Dzero Gateway 4X

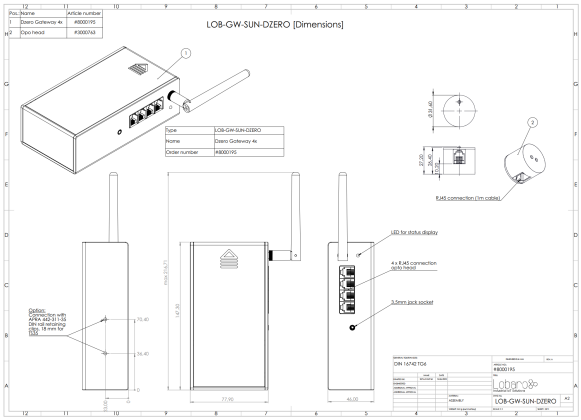
General Information

The Dzero Gateway 4X is a device that can be used to readout modern utility meters with standardized infrared "INFO" interface.

The meter outputs over its infrared "INFO" interface a serial protocol conforming to the Smart Meter Language Protocol 1.04 (SML) or IEC 62056-21 protocol. This interface is intended to be used by end-users and **not** for billing purposes of the electricity supplier.

The read information normally contains the current consumption values of the meter and gets interpreted and forwarded by the Dzero Gateway 4X via a LoRaWAN or Nb-IoT to web based applications interested in further processing this data.

Dimensions



Download_Dimensions

Datasheet

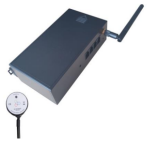
Lobaro
Industrial IoT Solutions

DZERO GATEWAY 4X

LOB-GW-SUN-DZERO

Version: 15.06.2023

OVERVIEW



The *Dzero Gateway 4x* is a battery powered device and used for remote reading of electricity meters with optical D0 / "IrNFO" interface.

FEATURES

- The device supports the SML protocol and IEC 62056-21 and has a protocol autodetection
- Uplink data via NB-IoT mobile radio (LTE bands: 3, 8, 20)) or LoRaWAN v1.0.2 (EU-868)) or wMBus
- Communication with meters takes place by means of the round Lobaro add-on module ("opto head")
- Up to four electricity meters can be read with only one Dzero Gateway 4x
- Automatic detection of the amount of connected opto heads
- Opto head is attached to meter with an integrated magnet
- One opto head is included, three more can be connected
- Connection to a TH 35 top-hat rail is given via a top-hat rail holder
- Interface D0 is intended to be used by end-users **and not for billing** purposes of the electricity supplier

APPLICATION

The *Dzero Gateway 4X* is a battery powered device. It is used for automated LoRaWAN, wMbus, Nb-IoT or LTE-M remote read out of novel "smart metering" devices via the standardized optical customer interface ("IrNFO") of the electricity meter.

TECHNICAL SPECIFICATIONS

General

Type	LOB-GW-SUN-DZERO
Purchase name	Dzero Gateway 4X
Input Voltage	3.3V to 5V, 3.6V (standard)
Item number	8000195

Cellular NB-IoT / LTE Cat. M1 Modem

LTE SIP	Nordic nRF9160
LTE bands	Standard: B8, B20, B3 (791 MHz - 969 MHz, 1710 MHz – 1785 MHz) Other Bands on request
SIM card	4FF (Nano-SIM)
Data Upload	Secure CoAP over DTLS (via Lobaro IoT Platform)

LoRaWAN

Type	Semtech SX1261
LoRaWAN Protocol Version	Class A LoRaWAN 1.0.2 (EU-868)
TX power	≤ +14 dBm
Activation method	Over the-air activation (OTAA) Activation by personalization (ABP)
Typical RF range	≤ 2km
Ideal RF range	≤ 10km (**free line of sight**)
Encryption	AES128

Antenna

Ext. type	SMA female connector ≤ 2 dBi
Internal type	Optional: Multiband PCB monopole (NB-IoT)

Battery

Approved type	SAFT LSH-20
Voltage	3.6V
Other types	On request
Chemistry	Li-SOCl2
Capacity	≤ 13 Ah
Max. Current	≤ 1.8 A
Weight	≤ 120 g
Connector	JST-XH 2-Pin

Additional I/O

Local configuration port	PC based initial configuration via USB adapter
LED	RGB Led to signal different operating modes
4 x RJ45	Connection of opto heads with 1m cable
Jack socket, 3.5mm	Connection for folding converter

DZERO GATEWAY 4X

LOB-GW-SUN-DZERO

Lobaro
Industrial IoT Solutions

TECHNICAL SPECIFICATIONS

Power Supply

Supply Voltage Connection	JST XH 2P Connector
Supply Voltage Range	3.3V – 5V Volt (DC)
Max. Power Consumption	≤ 2.5W
Typical Power Consumption	≤ 1.5W

Housing

Dimensions	147,3mm (l) x 77,9 mm (b) x 46mm (h)
Material	Polystyrol
Weight	342 g (incl. battery)
Protection class	IP4X
Color	black (RAL 9016)
Opto head	31,6(d) x 27,2(h)

Environment

Operating temp.	-20 °C to 55 °C
Storage temp.	0 °C to 30 °C

Certificates and approvals

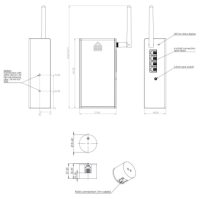
- EN 300 220-1 (RF)
- EN 300 220-2 (RF)
- EN 301 908-1 (RF)
- EN 301 908-13 (RF)
- EN 301 489-3 & EN 301 489-52 (EMC)
- EN 62368-1 (Electrical Safety)
- EN 62311:2008 (Electrical Safety)
- IEC 63000:2018 (RoHS)

RADIO FREQUENCY

RX: receive mode, TX: transmit mode

Direction	Protocol	Frequency	Bandwidth	Modulation	Duty Cycle	Transmit Power (EIRP)
RX	wMBus(S,T,C1)	868.95 MHz	+/- 0.002 MHz	FSK		
RX	wMBus(R 2)	868.3 MHz	+/- 0.250 MHz	FSK		
RX + TX	NB-IoT Band 8	880-915 MHz (up) 925-960 MHz (down)	20 MHz	NB-IoT		+ 23 dBm
RX + TX	NB-IoT Band 20	832-862 MHz (up) 791-821 MHz (down)	20 MHz	NB-IoT		+ 23 dBm
RX + TX	NB-IoT Band 3	1710-1785 MHz (up) 1805-1880Mhz (down)	75 MHz	NB-IoT		+ 23 dBm

DIMENSIONAL DRAWINGS



WARRANTY AND GUARANTEE

Warranty and guarantee claims can only be asserted if the device has been used as intended and the technical specifications and applicable technical rules have been observed.

Download_Datasheet

Warning

Older meters with "infrared pulse" output are **not** compatible to the Dzero Gateway 4X. Please check our list of **compatible meters** to make sure it is equipped with the correct interface.

Hint

Consider using the latest firmware on your hardware

- **See available firmware downloads**

Compatible utility meters

Any meter that adheres to the standard can be read. The following list contains meters that we successfully tested.

Electricity meter	Manufacturer
DTZ541-ZEBA	Holley
LK13 series	Logarex
OpenWay® 3.HZ	iTron
SGM-C4 series	efr
SGM-D series	efr
eHZ-K series	EMH
mMe4.0 series	EMH
ED300 series	EMH
eBZD series	EMH
E320	Landis+Gyr
MT681	ISKRA

Quickstart

1. Connect to the device with the Lobaro Tool using the Lobaro Config Adapter
2. Under Configuration click "Reload Config" and change the fields "ReadCron" and "ObisCode" for all connected heads as needed as well as choosing the desired Network Connection by setting the "WAN" followed by clicking on "Write to Device". The RJ12 port near the 3.5 mm jack is port 1, the RJ12 port near the Antenna is port 4.
3. If you selected lorawan as WAN: Register the device in your LoRaWAN network
4. Place the EDL21 opto head on the "Info" interface
5. Restart the device by pressing the reset button

Configuration

The configuration is done using Lobaro Maintenance Tool and the Lobaro USB PC adapter or remote via LoRaWAN downlink (see LoRaWAN Downlink Config) or LTE.

Network (general)

Name#LoRaWAN-RemoteConfiguration	Description	Default Value	Value Description & Examples

`WAN`	Radio technology used for connection to backend	`lte`	• `lte`: use either cellular NB-IoT or LTE-M • `nbiot`: use cellular NB-IoT • `ltem`: use cellular LTE-M • `lorawan`: use LoRaWAN with OTAA • `lorawan-abp`: use LoRaWAN with ABP
`Host`			

LoRaWAN

The connection to the LoRaWAN network is defined by multiple configuration parameters. This need to be set according to your LoRaWAN network and the way your device is supposed to be attached to it, or the device will not be able to send any data.

For a detailed introduction into how this values need to be configured, please refer to the chapter LoRaWAN configuration in our LoRaWAN background article.

Name#LoRaWAN-RemoteConfiguration	Description	Default Value	Value Description & Examples
`LostReboot`	Days without downlink before reboot (triggers downlinks)	5	days, `0`=don't reboot
`DevEUI`	DevEUI used to identify the Device		e.g. `0123456789abcdef`
`AppKey`	Key used for OTAA (v1.0 and v1.1)		
`NwkKey`	Key used for OTAA (v1.1 only)		
`JoinEUI`	Used for OTAA (called AppEUI in v1.0)		e.g. `0123456789abcdef`
`SF`	Spreading Factor	12	

LTE

Name#LoRaWAN-RemoteConfiguration	Description	Default Value	Value Description & Examples
`Host`	Hostname / IP of the Lobaro Platform API	coaps://platform.lobaro.com, coap://platform.lobaro.com	
`Operator`	Mobile Operator Code (optional)	`26201`	`26201` (=Deutsche Telekom), for other operators, see above. Empty = Auto detect (longer connecting time)
`Band`	NB-IoT Band		"8", "20", "8,20", Empty = Auto detect (longer connecting time)
`APN`	Mobile operator APN (optional)	*	1nce: iot.1nce.net Vodafone Easy Connect: lpwa.vodafone.com (l = littel L)
`PIN`	SIM PIN (optional)		Empty or 4 digits (e.g. `1234`)
`DNS`	DNS Server	9.9.9.9,1.1.1.1	
`UseNbiot`		`true`	
`UseLtem`		`true`	

wMBus (experimental)

Name#LoRaWAN-RemoteConfiguration	Description	Default Value	Value Description & Examples
`EncryptionMode`	7	7	5 or 7

Operation

Configuration values defining the behaviour of the device.

Name#LoRaWAN-RemoteConfiguration	Description	Default Value	Value Description & Examples
`ObisCode1`	Comma separated list of ObisCodes to select a subset of the available information on port 1	1-0:1.8.0*255	
`ObisCode2`	Comma separated list of ObisCodes to select a subset of the available information on port 2	1-0:2.8.0*255	

`ObisCode3`	Comma separated list of ObisCodes to select a subset of the available information on port 3	1-0:1.8.0*255	
`ObisCode4`	Comma separated list of ObisCodes to select a subset of the available information on port 4	1-0:2.8.0*255	
`PayloadFormat`	Format used for data upload (include timestamps or not)	int	`1`=no timestamp, 2=include timestamp
`ReadCron`	Cron expression defining when to read and upload	0 0/15 `****`	`0 0/15 * * * *` for every 15 minutes
`VerboseLogging`	Enables additional Debug output	false	true = enabled, false = disabled

See also our Introduction to Cron expressions and our Introduction to Obis Codes.

LED blinking patterns

The following pattery are explained in the order in which they appear after initial power on / reset of the device.

`red/green/blue`	500 ms each	initial pattern after reset

Payload (LoRaWAN)

Payload

Payload Format 1 (Port 3, with exponent)

This Format is used, when the configuration parameter `PayloadFormat` is set to `1`. Please refer to the documentation of Format 2. The only differences are the port and that there is no Unix Timestamp included in the header.

Payload Format 2 (Port 4, with exponent and timestamp)

This Format is used, when the configuration parameter `PayloadFormat` is set to 2 .

Each uploaded LoRaWAN-Message with data is prefixed with a 5 byte Timestamp, indicating when the values where received from the attached meter. This allows for a more precise timing information then using the time of reception, as the upload can be delayed quite heavily due to our random delay feature and potentially due to duty cycle restrictions. The timestamp also makes it easy to reassociate values from multiple uplinks to a single reading, when multiple uplinks must be used to upload all values. If a readout is spilt over multiple uplinks (because of LoRaWAN's length restrictions), every uplink from that reading will have the same timestamp (which is the time of received the values from the meter).

The Timestamp is sent as a UNIX-Timestamp encoded as a bigendian signed 40-bit number.

"Unit" is not yet implemented an will always be zero. Please be aware that SML relates to SI units whereas IEC can output units like kWh. The output of the device will not be automatically converted to SI units.

The payload consists of a header and multiple entries, one entry per OBIS code given in the configuration. Each entry follows the following structure:

Header

Head Number	**Unix Timestamp**
1 byte	5 bytes

Entry

OBISCode (hex)	**length of value (n)**	**value**	**exponent**	**unit**
6 bytes	1 byte	n bytes, LSB first	1 byte (signed)	1 byte

Example packet: 03 00 63 d1 47 80 01 00 01 08 00 ff 03 14 e3 31 ff 00 01 00 01 08 00 FE 08 FF 02 00 00 00 00 00 00 02 00

Header:

	Head Number	Unix Timestamp
Bytes	03	00 63 d1 47 80
Description	Opto Head connected to Port 3	Wednesday, 25. January 2023 15:15:12

Entry 1:

	OBISCode (hex)	length of value (n)	value	exponent	unit
Bytes	0100010800FF	03	14 e3 31 (=3269396)	ff	00
Description	1-0:1.8.0*255	3	326939.6 (3269396*10^-1)	-1	0

Entry 2:

	OBISCode (hex)	length of value (n)	value	exponent	unit
Bytes	0100010800FE	08	FF02000000000000 (=767)	02	00
Description	1-0:1.8.0*254	8	76700 (767*10^2)	2	0

Multiple messages

The Bridge puts as many values in a single data message as possible (respecting the current Spreading Factor). When it cannot fit all values in a single message, it will send multiple data messages until all values are uploaded. It will never split a single value. Since every value is prefixed with the Obis code, the parser can easily assign values to Obis codes.

Status Message (Port 64)

Example Payload: 45 44 4c 00 00 01 00 00 00 0e 31 00 dc 00

name	len	type	description	in example
Firmware Identifier	3	`String`	3 Charcter FW Identifier	45 44 4c EDL
Firmware Version	3	`uint8[3]`	Version: <major>.<minor>.<patch>	00 00 01 0.0.1
Status	1	`uint8`	RFU - always 0	00
Reboot reason	1	`uint8`	RFU - always 0	00
Final words	1	`uint8`	RFU - always 0	00
vBat	2	`int16`	battery Voltage in mV	0e 31 3633 mV 3.633 V
Temperature	2	`int16`	Internal temperature from controller in 1/10 °C	00 dc 220 22.0 °C
Custom Status	1	`uint8, bitfield`	Indicates global errors, i.e. if bit 5 is set no opto head could be found	00 no error, everything is fine

Payload (LTE)

Handled by the Platform

Payload (wMBus)

See Standard. APPKey will be used as key for encryption.

Reference decoder

This is a decoder written in JavaScript that can be used to parse the device's messages.

```
function readName(bytes, i) {
    return bytes.slice(i, i + 6);
}

function readValue(len, bytes, i) {
    if (len <= 0) {
        return [];
    }
    return bytes.slice(i, i + len);
}

function toHexString(byteArray) {
    var s = '';
    byteArray.forEach(function (byte) {
        s += ('0' + (byte & 0xFF).toString(16)).slice(-2);
    });
    return s;
}

function parse_sint16(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    var t = bytes[0] << 8 | bytes[1] << 0;
    if( (t & 1<<15) > 0){ // temp is negative (16bit 2's complement)
        t = ((~t)& 0xffff)+1; // invert 16bits & add 1 => now positive value
        t=t*-1;
    }
    return t;
}

function parse_uint16(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    var t = bytes[0] << 8 | bytes[1] << 0;
    return t;
}


function signed(val, bits) {
    if ((val & 1 << (bits - 1)) > 0) { // value is negative (16bit 2's complement)
        var mask = Math.pow(2, bits) - 1;
        val = (~val & mask) + 1; // invert all bits & add 1 => now positive value
        val = val * -1;
    }
    return val;
}
function uint40_BE(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    return bytes[0] << 32 |
        bytes[1] << 24 | bytes[2] << 16 | bytes[3] << 8 | bytes[4] << 0;
}
function uint16_BE(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    return bytes[0] << 8 | bytes[1] << 0;
}
function int40_BE(bytes, idx) {return signed(uint40_BE(bytes, idx), 40);}
function int16_BE(bytes, idx) {return signed(uint16_BE(bytes, idx), 16);}
function int8(bytes, idx) {return signed(bytes[idx || 0], 8);}

function toNumber(bytes) {
    var res = 0;

    for (var i = bytes.length-1; i >= 0 ; i--) {
        res *= 256;
        res += bytes[i];
    }

    return res;
```

```
}

function readVersion(bytes) {
    if (bytes.length<3) {
        return null;
    }
    return "v" + bytes[0] + "." + bytes[1] + "." + bytes[2];
}

function decodeStatus(bytes) {
    var decoded = {
        "version":readVersion(bytes),
        "flags": bytes[3],
        "vBat": uint16_BE(bytes, 4) / 1000,
        "temp": int16_BE(bytes, 6) / 10,
    };

    if (Device) {
        Device.setProperty("version", decoded.version);
        Device.setProperty("voltage", decoded.vBat);
        Device.setProperty("temperature", decoded.temp);
        Device.setProperty("status_flags", decoded.flags);
    }

    return decoded;
}

function decodeSmlValuesV1(bytes) {
    var decoded = {
        values: [],
    };

    if (bytes.length === 1) {
        // No Data! Read error?
        return decoded;
    }

    var pos = 0;
    while (pos < bytes.length) {
        var name = readName(bytes, pos);
        pos += 6;
        var len = bytes[pos];
        pos += 1;
        var value = readValue(len, bytes, pos);
        pos += len;

        var val = {
            nameHex: toHexString(name),
            len: len,
            value: toNumber(value),
            valueHex: toHexString(value)
        };

        decoded.values.push(val);
    }

    return decoded;
}

function decodeSmlValuesV2(bytes) {
    var decoded = {
        values: [],
    };

    if (bytes.length === 1) {
        // No Data! Read error?
        return decoded;
    }

    var pos = 0;
    var headNo = bytes[0];
```

```
    decoded.headNo = headNo;
    pos += 1;
    while (pos < bytes.length) {
        var name = readName(bytes, pos);
        pos += 6;
        var len = bytes[pos];
        pos += 1;
        var value = readValue(len, bytes, pos);
        pos += len;
        if (len > 0) {
            var exponent = int8(bytes, pos);
            pos += 1;
        }
        if (len > 0) {
            var unit = int8(bytes, pos);
            pos += 1;
        }
        var val;
        if (len > 0) {
            val = {
                obiscode: name[0] + "-" + name[1] +":" + name[2] + "." + name[3] + "." +name[4] + "*" +name
[5],
                //len: len,
                value: toNumber(value) * Math.pow(10, exponent),
                unit: toNumber(unit),
                //valueHex: toHexString(value),
            }
        } else {
            val = {
                obiscode: name[0] + "-" + name[1] +":" + name[2] + "." + name[3] + "." +name[4] + "*" +name
[5],
                //len: len,
                value: toNumber(value),
                unit: toNumber(unit),
                //valueHex: toHexString(value),
            }
        }

        decoded.values.push(val);
    }
    return decoded;
}

function decodeSmlValuesV3(bytes) {
      var decoded = {
        values: [],
    };

    if (bytes.length === 1) {
        // No Data! Read error?
        return decoded;
    }

    var pos = 0;
    var headNo = bytes[0];
    decoded.headNo = headNo;
    pos += 1;

    decoded.time = int40_BE(bytes, 1) * 1000;
    pos+=5;

    while (pos < bytes.length) {
        var name = readName(bytes, pos);
        pos += 6;
        var len = bytes[pos];
        pos += 1;
        var value = readValue(len, bytes, pos);
        pos += len;
        if (len > 0) {
            var exponent = int8(bytes, pos);
            pos += 1;
```

```
        }
        if (len > 0) {
            var unit = int8(bytes, pos);
            pos += 1;
        }
        var val;
        if (len > 0) {
            val = {
                obiscode: name[0] + "-" + name[1] +":" + name[2] + "." + name[3] + "." +name[4] + "*" +name
[5],
                //nameHex: toHexString(name),
                //len: len,
                value: toNumber(value) * Math.pow(10, exponent),
                //valueHex: toHexString(value),
                unit: toNumber(unit),
            }
        }
        else {
            val = {
                obiscode: name[0] + "-" + name[1] +":" + name[2] + "." + name[3] + "." +name[4] + "*" +name
[5],
                //len: len,
                value: toNumber(value),
                //valueHex: toHexString(value),
                unit: toNumber(unit),
            }
        }

        decoded.values.push(val);
    }
    return decoded;
}

function decode_status_code(code) {
  switch (code) {
    case 0:
      return "OK";
    default:
      return "UNKNOWN";
  }
}

function decode_reboot_reason(code) {
  // STM reboot code from our HAL:
  switch (code) {
    case 1:
      return "LOW_POWER_RESET";
    case 2:
      return "WINDOW_WATCHDOG_RESET";
    case 3:
      return "INDEPENDENT_WATCHDOG_RESET";
    case 4:
      return "SOFTWARE_RESET";
    case 5:
      return "POWER_ON_RESET";
    case 6:
      return "EXTERNAL_RESET_PIN_RESET";
    case 7:
      return "OBL_RESET";
    default:
      return "UNKNOWN";
  }
}

function DecoderPort64(bytes) {
  // legacy format, firmware 4.x
  // Decode an uplink message from a buffer
  // (array) of bytes to an object of fields.
  var firmware = String.fromCharCode.apply(null, bytes.slice(0, 3));
  var version = readVersion(bytes, 3);
  var status_code = bytes[6];
```

```
  var status_text = decode_status_code(status_code);
  var reboot_code = bytes[7];
  var reboot_reason = decode_reboot_reason(reboot_code);
  var final_code = bytes[8];
  var vcc = (parse_uint16(bytes, 9) / 1000) || 0.0;
  var temp = (parse_sint16(bytes, 11) / 10) || -0x8000;
  var error_state = bytes[13];

  Device.setProperty("firmware", firmware);
  Device.setProperty("version", version);
  Device.setProperty("status_code", status_code);
  Device.setProperty("status_text", status_text);
  Device.setProperty("reboot_code", reboot_code);
  Device.setProperty("reboot_reason", reboot_reason);
  Device.setProperty("final_code", final_code);
  Device.setProperty("error_state", error_state);
  Device.setProperty("temperature", temp);
  Device.setProperty("voltage", vcc);

  return {
    "firmware": firmware,
    "version": version,
    "status_code": status_code,
    "status_text": status_text,
    "reboot_code": reboot_code,
    "reboot_reason": reboot_reason,
    "final_code": final_code,
    "temperature": temp,
    "voltage": vcc,
    "error_state": error_state
  };
}

function Decoder(bytes, port) {
    // Decode an uplink message from a buffer
    // (array) of bytes to an object of fields.
    switch (port) {
        case 1:
            return decodeStatus(bytes);
        case 2:
            return decodeSmlValuesV1(bytes);
        case 3:
            return decodeSmlValuesV2(bytes);
        case 4:
            return decodeSmlValuesV3(bytes);
        case 64:
            return DecoderPort64(bytes);
    }
}

function NB_ParseDeviceQuery(input) {
    for (var key in input.d) {
        var v = input.d[key];
        switch (key) {
            case "temperature":
                v = v / 10.0;
                break;
            case "vbat":
                v = v / 1000.0;
                //NB_SetBatteryStatus(v)
                break;
        }
        Device.setProperty("device." + key, v);
    }
    return null;
}

function NB_ParseConfigQuery(input) {
    for (var key in input.d) {
        Device.setConfig(key, input.d[key]);
    }
```

```
    return null;
}

function ParseV2(input) {
  var decoded = {
        values: [],
    };

  decoded.headNo = input.d.head;
  decoded.time = new Date(input.d.timestamp*1000).toISOString();


  //return toNumber(parseBase64(input.d.batch[0].value)) * Math.pow(10, input.d.batch[0].scaler);
  //decoded.payload = input.d;
  var val;

  if ((input.d.batch) && (input.d.batch.length > 0) ) {
    for( i = 0; i<input.d.batch.length; i++){
      val = {
          obiscode: input.d.batch[i].obiscode,
          value: input.d.batch[i].value * Math.pow(10, input.d.batch[i].scaler),
          //valueHex: toHexString(input.d.batch[i].value),
          unit: toNumber(input.d.batch[i].unit),
      }
      decoded.values.push(val);
    }
  }
  return decoded;
}

function NB_ParseStatusQuery(input) {
  for (var key in input.d) {
    var v = input.d[key];
      switch (key) {
          case "temperature":
              v = v / 10.0;
              break;
          case "vbat":
              v = v / 1000.0;
              //NB_SetBatteryStatus(v)
              continue;
      }
      Device.setProperty("device." + key, v);
  }
  return null;
}


function ParseNBiot(input) {
  var query = input.q || "data";
  var res = null;
  switch (query) {
        case "device":
            res = NB_ParseDeviceQuery(input);
            break;
        case "config":
            res = NB_ParseConfigQuery(input);
            break;
        case "data":
            res = ParseV2(input);
            break;
        case "status":
            res = NB_ParseStatusQuery(input);
            break;

        default:
            throw new Error("Unknown message type: '" + query + "'");
  }

  if (res != null) {
    res.addr = input.i;
```

```
    res.fCnt = input.n;
  }
  return res;
}

// Wrapper for Lobaro Platform
function Parse(input) {
    // Decode an incoming message to an object of fields.
    var b = bytes(atob(input.data));

    // NB-IoT
    if (input.d) {
      return ParseNBiot(input)
    }

    // LoRaWAN
    return Decoder(b, input.fPort);
}
```