TWTG the r&d company

Subject	LoRa WAN Communication Protocol TS	Rev	2.3.0
		By	ML/ZZ/TB/JC
Product	NEON Temperature Sensor	Date	9-4-2020

Version	Date	Author	Changes
2.0.0	9-4-2020	ZZ / ML	Split combined protocol in VS-QT, VS-MT and TS specific protocol documents. Separate downlink to protocol configuration and application configuration
2.1.0	15-5-2020	ZZ / ML	Updated event configuration. Corrected ranges and cleanup.
2.1.1	18-5-2020	ML	Corrected double entries
2.2.0	5-6-2020	ZZ/TB	1. Removed 500 kHz channel from lorawan_fsb_mask 2. Added disclaimer about lorawan_fsb_mask 3. Updated application configuration json
2.2.1	07-10-2020	TB	1. Corrected spelling mistakes in the Application configuration. 2. Added 'Binary (hex)' for default values 3. Added CRC example
2.3.0	14-11-2024	JC / ML	1. Corrected broken links 2. Renamed *unconfirmed_repeat* to *number_of_unconfirmed_messages* to be equal to other NEON products 3. Updated configs ranges and defaults

General

All variables are little endian.

Message schemes

Activation

Activation of the temperature sensor is initiated by the operator. After activation an ***Activation Message*** is sent. The full sequence is depicted below.

In case of an error during activation, the procedure is aborted. The error situations can be as followed (see numbers in diagram):

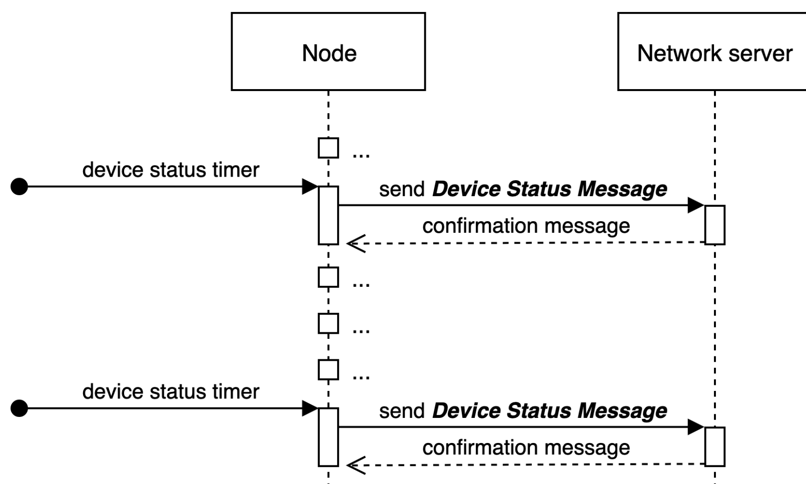
1. No LoRa confirmation message from network server is received: no LoRa coverage

Application Event

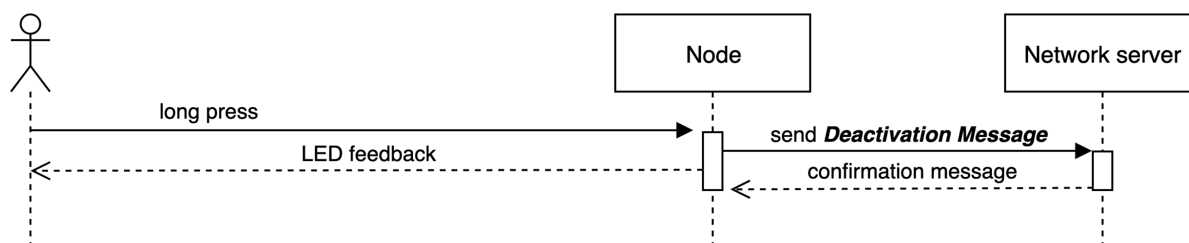
The temperature is measured periodically. When a condition change is detected an ***Application event message*** is sent.

Device Status

Periodically a ***Device status message*** is sent. Timing is independent from the temperature measure timer. The message contains information related to the device.



Deactivation



The operator can deactivate (in case of a temperature sensor) by a long press of the magnet key. This will result in a ***Deactivation Message***.

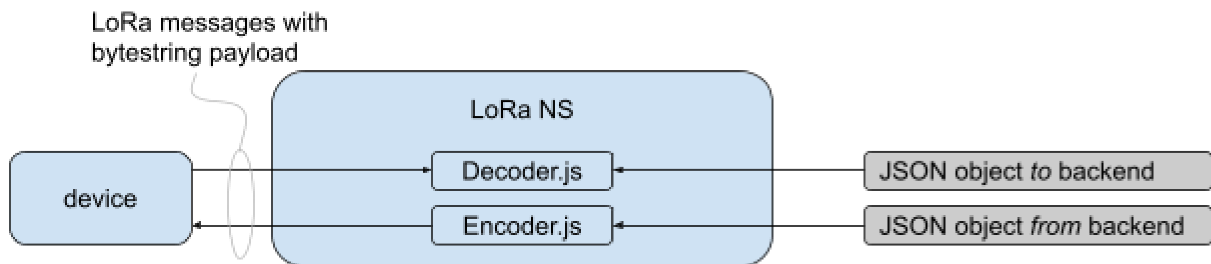
Message overview

ID	Name	Up/down	Purpose
0	Boot	up	Inform on device reboot
1	Activated	up	Indicates that a device is activated by the magnet key and operational.
2	Deactivated	up	Indicates that a device is deactivated by the magnet key.
3	Application event	up	Inform on the temperature. It is sent on event detection, or on a timer.
4	Device status	up	Informs on device health, battery info, counter for statistics etc. It is sent periodically.
5	Device configuration	down	Configure the device with radio setting

6	Application configuration	down	Configure the application specific settings

Decoding/Encoding

The messages are sent over LoRa in a binary bytestring. A LoRa network server can decode raw bytestrings coming from the LoRa devices into JSON objects. It can also encode JSON objects back into bytestrings that form the payload of downlink messages.



Below each message is described and how it is encoded in the bytestring. A new decoder / encoder will follow according to this document. If using this Decoder / Encoder the bytestring information below can be ignored.

Message header

The header is at the head of all messages. It indicates the protocol version and message type.

JSON structure

```
"header": {
      "message_type": <json string>,
      "protocol_version": <json number>
}
```

Binary encoding and description

Description	**Binary encoding**	**Byte index**
`message_type` Type of message, range: ● `boot` (0) ● `activated` (1) ● `deactivated` (2) ● `application_event` (3) ● `device_status` (4)	uint8 bits [3..0] message_id bits [7..4] protocol version number	0

● `device_configuration` (5) ● `application_configuration` (6) **`protocol_version`** Version of the protocol, range: ● For this protocol version the value is 2		

Uplink messages

Boot

The boot message is sent when the device is used for the first time or a reboot occurs. Reboots might occur intended in case of activating a newly received configuration. Or the device might reboot due to a system error as a matter to recovery (e.g. continues communication failures, unforeseen situations etc). The boot message contains information why it has (re)booted. Typically this information can be ignored and is only used for solving problems in the field. During normal operation and with sufficient network quality reboots seldomly occur, other than activating a newly received configuration.

JSON structure

```
"boot": {
      "device_type": <json string>,
      "version_hash": <json string>,
      "device_config_crc": <json string>,
      "application_config_crc": <json string>,
      "reset_flags": <json number>,
      "reboot_counter": <json number>,
      "reboot_info": <json string>,
      "last_device_state": <json number>,
      "bist": <json number>
}
```

Binary encoding and description

Description	Binary encoding	Byte index
Header as described in Message header.	uint8	0
`device_type` The device type as known in the NEON family. For this device, Temperature Sensor, the value is: ● "ts" (1)	uint8	1
`version_hash`	uint32	2..5

Version hash of the currently running firmware, represented as a hexadecimal string. Range: (hex string) `00000000 .. ffffffff`		
`device_config_crc` CRC of the currently loaded protocol configuration, represented as a hexadecimal string. This can be used to verify if the corrected configuration is loaded. Range: (hex string) `0000 .. ffff`	uint16	6..7
`application_config_crc` CRC of the currently loaded application configuration, represented as a hexadecimal string. This can be used to verify if the corrected configuration is loaded. Range (hex string): `0000 .. ffff`	uint16	8..9
`reset_flags` A bitmask field from the microcontroller that indicated the physical reason for the reset that caused the reboot. This can be used for analysis when a device is not working properly. bit 0 - Option Byte Loader bit 1 - Pin bit 2 - Power On bit 3 - Software. bit 4 - IWDG. bit 5 - WWDG bit 6 - Low Power	uint8	10
`reboot_counter` Counter of the number of reboots. Each time a reboot occurs the counter is increased. The counter is 8bit and will wrap after 255 to 0. This can be used for detecting abnormal rebooting behavior. Range: `0 .. 255`	uint8	11
`reboot_info` Informational string with: information on the reboot. Example values:	uint8[9] byte [0] reboot type	12..20

• `"swdog (ABCD)"` • `"assert (test:1234)"` • `"application (0xaabbccdd)"` • `"system (0xaabbccdd)"`	byte [1..8] reboot type specific payload	
`last_device_state` Last state the device was in before reboot. This can be used for analysis when a device is not working properly.	uint8	21
`bist` A bitmask with the result of the build in self test. At boot the device performs a self test in order to verify the working of essential components. This can be used for analysis when a device is not working properly. bit value: • 0: test failed • 1: test succeeded bit 0: reserved (always 1) bit 1: temperature sensor bit 2: battery measurement bit 3: reserved (always 0) bit 4: reserved (always 0) bit 5: lora module bit 6: provisioning bit 7: activated	uint8	22

Activated

On activation an activated message is sent. It is an indication that the device is operational.

JSON structure

The activation message does not have any payloadBinary encoding and description

Description	Binary encoding	Byte index
Header as described in Message header.	uint8	0

Deactivated

When the device is deactivated by the magnet key a deactivation message is sent. It is an indication that the device is not operational.

JSON structure

The deactivation message does not have any payload

Binary encoding and description

Description	**Binary encoding**	**Byte index**
Header as described in Message header.	uint8	0

Application event

The application event message contains information on a temperature. It is either triggered by an event detection or periodic (scheduled).

JSON structure

```
"application_event": {
	"trigger": <json string>,
	"temperature": {
		"min": <json number>,
		"max": <json number>,
		"avg": <json number>
	},
	“condition_0”: <json number>,
	“condition_1”: <json number>,
	“condition_2”: <json number>,
	“condition_3”: <json number>
}
```

Binary encoding and description

Description	**Binary encoding**	**Byte index**
Header as described in Message header.	uint8	0
`trigger` Source of trigger for the application event message. String withh the possible values:	uint8	1

• `"timer" (0)` ○ On the periodic timer • `"condition_0" (1)` • `"condition_1" (2)` • `"condition_2" (3)` • `"condition_3" (4)`		
`temperature` The temperature in units of `0.01 °C`. It is reported in the **`min`**, **`max`** and **`avg`** temperature since the last event message. Range: `-40.00 °C .. 80.00 °C`	int16[3] `0.01 °C` per LSB	2..7
`condition_n` The current state of each condition. • True condition is represented by 1 • False condition is represented by 0	uint8 bit0 = condition_0 bit1 = condition_1 bit2 = condition_2 bit3 = condition_3	8

Device status

This message contains information for maintenance, device health and other analysis. It is sent periodically.

JSON structure

```
"device_status": {
      "device_config_crc": <json string>,
      "application_config_crc": <json string>,
      "event_counter": <json number>,
      "battery_voltage": {
            "low": <json number>,
            "high": <json number>,
            "settle": <json number>
      },
      "temperature": {
            "min": <json number>,
            "max": <json number>,
            "avg": <json number>
      },
      "tx_counter": <json number>,
      "avg_rssi": <json number>,
      "avg_snr": <json number>,
}
```

Binary encoding and description

Description	Binary encoding	Byte index
Header as described in Message header.	uint8	0
`device_config_crc` CRC of the currently loaded protocol configuration, represented as a hexadecimal string. This can be used to verify if the corrected configuration is loaded. Range: (hex string) `0000 .. ffff`	uint16	1..2
`application_config_crc` CRC of the currently loaded application configuration, represented as a hexadecimal string. This can be used to verify if the corrected configuration is loaded. Range (hex string): `0000 .. ffff`	uint16	3..4
`event_counter` Counter for number of events. Every time an event message is sent this counter is increased. This can be used to detect missed event messages. Range: 0 .. 255 After 255 it will wrap to 1. Zero value is reserved for after boot only.	uint8	5
`battery_voltage` Voltage measurement in Volt, meant as input for battery charge estimation in the backend. Note that a single voltage level on itself is not suitable for battery charge determination because of the type of battery. To feed the model on the backend with accurate data the voltage is measured on different moments. During **`low`** load, **`high`** load and after a **`settle`** time after high load. Range: `0.000 V .. 4.000 V`	uint16[3] `0.001 V` per LSB	6..11
`temperature`	int8[3]	12..14

PCB temperature in units of `1 °C`. It is reported in the **`min`**, **`max`** and **`avg`** temperature since the last device status message. The number of temperature measurements can be configured. Range: `-40 °C .. 80 °C`	`1 °C` per LSB	
`tx_counter` Number of LoRa transmissions since the last device status message. It is reset after the device status message is sent. If the number of LoRa transmissions becomes bigger than 255 this field is set to 255.	uint8	15
`avg_rssi` The average RSSI of received messages since the last device status message. The value is in units of `1 dBm`. Range: `-255 dBm .. 0 dBm`	uint8 `-1dBm` per LSB	16
`avg_snr` The average SNR of received messages since the last device status message. Range: `0 dB .. 255 dB`	uint8	17

Downlink

Device configuration

In the device configuration the non application related behavior can be configured. Changing these parameters will have an effect on battery life and quality of service.

JSON structure

```
"device_config" = {
        "switch_mask": {
                "enable_confirmed_changed_message": <json bool>
        },
        "communication_max_retries": <json number>,
        "unconfirmed_repeat": <json number>,
        "periodic_message_random_delay_seconds": <json number>,
        "status_message_interval_seconds": <json number>,
        "status_message_confirmed_interval": <json number>,
```

```
        "lora_failure_holdoff_count": <json number>,
        "lora_system_recover_count": <json number>,
        "lorawan_fsb_mask": [
              <json number>,
              <json number>,
              <json number>,
              <json number>,
              <json number>
        ]
}
```

Binary encoding and description

Description	Default value	Binary encoding	Byte index
Header as described in Message header.		uint8	0
`switch_mask` Booleans to turn features on or off. **`enable_confirmed_changed_message`** `(0)` Enable confirmed *application event* message	 `false` Binary (hex): 0x00	uint8 bitmask bit 0 (other bits are unused)	1
`communication_max_retries` The maximum number of retries on failing confirmed messages. Range: `1 .. 10`	3 Binary (hex): 0x03	uint8	2
`number_of_unconfirmed_messages` The number of repeating on unconfirmed messages. Range: `1 .. 5`	1 Binary (hex): 0x01	uint8	3
`periodic_message_random_delay_seconds` To avoid clustering and collisions of uplink transmissions of multiple devices a random delay is added to periodic messages (device status message and timer triggered event message).	60 Binary (hex): 0x3C	uint8	4

Range: `0 .. 255 seconds`			
`status_message_interval_seconds` Interval in seconds at which periodic device status messages are sent. Range: `60 .. 604800 seconds` (= 7 days)	86400 (once per day) Binary (hex): 0xA0 0x05	uint16 60 seconds per LSB	5..6
`status_message_confirmed_interval` Confirm every *n* messages, the messages in between are sent unconfirmed. Default is 1, such that all periodic messages are confirmed. The number can be increased to require less downlinks (for reasons of gateway RF duty cycle or network server costs), but will degrade the quality of service. Range: `0 .. 100` DISCLAIMER: Although it is possible to make status messages always unconfirmed (by setting the value to 0) it is highly recommended to not to use this to avoid problems when join sessions are invalidated by the network server (due to network server problems).	1 Binary (hex): 0x01	uint8	7
`lora_failure_holdoff_count` In case of persistent network problems (not receiving acknowledgements on confirmed messages) the device tries to recover by a device reboot. This parameter configures the number of consecutive failed confirmed messages needed before it reboots. Range: `0 .. 5`	5 Binary (hex): 0x05	uint8	8
`lora_system_recover_count` The number of attempts the LoRa handler is trying to recover from a system failure (not responsive radio). Range: `0 .. 5`	1 Binary (hex): 0x01	uint8	9

DISCLAIMER: This value should not be changed for normal use.			
lorawan_fsb_mask Frequency sub-band (FSB) mask for upstream [Only applicable for US915 and AU915] DISCLAIMER: The EUT is only tested and certified for the FCC Hybrid transmission mode (1st 8 channels), therefore this channel configuration should not be changed for operational conditions Note that for older devices (year<2024) setting the 500 kHz channels is not allowed. If the config gets rejected by the device, set the last entry to zero: - old default: {"0x00FF", "0x0000", "0x0000", "0x0000", "0x0000"}	{"0x00FF", "0x0000", "0x0000", "0x0000", "0x0001"} Binary (hex): 0xFF 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00	uint16[5]	10..19
Device config CRC CRC of the configuration values above. It is calculated by the Encoder.	See Configuration CRC	uint16	20..21

Example

The device configuration message is constructed of the 1 byte header, the configuration payload and the 2 byte CRC. The default configuration is obtained by concatenating the header (0x25) with the default values of config payload and the calculated CRC.

Header (hex)	Config payload (hex)	CRC (hex)
25	00,03,01,3c,a0,05,01,05,01,ff,00,00,00,00,00,00,00,01,00	d2,81

The calculation of the CRC is explained in chapter Configuration CRC.

Application configuration

JSON structure

```
"application_config" = {
      "device_type": <json string>,
```

TWTG the r&d company

```
        "temperature_measurement_interval_seconds": <json number>,

        "periodic_event_message_interval": <json number>,
        "events": [
                {
                        "mode": <json string>,
                        "threshold_temperature": <json number>,
                        "measurements_window": <json number>
                },
                {
                        "mode": <json string>,
                        "threshold_temperature": <json number>,
                        "measurements_window": <json number>
                },
                {
                        "mode": <json string>,
                        "threshold_temperature": <json number>,
                        "measurements_window": <json number>
                },
                {
                        "mode": <json string>,
                        "threshold_temperature": <json number>,
                        "measurements_window": <json number>
                }
        ]
}
```

Binary encoding and description

Description	Default value	Binary encoding	Byte index
Header as described in Message header.		uint8	0
`device_type` The device type as known in the NEON family. For this device the value is: "ts" (1)	"ts" Binary (hex): 0x01	uint8 "ts" = 1	1
`temperature_measurement_interval_seconds` Interval in seconds, at which the temperature sensor is read. Changing this value has an effect on responsiveness and battery life.	900 (15 minutes) Binary (hex): 0x84 0x03	uint16	2..3

Range: `1 .. 28800 seconds (8 hours)`			
`periodic_event_message_interval` Interval in number of measurements at which the application event messages are periodically sent. The periodic counter is reset on every event message. The periodic counter is reset on every event message. Range: `0 .. 28800 measurements` Example setups of periodic event message interval: ● once per 4 hours (default):set this parameter to 16 (default) and measurement_interval_seconds to 900 seconds (15 minutes) ● once per 8 hours: set this parameter to 28800 (max) and measurement_interval_secondsis set to 1 second ● never: set this parameter to 0	16 Binary (hex): 0x10 0x00	uint16 1 measurement per LSB	4..5

`events`
Configuration of four independent events.
 	
See Binary encoding and description temperature event config
	[
	`{"mode":"off"},`
	`{"mode":"off"},`
	`{"mode":"off"},`
	`{"mode":"off"}`
	]
	
	Binary (hex):
	0x00
	0x00
	0x00
	0x00
	0x00
	0x00
	0x00
	0x00
	0x00
	0x00
	0x00
	0x00
	0x00
	0x00
		event[4]	6..21

	0x00 0x00		
`application_config_crc` CRC of the configuration values above. It is calculated by the Encoder.	See Configuration CRC	uint16	22..23

Binary encoding and description temperature event config

Description	Default value	Binary encoding	Relative byte index
`mode` The operational mode of this event: ● “off” ● “above” ● “below” ● “increasing” ● “decreasing” **off** mode: Disables condition detection. **above** mode: If the maximum temperature along the **measurement_window** is above the **temperature_threshold** then the condition is **true**. A transition of the condition from false to true or true to false will trigger an event message. **below** mode: If the minimum temperature along the **measurement_window** is below the **temperature_threshold** then the condition is **true**. A transition of the condition from false to true or true to false will trigger an event message. **increasing** mode: The condition is true when the current temperature is at least **temperature_threshold** higher than the minimum temperature in the **measurement_window**. A transition of the	“off” Binary (hex): 0x00	“off”=0 “above”=1 “below”=2 “increasing”=3 “decreasing”=4	0

condition from false to true or true to false will trigger an event message. **decreasing** mode: The condition is true when the current temperature is at least **temperature_threshold** lower than the maximum temperature in the **measurement_window**. A transition of the condition from false to true or true to false will trigger an event message.			
`threshold_temperature` The threshold temperature in units of `0.01 °C`. Range: `-120.00 °C .. 120.00 °C` Only *positive* in mode *increasing* and *decreasing*.	0 Binary (hex): 0x00 0x00	int16 `0.01 °C` per LSB	1..2
`measurements_window` The maximum number of measurements to observe delta temperature to trigger an event. For mode "off" the default value is 0. For all other modes: Range: `1 .. 144 measurements`	1 Binary (hex): 0x00	uint8	3

Example

The application configuration message is constructed of the 1 byte header, the configuration payload and the 2 byte CRC. The default configuration is obtained by concatenating the header (0x26) with the default values of config payload and the calculated CRC, which results in the following byte string:

Header (hex)	Config payload (hex)	CRC (hex)
26	01, 84, 03, 10, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00, 00	2e,c8

The calculation of the CRC is explained in chapter Configuration CRC.

Configuration CRC

The config messages contain a CRC to identify different configuration payloads. This CRC is reported back to the backend via the boot and status messages. The messages are constructed as followed:

Part	Size (bytes)
Header	1
Config payload	n
CRC	2

The payload length n is fixed per config type. The CRC is calculated over the config payload only, so without the message header.

CRC calculation

The two CRC bytes are the inverse of the lower two bytes of a CRC32 calculation. For the CRC32 calculation the CRC-32/ISO-HDLC is used. This is the default in many protocols and the implementation is widely available in many libraries. The two CRC bytes in the protocol can be extracted as followed (pseudo code):

```
crc = default_crc32([config_payload])
inverse_crc = crc ^ 0xFFFFFFFF
crc_byte0 = inverse_crc & 0xFF
crc_byte1 = (inverse_crc >> 8) & 0xFF
```

Python

CRC function in Python

```
import binascii
def calc_crc(bs):
    crc = binascii.crc32(bs)
    return (crc & 0xFFFF) ^ 0xFFFF
```

Example using the Python CRC function

```
bs = binascii.a2b_hex("0103023ca00501020100000000000000000000")
crc = calc_crc(bs)
print(hex(crc))
```

This results in: `0x55d4`, which are encoded as the two CRC bytes (hex): `d4,55`
Representing the CRC as little endian uint16 bytestring (pack the uint16 CRC and print as hex):

```
from struct import pack
print(binascii.b2a_hex(pack("<H", crc)))
```

Results in `b'd455'`

TW
TG
L the r&d company

Javascript

CRC function in Javascript

```
// calc_crc inspired by https://github.com/SheetJS/js-crc32
function calc_crc(buf) {
  function signed_crc_table() {
    var c = 0, table = new Array(256);

    for (var n = 0; n != 256; ++n) {
      c = n;
      c = ((c & 1) ? (-306674912 ^ (c >>> 1)) : (c >>> 1));
      c = ((c & 1) ? (-306674912 ^ (c >>> 1)) : (c >>> 1));
      c = ((c & 1) ? (-306674912 ^ (c >>> 1)) : (c >>> 1));
      c = ((c & 1) ? (-306674912 ^ (c >>> 1)) : (c >>> 1));
      c = ((c & 1) ? (-306674912 ^ (c >>> 1)) : (c >>> 1));
      c = ((c & 1) ? (-306674912 ^ (c >>> 1)) : (c >>> 1));
      c = ((c & 1) ? (-306674912 ^ (c >>> 1)) : (c >>> 1));
      c = ((c & 1) ? (-306674912 ^ (c >>> 1)) : (c >>> 1));
      table[n] = c;
    }

    return typeof Int32Array !== 'undefined' ? new Int32Array(table) :
            table;
  }
  var T = signed_crc_table();

  var C = -1, L = buf.length - 3;
  var i = 0;
  while (i < buf.length) C = (C >>> 8) ^ T[(C ^ buf[i++]) & 0xFF];
  return C & 0xFFFF;
}
```

Example using the Javascript CRC function

The `calc_crc` function can be used as followed (this can be tested using online services like https://jsfiddle.net/):

```
var configPayload = [0x01, 0x03, 0x02, 0x3c, 0xa0, 0x05, 0x01, 0x02, 0x01, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00];
var crc = calc_crc(configPayload)
console.log("Calculated CRC: 0x" + crc.toString(16))
```

This results in `"Calculated CRC: 0x55d4"`, which are encoded as the two CRC bytes (hex): `d4,55`