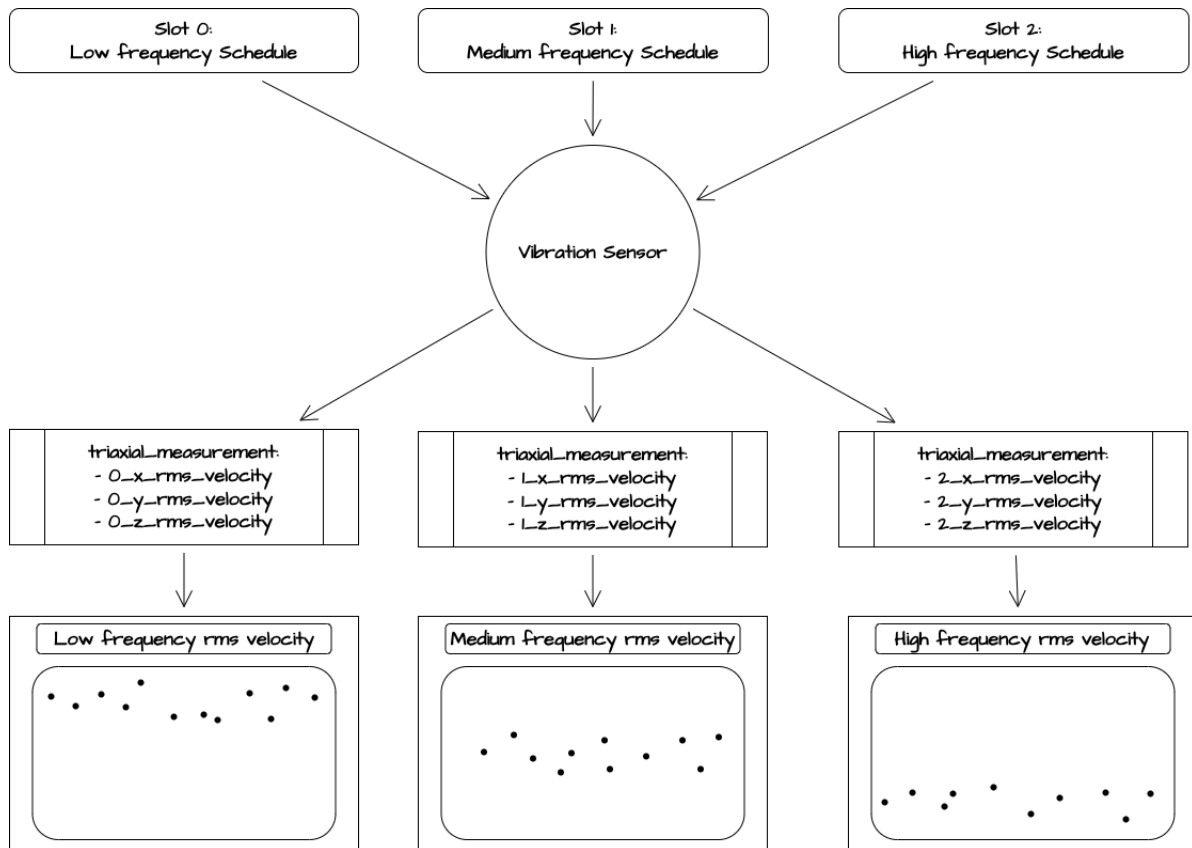


Figure 6.1: Different tasks lead to different graphs

6.3 Triaxial Measurement

The NEON Vibration Sensor can report the overall sensor values in two ways, per axis, or per sensor value. This section will describe reporting per sensor value, using the **triaxial_measurement** uplink. This message holds the selected sensor value for all three axes, and the `"sensor_temperature"`.

6.3.1 vb_triaxial_measurement task

triaxial_measurement uplinks are scheduled using the `schedule_update_request` (as described in Section 4.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 6.

Important remarks regarding spectrum measurements:

- For a set of overlapping **triaxial_measurement** schedules, only one uplink is sent per sensor value.
- For the envelope values, the `f_min_envelope_filter` and `f_max_envelope_filter` settings are used to set the frequency range of the bandpass filter that is used for the calculation of the envelope, as described in section Section 6.1. For the other other value types, these settings are ignored.

Task Settings

```
{
  "type": "vb_triaxial_measurement",
  "version": 0,
  "range": <string>,
  "sensor_value_selection": {
    "rms_velocity": <boolean>,
    "rms_acceleration": <boolean>,
    "peak_acceleration": <boolean>,
    "rms_envelope": <boolean>,
    "peak_envelope": <boolean>
  },
  "f_min": <number>,
  "f_max": <number>,
  "f_min_envelope_filter": <number>,
  "f_max_envelope_filter": <number>,
  "enable_confirmed_message": <boolean>,
  "send_condition": {
    "value_type": <string>,
    "threshold": <number>
  }
}
```

Listing 24: JSON structure for **vb_triaxial_measurement** task settings

Binary Encoding and Description

Table 6.1: Binary encoding for **vb_triaxial_measurement** Task settings

Field ID	Type	Bit Index	Description
type	uint12	112..123	Task type, for vb_triaxial_measurement : 23
version	uint4	124..127	Task version: 0.
range	uint2	128..129	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. For value mapping see the table below.
sensor_value_selection.rms_velocity	bool	130	If true , the sensor measures RMS velocity in mms^{-1} [ins^{-1}].
sensor_value_selection.rms_acceleration	bool	131	If true , the sensor measures RMS acceleration in g.
sensor_value_selection.peak_acceleration	bool	132	If true , the sensor measures peak acceleration in g.
sensor_value_selection.rms_envelope	bool	133	If true , the sensor measures RMS envelope in g.
sensor_value_selection.peak_envelope	bool	134	If true , the sensor measures peak envelope in g.
f_min	pfloat15	135..149	Lower bound of the frequency range to be analyzed in Hz.
f_max	pfloat15	150..164	Upper bound of the frequency range to be analyzed in Hz.
f_min_envelope_filter	pfloat15	165..179	Lower bound of the frequency range to be analyzed in Hz.
f_max_envelope_filter	pfloat15	180..194	Upper bound of the frequency range to be analyzed in Hz.
enable_confirmed_message	bool	195	Enable confirmed messages.
send_condition.value_type	uint4	196..199	Type of the <i>send condition</i> for the schedule. The device only transmits the scheduled up-link if it is satisfied. If set to "always" , the scheduled messages are sent unconditionally. For value mapping see the table below.
send_condition.threshold	float16	200..215	Value that needs to be reached for the send condition. To be used in combination with send_condition.value_type .

Enum Value Mappings

Table 6.2: Value mapping of **vb_range_v0**

Value mapping	Value
"gscale_2"	0
"gscale_4"	1
"gscale_8"	2
"gscale_16"	3

Table 6.3: Value mapping of **vb_send_condition_v0**

Value mapping	Value
"always"	0
"peak_acceleration_above"	1
"rms_acceleration_above"	2
"rms_velocity_above"	3
"temperature_above"	4
"temperature_below"	5

6.3.2 triaxial-measurement message

The **triaxial_measurement** message contains the specified "sensor_value" calculated from a vibration signal measured according to the scheduled "vb_triaxial_measurement" task. Note that one schedule can result in multiple uplinks, one for each "sensor_value", as specified by the "sensor_value_selection" bitmask.

Message Structure

```
{
  "message_name": "triaxial_measurement",
  "message_version": 0,
  "timestamp": <string>,
  "sensor_temperature": <number>,
  "<slot>_x_<sensor_value_selection>": <number>,
  "<slot>_y_<sensor_value_selection>": <number>,
  "<slot>_z_<sensor_value_selection>": <number>
}
```

Listing 25: JSON structure for **triaxial_measurement** message

Binary Encoding and Description

Table 6.4: Binary encoding for **triaxial_measurement** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 8.
message_version	uint4	4..7	Message version: 0.
slot	uint4	8..11	The slot of the scheduled task.
timestamp	short_timestamp	12..27	UTC ISO timestamp of the measurement. Detailed information about the binary encoding is provided in Section 1.3.1.
sensor_value_selection	uint4	28..31	Type of the measurement. For value mapping see the table below.
sensor_temperature	int8	32..39	Measured sensor temperature in °C [°F].
x	float16	40..55	Measured value in g or mms^{-1} [ins^{-1}] for the x axis depending on selection.
y	float16	56..71	Measured value in g or mms^{-1} [ins^{-1}] for the y axis depending on selection.
z	float16	72..87	Measured value in g or mms^{-1} [ins^{-1}] for the z axis depending on selection.

Enum Value Mappings

Table 6.5: Value mapping of `vb_sensor_value_selection_v0`

Value mapping	Value
"rms_velocity"	0
"rms_acceleration"	1
"peak_acceleration"	2
"rms_envelope"	3
"peak_envelope"	4

6.3.3 Example

To send a **triaxial_measurement** uplink for RMS velocity, RMS acceleration, peak acceleration, RMS envelope, and peak envelope every hour.

```
{
  "message_name": "schedule_update_request",
  "message_version": 0,
  "slot": 0,
  "command": "set",
  "timing": 60,
  "triggered_on_button_press": false,
  "send": true,
  "task": {
    "type": "vb_triaxial_measurement",
    "version": 0,
    "sensor_value_selection": {
      "rms_velocity": true,
      "rms_acceleration": true,
      "peak_acceleration": true,
      "rms_envelope": true,
      "peak_envelope": true
    },
    "range": "gscale_16",
    "f_min": 10.0,
    "f_max": 6600.0,
    "f_min_envelope": 500,
    "f_max_envelope": 6600.0,
    "enable_confirmed_message": false,
    "send_condition": {
      "value_type": "always",
      "threshold": 0.0
    }
  }
}
```

Listing 26: Example of the JSON structure for scheduling a **triaxial_measurement** uplink.

The following is an example of a **triaxial_measurement** uplink generated by this schedule. Note that this schedule results in multiple uplinks, one for each **"sensor_value_selection"**.

```
{
  "message_name": "triaxial_measurement",
  "message_version": 0,
  "timestamp": "2025-04-02T14:43:00.000Z",
  "sensor_temperature": 20,
  "0_x_rms_velocity": 0.2326,
  "0_y_rms_velocity": 0.2352,
  "0_z_rms_velocity": 0.5242
}
```

Listing 27: Example of the JSON structure for **triaxial_measurement** uplink.

6.4 Measurement

The NEON Vibration Sensor can report the overall sensor values in two ways, per axis, or per sensor value. This section will describe reporting per axis, using the **measurement** uplink. This message holds the "rms_velocity", "rms_acceleration", "peak_acceleration" for one axis.

6.4.1 vb_measurement task

measurement uplinks are scheduled using the `schedule_update_request` (as described in Section 4.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 6.

For a set of overlapping **measurement** tasks, only one uplink is sent per axis.

Task Settings

```
{
  "type": "vb_measurement",
  "version": 1,
  "axis": <string>,
  "range": <string>,
  "f_min": <number>,
  "f_max": <number>,
  "enable_confirmed_message": <boolean>,
  "send_condition": {
    "value_type": <string>,
    "threshold": <number>
  }
}
```

Listing 28: JSON structure for **vb_measurement** task settings

Binary Encoding and Description

Table 6.6: Binary encoding for **vb_measurement** Task settings

Field ID	Type	Bit Index	Description
type	uint12	112..123	Task type, for vb_measurement : 1
version	uint4	124..127	Task version: 1.
axis	uint2	128..129	Axis for which the vibration is measured. If "all" is selected, the schedule will result in one message for each of the three axes. For value mapping see the table below.
range	uint2	130..131	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. For value mapping see the table below.
f_min	pfloat15	132..146	Lower bound of the frequency range to be analyzed in Hz.
f_max	pfloat15	147..161	Upper bound of the frequency range to be analyzed in Hz.
enable_confirmed_message	bool	162	Enable confirmed messages.
send_condition.value_type	uint4	163..166	Type of the <i>send condition</i> for the schedule. The device only transmits the scheduled uplink if it is satisfied. If set to "always", the scheduled messages are sent unconditionally. For value mapping see the table below.
send_condition.threshold	float16	167..182	Value that needs to be reached for the send condition. To be used in combination with <i>send_condition.value_type</i> .

Enum Value Mappings

Table 6.7: Value mapping of **vb_config_axis_v0**

Value mapping	Value
"x"	0
"y"	1
"z"	2
"all"	3

Table 6.8: Value mapping of **vb_range_v0**

Value mapping	Value
"gscale_2"	0
"gscale_4"	1
"gscale_8"	2
"gscale_16"	3

Table 6.9: Value mapping of **vb_send_condition_v0**

Value mapping	Value
"always"	0
"peak_acceleration_above"	1
"rms_acceleration_above"	2
"rms_velocity_above"	3
"temperature_above"	4
"temperature_below"	5

6.4.2 measurement message

The **measurement** message contains the overall values for a vibration signal measured according to the scheduled "vb_measurement" task. The **slot** links the **measurement** message to the **measurement** task that triggered it.

Message Structure

```
{
  "message_name": "measurement",
  "message_version": 1,
  "timestamp": <string>,
  "sensor_temperature": <number>,
  "<slot>_<axis>_peak_acceleration": <number>,
  "<slot>_<axis>_rms_acceleration": <number>,
  "<slot>_<axis>_rms_velocity": <number>
}
```

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

Binary Encoding and Description

Table 6.10: Binary encoding for **measurement** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 1.
message_version	uint4	4..7	Message version: 1.
slot	uint4	8..11	The slot of the scheduled task.
timestamp	short_timestamp	12..27	UTC ISO timestamp of the measurement. Detailed information about the binary encoding is provided in Section 1.3.1.
axis	uint2	28..29	Axis for which the vibration is measured. For value mapping see the table below.
sensor_temperature	int8	30..37	Measured sensor temperature in °C [°F].
peak_acceleration	float16	38..53	Measured peak acceleration in g.
rms_acceleration	float16	54..69	Measured RMS acceleration in g.
rms_velocity	float16	70..85	Measured RMS velocity in mms^{-1} [ins^{-1}].
Reserved for Future Use	uint2	86..87	

Enum Value Mappings

Table 6.11: Value mapping of **vb_axis_v0**

Value mapping	Value
"x"	0
"y"	1
"z"	2

6.4.3 Example

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

```
{
  "message_name": "schedule_update_request",
  "message_version": 0,
  "slot": 0,
  "command": "set",
  "timing": 60,
  "triggered_on_button_press": false,
  "send": true,
  "task": {
    "type": "vb_measurement",
    "version": 1,
    "axis": "all",
    "range": "gscale_16",
    "f_min": 10.0,
    "f_max": 6600.0,
    "enable_confirmed_message": false,
    "send_condition": {
      "value_type": "always",
      "threshold": 0.0
    }
  }
}
```

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

The following is an example of a **measurement** uplink generated by this schedule. Note that this schedule results in multiple uplinks, one for each axis.

```
{
  "message_name": "measurement",
  "message_version": 1,
  "timestamp": "2023-07-27T18:47:00.000Z",
  "sensor_temperature": -12,
  "0_x_peak_acceleration": 0.035675048828125,
  "0_x_rms_acceleration": 0.01322174072265625,
  "0_x_rms_velocity": 0.33447265625
}
```

Listing 31: Example of the JSON structure for **measurement** uplink.

6.5 Measurement Envelope

The NEON Vibration Sensor can report the overall sensor values in two ways, per axis, or per sensor value. This section will describe reporting per axis, using the **measurement_envelope** uplink. This message holds the `"rms_envelope"`, `"peak_envelope"`, `"rms_velocity"` for one axis, and the `"sensor_temperature"`.

6.5.1 vb_measurement_envelope task

measurement_envelope uplinks are scheduled using the `schedule_update_request` (as described in Section 4.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 6.

For a set of overlapping **measurement_envelope** tasks, only one uplink is sent per axis.

Task Settings

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

Listing 32: JSON structure for **vb_measurement_envelope** task settings

Binary Encoding and Description

Table 6.12: Binary encoding for **vb_measurement_envelope** Task settings

Field ID	Type	Bit Index	Description
type	uint12	112..123	Task type, for vb_measurement_envelope : 22
version	uint4	124..127	Task version: 0.
axis	uint2	128..129	Axis for which the vibration is measured. If "all" is selected, the schedule will result in one message for each of the three axes. For value mapping see the table below.
range	uint2	130..131	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. For value mapping see the table below.
f_min	pfloat15	132..146	Lower bound of the frequency range to be analyzed in Hz.
f_max	pfloat15	147..161	Upper bound of the frequency range to be analyzed in Hz.
f_min_envelope_filter	pfloat15	162..176	Lower bound of the frequency range to be analyzed in Hz.
f_max_envelope_filter	pfloat15	177..191	Upper bound of the frequency range to be analyzed in Hz.
enable_confirmed_message	bool	192	Enable confirmed messages.
send_condition.value_type	uint4	193..196	Type of the <i>send condition</i> for the schedule. The device only transmits the scheduled uplink if it is satisfied. If set to "always", the scheduled messages are sent unconditionally. For value mapping see the table below.
send_condition.threshold	float16	197..212	Value that needs to be reached for the send condition. To be used in combination with <i>send_condition.value_type</i> .

Enum Value Mappings

Table 6.13: Value mapping of **vb_config_axis_v0**

Value mapping	Value
"x"	0
"y"	1
"z"	2
"all"	3

Table 6.14: Value mapping of **vb_range_v0**

Value mapping	Value
"gscale_2"	0
"gscale_4"	1
"gscale_8"	2
"gscale_16"	3

Table 6.15: Value mapping of **vb_send_condition_v0**

Value mapping	Value
"always"	0
"peak_acceleration_above"	1
"rms_acceleration_above"	2
"rms_velocity_above"	3
"temperature_above"	4
"temperature_below"	5

6.5.2 measurement_envelope message

The **measurement_envelope** message contains the overall values for a vibration signal measured according to the scheduled "vb_measurement_envelope" task.

Message Structure

```
{
  "message_name": "measurement_envelope",
  "message_version": 0,
  "timestamp": <string>,
  "sensor_temperature": <number>,
  "<slot>_<axis>_peak_envelope": <number>,
  "<slot>_<axis>_rms_envelope": <number>,
  "<slot>_<axis>_rms_velocity": <number>
}
```

Listing 33: JSON structure for **measurement_envelope** message

Binary Encoding and Description

Table 6.16: Binary encoding for **measurement_envelope** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 7.
message_version	uint4	4..7	Message version: 0.
slot	uint4	8..11	The slot of the scheduled task.
timestamp	short_timestamp	12..27	UTC ISO timestamp of the measurement. Detailed information about the binary encoding is provided in Section 1.3.1.
axis	uint2	28..29	Axis for which the vibration is measured. For value mapping see the table below.
sensor_temperature	int8	30..37	Measured sensor temperature in °C [°F].
peak_envelope	float16	38..53	Measured peak envelope in g.
rms_envelope	float16	54..69	Measured RMS envelope in g.
rms_velocity	float16	70..85	Measured RMS velocity in mms^{-1} [ins^{-1}].
<i>Reserved for Future Use</i>	uint2	86..87	

Enum Value Mappings

Table 6.17: Value mapping of **vb_axis_v0**

Value mapping	Value
"x"	0
"y"	1
"z"	2

6.5.3 Example

To send a **measurement_envelope** uplink for the x, y, and z axis every hour.

```
{
  "message_name": "schedule_update_request",
  "message_version": 0,
  "slot": 0,
  "command": "set",
  "timing": 60,
  "triggered_on_button_press": false,
  "send": true,
  "task": {
    "type": "vb_measurement_envelope",
    "version": 0,
    "axis": "all",
    "range": "gscale_16",
    "f_min": 10.0,
    "f_max": 6600.0,
    "f_min_envelope_filter": 500.0,
    "f_max_envelope_filter": 6600.0,
    "enable_confirmed_message": false,
    "send_condition": {
      "value_type": "always",
      "threshold": 0.0
    }
  }
}
```

Listing 34: Example of the JSON structure for scheduling a **measurement_envelope** uplink.

The following is an example of a **measurement_envelope** uplink generated by this schedule. Note that this schedule results in multiple uplinks, one for each axis.

```
{
  "message_name": "measurement_envelope",
  "message_version": 0,
  "timestamp": "2023-07-27T18:47:00.000Z",
  "sensor_temperature": -12,
  "0_x_peak_envelope": 0.035675048828125,
  "0_x_rms_envelope": 0.01322174072265625,
  "0_x_rms_velocity": 0.33447265625
}
```

Listing 35: Example of the JSON structure for **measurement_envelope** uplink.

6.6 Past Measurement Request

The sensor stores a **measurement** uplink or a **triaxial_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** or **triaxial_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. The device is capable of storing up to 1000 measurements.
- It is not possible to request a measurement older than 21 days (due to **short_timestamp** type restrictions, see Section 1.3.1).

6.6.1 past_measurement_request message

Message Structure

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

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

Binary Encoding and Description

Table 6.18: Binary encoding for **past_measurement_request** message sent on FPort 14

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 0.
message_version	uint4	4..7	Message version: 0.
timestamp	timestamp	8..39	UTC ISO timestamp of the measurement.

6.7 Statistics

The NEON Vibration Sensor can keep track of the average and extremes of all overall sensor values described in Section 6.4. This is enabled by scheduling the **vb_statistics** task. It specifies which sensor values to track, over what duration, and over how many measurements.

Key Features:

- Monitors average, minimum, and maximum values.
- Aggregates data from measurements started by this schedule since the last **statistics** uplink for the given type.
- Only measurements for which the `statistics_update_condition` is satisfied contribute to the statistics.
- Results in one statistics message type per tracked statistic.
- Statistics duration can be up to 3 weeks.
- There is only one statistics schedule to set all statistics.
- The `timing` specified in the `schedule_update_request` is the time between two measurements started by the schedule. Once `number_of_measurements` is reached, the statistics uplink is sent.

When a statistics uplink is sent, the statistics for that type are reset. The reception timestamp should be used to determine the timing of the data.

6.7.1 vb_statistics task

statistics uplinks are scheduled using the `schedule_update_request` (as described in Section 4.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 1.

Task Settings

```
{
  "type": "vb_statistics",
  "version": 0,
  "range": <string>,
  "f_min": <number>,
  "f_max": <number>,
  "f_min_envelope_filter": <number>,
  "f_max_envelope_filter": <number>,
  "number_of_measurements": <number>,
  "statistics_selection": {
    "x_rms_velocity": <boolean>,
    "x_rms_acceleration": <boolean>,
    "x_peak_acceleration": <boolean>,
    "x_rms_envelope": <boolean>,
    "x_peak_envelope": <boolean>,
    "y_rms_velocity": <boolean>,
    "y_rms_acceleration": <boolean>,
    "y_peak_acceleration": <boolean>,
    "y_rms_envelope": <boolean>,
    "y_peak_envelope": <boolean>,
    "z_rms_velocity": <boolean>,
    "z_rms_acceleration": <boolean>,
    "z_peak_acceleration": <boolean>,
    "z_rms_envelope": <boolean>,
    "z_peak_envelope": <boolean>,
    "temperature": <boolean>
  },
  "statistics_update_condition": {
    "value_type": <string>,
    "threshold": <number>
  }
}
```

Listing 37: JSON structure for **vb_statistics** task settings

Binary Encoding and Description

Table 6.19: Binary encoding for **vb_statistics** Task settings

Field ID	Type	Bit Index	Description
type	uint12	112..123	Task type, for vb_statistics : 24
version	uint4	124..127	Task version: 0.
range	uint2	128..129	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. For value mapping see the table below.
f_min	pfloat15	130..144	Lower bound of the frequency range to be analyzed in Hz.

Continues on next page

Table 6.19 Continued: Binary encoding for **vb_statistics** Task settings

Field ID	Type	Bit Index	Description
f_max	pfloat15	145..159	Upper bound of the frequency range to be analyzed in Hz.
f_min_envelope_filter	pfloat15	160..174	Lower bound of the frequency range to be analyzed in Hz.
f_max_envelope_filter	pfloat15	175..189	Upper bound of the frequency range to be analyzed in Hz.
number_of_measurements	uint8	190..197	Number of measurements to be used for the statistics. Note that the timing specified in the <code>schedule_update_request</code> is the time between two measurements started by the schedule. Once <code>number_of_measurements</code> is reached, the statistics uplink is sent.
statistics_selection.x_rms_velocity	bool	198	If <code>true</code> , the sensor measures RMS velocity in mms^{-1} [ins^{-1}] for the x axis.
statistics_selection.x_rms_acceleration	bool	199	If <code>true</code> , the sensor measures RMS acceleration in g for the x axis.
statistics_selection.x_peak_acceleration	bool	200	If <code>true</code> , the sensor measures peak acceleration in g for the x axis.
statistics_selection.x_rms_envelope	bool	201	If <code>true</code> , the sensor measures RMS envelope in g for the x axis.
statistics_selection.x_peak_envelope	bool	202	If <code>true</code> , the sensor measures peak envelope in g for the x axis.
statistics_selection.y_rms_velocity	bool	203	If <code>true</code> , the sensor measures RMS velocity in mms^{-1} [ins^{-1}] for the y axis.
statistics_selection.y_rms_acceleration	bool	204	If <code>true</code> , the sensor measures RMS acceleration in g for the y axis.
statistics_selection.y_peak_acceleration	bool	205	If <code>true</code> , the sensor measures peak acceleration in g for the y axis.
statistics_selection.y_rms_envelope	bool	206	If <code>true</code> , the sensor measures RMS envelope in g for the y axis.
statistics_selection.y_peak_envelope	bool	207	If <code>true</code> , the sensor measures peak envelope in g for the y axis.

Continues on next page

Table 6.19 Continued: Binary encoding for **vb_statistics** Task settings

Field ID	Type	Bit Index	Description
<code>statistics_selection.z_rms_velocity</code>	bool	208	If <code>true</code> , the sensor measures RMS velocity in mms^{-1} [ins^{-1}] for the z axis.
<code>statistics_selection.z_rms_acceleration</code>	bool	209	If <code>true</code> , the sensor measures RMS acceleration in g for the z axis.
<code>statistics_selection.z_peak_acceleration</code>	bool	210	If <code>true</code> , the sensor measures peak acceleration in g for the z axis.
<code>statistics_selection.z_rms_envelope</code>	bool	211	If <code>true</code> , the sensor measures RMS envelope in g for the z axis.
<code>statistics_selection.z_peak_envelope</code>	bool	212	If <code>true</code> , the sensor measures peak envelope in g for the z axis.
<code>statistics_selection.temperature</code>	bool	213	If <code>true</code> , the sensor measures temperature in $^{\circ}\text{C}$ [$^{\circ}\text{F}$].
<code>statistics_update_condition.value_type</code>	uint4	214..217	Type of the <i>statistics_update_condition</i> for the schedule. The device only updates the statistics with the measured value if it is satisfied. If set to "always", the statistics are updated unconditionally. This can be used to ignore values that were measured when the asset is off. This is the same as the <i>send_condition</i> . For value mapping see the table below.
<code>statistics_update_condition.threshold</code>	float16	218..233	Value that needs to be reached for the <i>statistics_update_condition</i> .

Enum Value Mappings

Table 6.20: Value mapping of **vb_range_v0**

Value mapping	Value
"gscale_2"	0
"gscale_4"	1
"gscale_8"	2
"gscale_16"	3

Table 6.21: Value mapping of **vb_send_condition_v0**

Value mapping	Value
"always"	0
"peak_acceleration_above"	1
"rms_acceleration_above"	2
"rms_velocity_above"	3
"temperature_above"	4
"temperature_below"	5

6.7.2 statistics message

Message Structure

```
{
  "message_name": "statistics",
  "message_version": 0,
  "<selection>_min": <number>,
  "<selection>_max": <number>,
  "<selection>_avg": <number>,
  "<selection>_max_timestamp": <string>
}
```

Listing 38: JSON structure for **statistics** message

Binary Encoding and Description

Table 6.22: Binary encoding for **statistics** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 4.
message_version	uint4	4..7	Message version: 0.
selection	uint4	8..11	Type of the statistics. For value mapping see the table below.
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.
<i>Reserved for Future Use</i>	uint4	76..79	

Enum Value Mappings

Table 6.23: Value mapping of `vb_statistics_selection_v0`

Value mapping	Value
"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
"x_rms_envelope"	10
"x_peak_envelope"	11
"y_rms_envelope"	12
"y_peak_envelope"	13
"z_rms_envelope"	14
"z_peak_envelope"	15

6.7.3 Example

To send a **statistics** uplink for all sensor values every day, with a measurement every 15 minutes. Since there can only be one statistics schedule, the slot is always 0.

```
{
  "message_name": "schedule_update_request",
  "message_version": 0,
  "slot": 0,
  "command": "set",
  "timing": 15,
  "triggered_on_button_press": false,
  "send": true,
  "task": {
    "type": "vb_statistics",
    "version": 0,
    "f_min": 10.0,
    "f_max": 6600.0,
    "number_of_measurements": 96,
    "statistics_selection": {
      "x_rms_velocity": true,
      "x_rms_acceleration": true,
      "x_peak_acceleration": true,
      "x_rms_envelope": true,
      "x_peak_envelope": true,
      "y_rms_velocity": true,
      "y_rms_acceleration": true,
      "y_peak_acceleration": true,
      "y_rms_envelope": true,
      "y_peak_envelope": true,
      "z_rms_velocity": true,
      "z_rms_acceleration": true,
      "z_peak_acceleration": true,
      "z_rms_envelope": true,
      "z_peak_envelope": true,
      "temperature": true
    },
    "statistics_update_condition": {
      "value_type": "always",
      "threshold": 0.0
    }
  }
}
```

Listing 39: Example of the JSON structure for scheduling a **statistics** uplink for all possible sensor values..

6.8 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. 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. Alert triggered
 3. Schedule triggered
- For the envelope spectrum, the `f_min_envelope_filter` and `f_max_envelope_filter` settings are used to set the frequency range of the bandpass filter that is used for the calculation of the envelope, as described in section Section 6.1. For other spectrum types, these settings are ignored.

6.8.1 Resolution and Frequency Range

The frequency range of the spectrum is set by the `f_min`, `f_max` settings. These settings set the -3dB points of the measurement over which the spectrum is calculated. Since the rest is filtered out, the spectrum that is sent will only contain the frequencies between `f_min` and `f_max`. Together with the `num_spectral_lines` setting, these settings determine the resolution of the spectrum. The `num_spectral_lines` setting determines the number of spectral lines in the spectrum, so the size of the spectrum uplink. Not all combinations of these settings are possible. For a detailed explanation of the calculation of possible combinations. Please use the online configuration tool at <https://neon-configurator.twtg.io/neon/vb/v4.1/>, which handles these calculations.

6.8.2 vb_spectrum task

spectrum uplinks are scheduled using the `schedule_update_request` (as described in Section 4.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 6.

Task Settings

```
{
  "type": "vb_spectrum",
  "version": 1,
  "axis": <string>,
  "range": <string>,
  "num_spectral_lines": <number>,
  "f_min": <number>,
  "f_max": <number>,
  "f_min_envelope_filter": <number>,
  "f_max_envelope_filter": <number>,
  "spectrum_type": <string>,
  "averaging": <number>,
  "time_to_transmit_min": <number>,
  "send_condition": {
    "value_type": <string>,
    "threshold": <number>,
    "hold_off_hours": <number>
  }
}
```

Listing 40: JSON structure for **vb_spectrum** task settings

Binary Encoding and Description

Table 6.24: Binary encoding for **vb_spectrum** Task settings

Field ID	Type	Bit Index	Description
type	uint12	112..123	Task type, for vb_spectrum : 3
version	uint4	124..127	Task version: 1.
axis	uint2	128..129	Axis for which the vibration is measured. For value mapping see the table below.
range	uint2	130..131	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. For value mapping see the table below.
num_spectral_lines	uint14	132..145	The number of lines in the output spectrum. It is recommended to use the online configurator for the calculation of the allowed values for a required <code>f_min</code> and <code>f_max</code> .
f_min	pfloat15	146..160	Lower bound of the frequency range to be analyzed and/or sent in Hz.

Continues on next page

Table 6.24 Continued: Binary encoding for **vb_spectrum** Task settings

Field ID	Type	Bit Index	Description
f_max	pfloat15	161..175	Upper bound of the frequency range to be analyzed and/or sent in Hz.
f_min_envelope_filter	pfloat15	176..190	Lower bound of the frequency range to be analyzed and/or sent in Hz.
f_max_envelope_filter	pfloat15	191..205	Upper bound of the frequency range to be analyzed and/or sent in Hz.
spectrum_type	uint2	206..207	Type of frequency spectrum. For value mapping see the table below.
averaging	uint3	208..210	Amount of spectrums to average over. Used to reduce noise in the spectrums.
time_to_transmit_min	pfloat15	211..225	Specifies the duration of time used to transmit the fragmented uplink in min.
send_condition.value_type	uint4	226..229	Type of the <i>send condition</i> for the schedule. The device only transmits the scheduled uplink if it is satisfied. If set to "always", the scheduled messages are sent unconditionally. For value mapping see the table below.
send_condition.threshold	float16	230..245	Value that needs to be reached for the send condition. To be used in combination with <i>send_condition.value_type</i> .
send_condition.hold_off_hours	uint10	246..255	Time to wait before sending a new spectrum message triggered by the <i>send_condition</i> h.

Enum Value Mappings

Table 6.25: Value mapping of **vb_axis_v0**

Value mapping	Value
"x"	0
"y"	1
"z"	2

Table 6.26: Value mapping of **vb_range_v0**

Value mapping	Value
"gscale_2"	0
"gscale_4"	1
"gscale_8"	2
"gscale_16"	3

Table 6.27: Value mapping of **vb_spectrum_type_v0**

Value mapping	Value
"acceleration"	0
"velocity"	1
"envelope"	2

Table 6.28: Value mapping of **vb_send_condition_v0**

Value mapping	Value
"always"	0
"peak_acceleration_above"	1
"rms_acceleration_above"	2
"rms_velocity_above"	3
"temperature_above"	4
"temperature_below"	5

6.8.3 spectrum message

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

If multiple "vb_spectrum" tasks 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. Alert triggered
3. Schedule triggered

Message Structure

```
{
  "message_name": "spectrum",
  "message_version": 1,
  "timestamp": <string>,
  "trigger_reason": <string>,
  "slot": <number>,
  "sensor_temperature": <number>,
  "<axis>_peak_acceleration": <number>,
  "<axis>_rms_acceleration": <number>,
  "<axis>_rms_velocity": <number>,
  "<axis>_rpm": <number>,
  "<axis>_<spectrum_type>_frequencies": <array of numbers>,
  "<axis>_<spectrum_type>_magnitudes": <array of numbers>
}
```

Listing 41: JSON structure for **spectrum** message

Binary Encoding and Description

Table 6.29: Binary encoding for **spectrum** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 5.
message_version	uint4	4..7	Message version: 1.
timestamp	timestamp	8..39	UTC ISO timestamp of the measurement.
trigger_reason	uint3	40..42	Reason for the spectrum uplink. For value mapping see the table below.
slot	uint4	43..46	The slot of the scheduled task.
axis	uint2	47..48	Axis for which the vibration is measured. For value mapping see the table below.
spectrum_type	uint2	49..50	Type of frequency spectrum. For value mapping see the table below.
sensor_temperature	int8	51..58	Measured sensor temperature in °C [°F].
df	float16	59..74	Frequency bin size in Hz.
f_min	float16	75..90	Lower bound of the frequency range to be analyzed and/or sent in Hz.

Continues on next page

Table 6.29 Continued: Binary encoding for **spectrum** message sent on FPort 17

Field ID	Type	Bit Index	Description
peak_acceleration	float16	91..106	Measured peak acceleration in g.
rms_acceleration	float16	107..122	Measured RMS acceleration in g.
rms_velocity	float16	123..138	Measured RMS velocity in mms^{-1} [ins^{-1}].
rpm	float32	139..170	Estimated rotating speed in rpm. Bound by rpm_min and rpm_max set in the vb_asset configuration (see Section 6.12.1).
magnitudes_scaling	pfloat15	171..185	Scaling value used to convert magnitude_values to Hz, based on the maximum magnitude measured.
<i>Reserved for Future Use</i>	uint6	186..191	
magnitude_values	uint10[]	192..	Array with the scaled values of the spectrum calculated.
frequencies			Array of spectrum frequencies: $\text{frequencies}[i] = f_{\text{min}} + df \times i$ where i is the index of magnitude_values (starting from 0).
magnitudes			Array of spectrum magnitudes: $\text{magnitudes}[i] = \text{magnitude_values}[i] \times \text{magnitudes_scaling}$ where i is the index of magnitude_values (starting from 0).

Enum Value Mappings

Table 6.30: Value mapping of **vb_trigger_reason_v0**

Value mapping	Value
"periodic"	0
"spectrum_alert"	1
"machine_fault"	2
"sensor_alert"	3

Table 6.31: Value mapping of **vb_axis_v0**

Value mapping	Value
"x"	0
"y"	1
"z"	2

Table 6.32: Value mapping of **vb_spectrum_type_v0**

Value mapping	Value
"acceleration"	0
"velocity"	1
"envelope"	2

6.8.4 Example

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

```
{
  "message_name": "schedule_update_request",
  "message_version": 0,
  "slot": 0,
  "command": "set",
  "timing": 10080,
  "triggered_on_button_press": false,
  "send": true,
  "task": {
    "type": "vb_spectrum",
    "version": 1,
    "axis": "z",
    "range": "gscale_16",
    "num_spectral_lines": 3200,
    "f_min": 10.0,
    "f_max": 6600.0,
    "f_min_envelope_filter": 0.0,
    "f_max_envelope_filter": 0.0,
    "spectrum_type": "acceleration",
    "averaging": 0,
    "time_to_transmit_min": 60,
    "send_condition": {
      "value_type": "always",
      "threshold": 0.0,
      "hold_off_hours": 0
    }
  }
}
```

Listing 42: Example of the JSON structure for scheduling a **spectrum** uplink.

6.9 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 6.8.1). 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.

6.9.1 vb_machine_fault_indicator task

machine_fault_indicator uplinks are scheduled using the `schedule_update_request` (as described in Section 4.3), with the *Task Settings* described below. Note that the index for the first bit is 112 as the previous fields are described in Table 4.2. The number of supported schedules of this type is 1.

Task Settings

```
{
  "type": "vb_machine_fault_indicator",
  "version": 1,
  "enable_spectrum_on_fault": <boolean>,
  "spectrum_hold_off_hours": <number>,
  "spectrum_time_to_transmit_min": <number>,
  "axis": <string>,
  "range": <string>,
  "f_min": <number>,
  "f_max": <number>,
  "f_min_envelope_filter": <number>,
  "f_max_envelope_filter": <number>,
  "machine_fault_selection": {
    "common_faults": <boolean>,
    "bearing_faults": <boolean>
  },
  "send_condition": {
    "value_type": <string>,
```

```

    "threshold": <number>
  }
}

```

Listing 43: JSON structure for **vb_machine_fault_indicator** task settings

Binary Encoding and Description

Table 6.33: Binary encoding for **vb_machine_fault_indicator** Task settings

Field ID	Type	Bit Index	Description
type	uint12	112..123	Task type, for vb_machine_fault_indicator : 2
version	uint4	124..127	Task version: 1.
enable_spectrum_on_fault	bool	128	If true, a spectrum uplink is sent when a machine fault is found. The type of the spectrum is the spectrum in which the fault was found. For common_faults , the spectrum type is velocity, for bearing_faults the spectrum type is envelope.
spectrum_hold_off_hours	uint10	129..138	Time to wait before sending a new spectrum message triggered by the detection of a fault in h.
spectrum_time_to_transmit_min	pfloat15	139..153	Specifies the duration of time used to transmit the fragmented spectrum uplink in min.
axis	uint2	154..155	Axis for which the vibration is measured. If "all" is selected, the schedule will result in one message for each of the three axes. For value mapping see the table below.
range	uint2	156..157	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. For value mapping see the table below.
f_min	pfloat15	158..172	Lower bound of the frequency range to be analyzed in Hz.
f_max	pfloat15	173..187	Upper bound of the frequency range to be analyzed in Hz.
f_min_envelope_filter	pfloat15	188..202	Lower bound of the frequency range to be analyzed in Hz.

Continues on next page

Table 6.33 Continued: Binary encoding for **vb_machine_fault_indicator** Task settings

Field ID	Type	Bit Index	Description
f_max_envelope_filter	pfloat15	203..217	Upper bound of the frequency range to be analyzed in Hz.
machine_fault_selection.common_faults	bool	218	If true , the sensor detects common faults.
machine_fault_selection.bearing_faults	bool	219	If true , the sensor detects bearing faults.
send_condition.value_type	uint4	220..223	Type of the <i>send condition</i> for the schedule. The device only transmits the scheduled uplink if it is satisfied. If set to "always", the scheduled messages are sent unconditionally. For value mapping see the table below.
send_condition.threshold	float16	224..239	Value that needs to be reached for the send condition. To be used in combination with <code>send_condition.value_type</code> .

Enum Value Mappings

Table 6.34: Value mapping of **vb_config_axis_v0**

Value mapping	Value
"x"	0
"y"	1
"z"	2
"all"	3

Table 6.35: Value mapping of **vb_range_v0**

Value mapping	Value
"gscale_2"	0
"gscale_4"	1
"gscale_8"	2
"gscale_16"	3

Table 6.36: Value mapping of **vb_send_condition_v0**

Value mapping	Value
"always"	0
"peak_acceleration_above"	1
"rms_acceleration_above"	2
"rms_velocity_above"	3
"temperature_above"	4
"temperature_below"	5

6.9.2 machine_fault_indicator message

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

Message Structure

```
{
  "message_name": "machine_fault_indicator",
  "message_version": 0,
  "timestamp": <string>,
  "<axis>_<fault_type>_category": <string>,
  "<axis>_<fault_type>_harmonic_frequency_0": <number>,
  "<axis>_<fault_type>_harmonic_frequency_1": <number>,
  "<axis>_<fault_type>_harmonic_frequency_2": <number>,
  "<axis>_<fault_type>_harmonic_frequency_3": <number>,
  "<axis>_<fault_type>_harmonic_frequency_4": <number>,
  "<axis>_<fault_type>_harmonic_frequency_5": <number>,
  "<axis>_<fault_type>_harmonic_frequency_6": <number>,
  "<axis>_<fault_type>_harmonic_frequency_7": <number>,
  "<axis>_<fault_type>_harmonic_frequency_8": <number>,
  "<axis>_<fault_type>_harmonic_frequency_9": <number>,
  "<axis>_<fault_type>_harmonic_amplitude_0": <number>,
  "<axis>_<fault_type>_harmonic_amplitude_1": <number>,
  "<axis>_<fault_type>_harmonic_amplitude_2": <number>,
  "<axis>_<fault_type>_harmonic_amplitude_3": <number>,
  "<axis>_<fault_type>_harmonic_amplitude_4": <number>,
  "<axis>_<fault_type>_harmonic_amplitude_5": <number>,
  "<axis>_<fault_type>_harmonic_amplitude_6": <number>,
  "<axis>_<fault_type>_harmonic_amplitude_7": <number>,
  "<axis>_<fault_type>_harmonic_amplitude_8": <number>,
  "<axis>_<fault_type>_harmonic_amplitude_9": <number>
}
```

Listing 44: JSON structure for **machine_fault_indicator** message

Binary Encoding and Description

Table 6.37: Binary encoding for **machine_fault_indicator** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 3.
message_version	uint4	4..7	Message version: 0.
timestamp	short_timestamp	8..23	UTC ISO timestamp of the measurement. Detailed information about the binary encoding is provided in Section 1.3.1.
axis	uint2	24..25	Axis for which the vibration is measured. For value mapping see the table below.
fault_type	uint2	26..27	Machine fault type. For value mapping see the table below.
category	uint6	28..33	Machine fault category. For value mapping see the table below.
frequency_first_harmonic	pfloat15	34..48	Ground frequency of the harmonics.

Continues on next page

Table 6.37 Continued: Binary encoding for **machine_fault_indicator** message sent on FPort 17

Field ID	Type	Bit Index	Description
amplitude_first_harmonic	pfloat15	49..63	Amplitude in the spectrum at the ground frequency of the harmonics.
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_frequency_0			Decoded frequency of the first harmonic in Hz.
harmonic_frequency_1..9			Harmonic frequency: $\text{frequency_first_harmonic} \times i$ where <i>i</i> is 1, 2, .., number of the harmonic.
harmonic_amplitude_0			Decoded amplitude of the first harmonic in g.
harmonic_amplitude_1..9			Decoded amplitude of the second harmonic in g. Calculated using: $\frac{\text{first}}{102} \times \text{nth_harmonic_amplitudes}[i]$ where <i>i</i> is the index (starting from 0) and <i>first</i> is the first harmonic amplitude.

Enum Value Mappings

Table 6.38: Value mapping of **vb_axis_v0**

Value mapping	Value
"x"	0
"y"	1
"z"	2

Table 6.39: Value mapping of **vb_fault_type_v0**

Value mapping	Value
"common_fault"	0
"bearing_fault"	1

Table 6.40: Value mapping of **vb_fault_category_v0**

Value mapping	Value
"none"	0
"one_x"	1
"two_x"	2
"n_x"	3
"bearing"	4

6.10 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. When a measurement triggers an alert, **false** to **true**, the measurement uplink is sent. This is true for all three types of measurements.

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.

Figure 6.2 shows the alert status change based on the selection, threshold, hysteresis values.

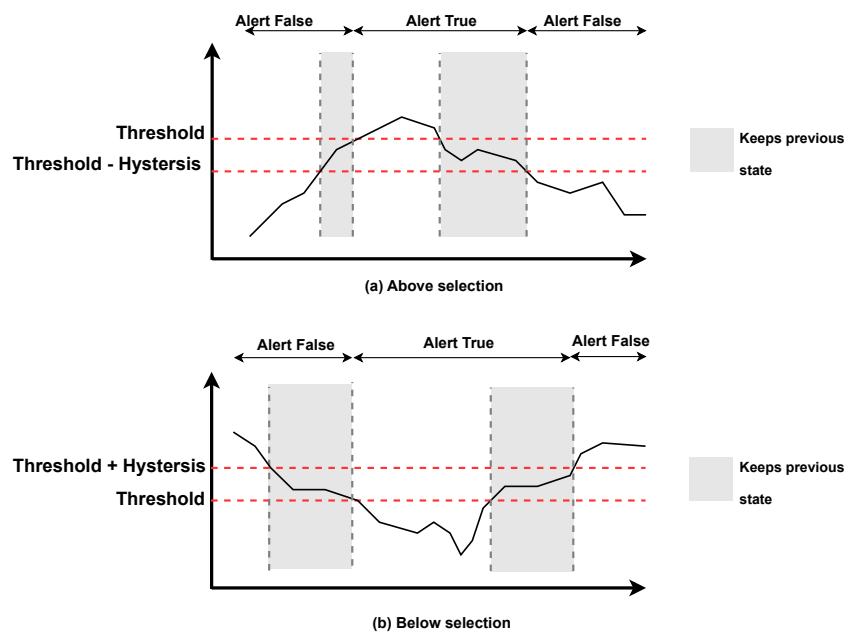


Figure 6.2: Alerts status change based on the selection, threshold, hysteresis values.

6.10.1 vb_alert configuration

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

Configuration Payload

```
{
  "type": "vb_alert",
  "version": 1,
  "enable_confirmed_alert": <boolean>,
  "enable_spectrum_on_alert": <boolean>,
  "spectrum_type": <string>,
  "time_to_transmit_min": <number>,
  "hold_off_hours": <number>,
  "num_spectral_lines": <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 45: JSON structure for **vb_alert** configuration payload

Binary Encoding and Description

Table 6.41: 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: 1.
enable_confirmed_alert	bool	56	If true, the alert uplinks are sent confirmed.
enable_spectrum_on_alert	bool	57	If true, the alert uplinks are sent confirmed.
spectrum_type	uint2	58..59	Type of frequency spectrum. For value mapping see the table below.
time_to_transmit_min	pfloat15	60..74	Specifies the duration of time used to transmit the fragmented uplink in min.

Continues on next page

Table 6.41 Continued: Binary encoding for **vb_alert** configuration payload

Field ID	Type	Bit Index	Description
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.
num_spectral_lines	uint14	85..98	The number of lines in the output spectrum. It is recommended to use the online configurator for the calculation of the allowed values for a required f_min and f_max.
alert_0.selection	uint5	99..103	Mode for sensor_alert_0. Similarly, alert_N.selection defines the mode for sensor_alert_N. For value mapping see the table below.
alert_0.threshold	float16	104..119	Threshold for sensor_alert_0. Similarly, alert_N.threshold defines the threshold for sensor_alert_N.
alert_0.hysteresis	pfloat15	120..134	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	uint5	135..139	
alert_1.threshold	float16	140..155	
alert_1.hysteresis	pfloat15	156..170	
alert_2.selection	uint5	171..175	
alert_2.threshold	float16	176..191	
alert_2.hysteresis	pfloat15	192..206	
alert_3.selection	uint5	207..211	
alert_3.threshold	float16	212..227	
alert_3.hysteresis	pfloat15	228..242	
alert_4.selection	uint5	243..247	
alert_4.threshold	float16	248..263	
alert_4.hysteresis	pfloat15	264..278	
alert_5.selection	uint5	279..283	
alert_5.threshold	float16	284..299	
alert_5.hysteresis	pfloat15	300..314	
alert_6.selection	uint5	315..319	
alert_6.threshold	float16	320..335	
alert_6.hysteresis	pfloat15	336..350	
alert_7.selection	uint5	351..355	
alert_7.threshold	float16	356..371	

Continues on next page

Table 6.41 Continued: Binary encoding for **vb_alert** configuration payload

Field ID	Type	Bit Index	Description
alert_7.hysteresis	pfloat15	372..386	
<i>Reserved for Future Use</i>	uint6	387..392	

Enum Value Mappings

Table 6.42: Value mapping of **vb_spectrum_type_v0**

Value mapping	Value
"acceleration"	0
"velocity"	1
"envelope"	2

Table 6.43: Value mapping of **vb_alert_selection_v1**

Value mapping	Value
"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
"x_rms_envelope_above"	12
"x_peak_envelope_above"	13
"y_rms_envelope_above"	14
"y_peak_envelope_above"	15
"z_rms_envelope_above"	16
"z_peak_envelope_above"	17

6.10.2 vb_spectrum_alert configuration

The spectrum alerts are configured using the **vb_spectrum_alert** configuration. The same limits for the resolution and frequency as discussed in Section 6.8.1 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 46: JSON structure for **vb_spectrum_alert** configuration payload

Binary Encoding and Description

Table 6.44: 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. For value mapping see the table below.
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.
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	

Continues on next page

Table 6.44 Continued: Binary encoding for **vb_spectrum_alert** configuration payload

Field ID	Type	Bit Index	Description
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	

Enum Value Mappings

Table 6.45: Value mapping of **vb_spectrum_alert_selection_v0**

Value mapping	Value
"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

6.10.3 alert message

Message Structure

```
{
  "message_name": "alert",
  "message_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 47: JSON structure for **alert** message

Binary Encoding and Description

Table 6.46: Binary encoding for **alert** message sent on FPort 17

Field ID	Type	Bit Index	Description
id	uint4	0..3	Message ID: 2.
message_version	uint4	4..7	Message version: 0.
timestamp	short_timestamp	8..23	UTC ISO timestamp of the measurement. Detailed information about the binary encoding is provided in Section 1.3.1.
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.

Continues on next page

Table 6.46 Continued: Binary encoding for **alert** message sent on FPort 17

Field ID	Type	Bit Index	Description
spectrum_alert_1	bool	33	
spectrum_alert_2	bool	34	
spectrum_alert_3	bool	35	
spectrum_alert_4	bool	36	
<i>Reserved for Future Use</i>	uint3	37..39	

6.11 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.

6.11.1 sensor_boot message

Message Structure

```
{
  "message_name": "sensor_boot",
  "message_version": 0,
  "sensor_reboot_reason": <string>
}
```

Listing 48: JSON structure for **sensor_boot** message

Binary Encoding and Description

Table 6.47: Binary encoding for **sensor_boot** message sent on FPort 17

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

Enum Value Mappings

Table 6.48: Value mapping of `reboot_reason_v0`

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

6.12 Configuration

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

6.12.1 vb_asset configuration

Configuration Payload

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

Listing 49: JSON structure for **vb_asset** configuration payload

Binary Encoding and Description

Table 6.49: 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. Must be larger than 0.
rpm_max	uint16	72..87	Upper bound of the RPM estimation in rpm. Must be larger than rpm_min.

Default

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

Listing 50: 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 51: 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 52: 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 + 0x7f800000; // 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 53: 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 54: 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 | 0x7c00; // 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 55: 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 56: 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 57: 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 58: 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 59: 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 60: Code example for calculating a CRC32 in JavaScript

Process Automation

The information in this document is provided for general informational purposes only. Specifications for products and services are subject to change without prior notice. IMI plc and its subsidiaries own all product brands mentioned herein.

IMI makes no warranties or representations about the accuracy or completeness of the content in this document and assumes no liability for any errors or omissions it may contain. We reserve the right to modify, enhance, or discontinue any product or service described herein without prior notification..

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

