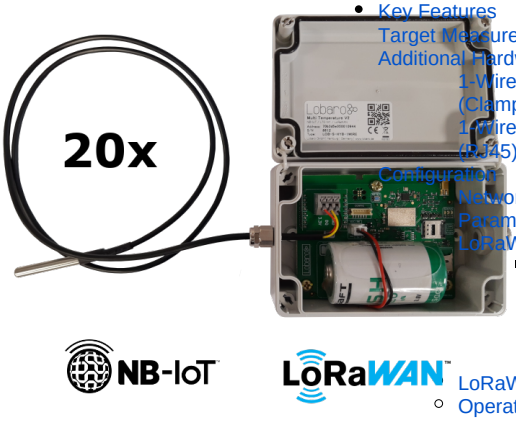
Multi Temperature V2 (LoRaWAN, NB-IoT)

Typ: LOB-S-HYB-1WIRE



Order Numbers/Variants:

8000186 - Multi Temperatur Box v2 (LoRaWAN, NB-IoT, RJ45)	8000170 - Multi Temperatur V2 (LoRaWAN, NB-IoT, 1m Temp.-Fühler)

Compatible Replacement for old articles:

- 8000001 - LoRaWAN Multi Temperatur Box (Kabeldurchführung, AA-Batterien)
- 8000072 - LoRaWAN Multi Temperatur Box (XH Batterieanschluss, IP66 Gehäuse, DAE, 1m Temp.-Fühler)
- 8000094 - NB-IoT Multi Temperatur Box (XH Batterieanschluss, IP66 Gehäuse, DAE, 1m Temp.-Fühler)

Table of Contents

Quickstart

1. Connect to the device with the Lobaro Tool using the Lobaro Config Adapter
2. Under Configuration click "Reload Config" and change the configuration according to your needs
3. Register the device in your LoRaWAN network
4. Connect D-cell Battery via XH connector

Key Features

- Supports up to 20 DS18x20 1-Wire sensors
- 8-Port easy installation Hub available
- IP67 outdoor housing with pressure compensating element
- Comes with one sensor attached
- Sensor output order can be configured
- Testmode for easy sensor identification
- Big 19Ah size "D" battery for 10 years+ possible battery lifetime (not included)
- LoRaWAN 1.0.x and 1.1 network servers supported
- LoRaWAN Class A
- LoRaWAN time synchronisation
- Variant with external power-supply and or external antenna available on request
- Quick closing screws with cover retainer on housing

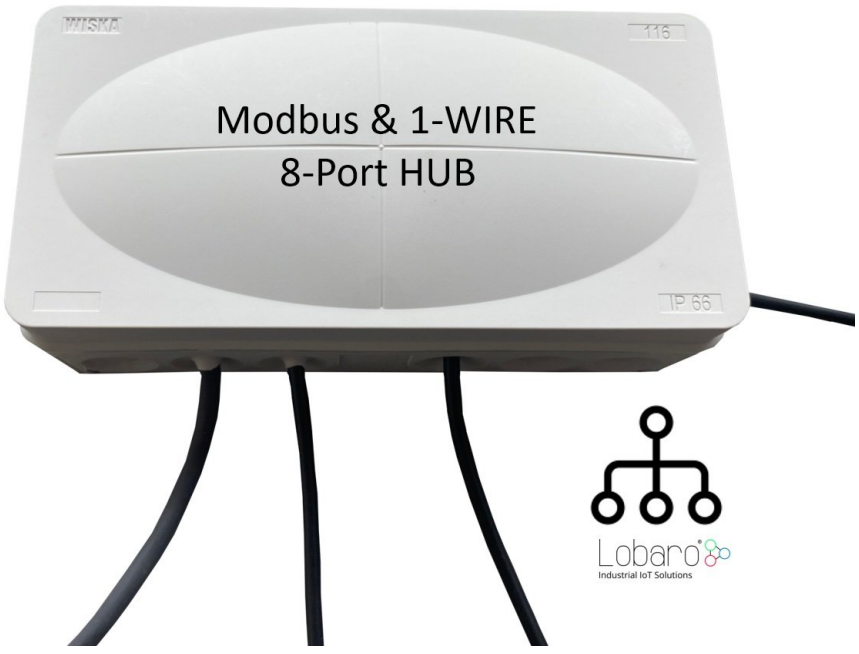
Target Measurement / Purpose

The Multi Temperature Sensor Box is used to read out up to up to 25 DS18x20 1-Wire temperature sensors. The temperature from all sensors is read regualarly and sent via LoRaWAN uplink automatically splitting the data into multiple uplinks if the payload gets too big for a single LoRaWAN message. Sensors can be attached in series or in star configuration.

Additional Hardware

1-Wire 8-Port Hub (Clamp System)

For easy connection of up to 7 DS18x20 1-Wire sensors consider using our Modbus & 1-Wire 8-Port Hub (`Order number: 8000130`). Multiple Hubs can be connected to each other to achieve the limit of 25 sensors per Box.



Order Numbers/Variants:

- 8000130 HUB IP66 (8x: 1-Wire or Modbus)

1-Wire 8-Port Hub (RJ45) System

	
Order Numbers/Variants:	
	1-wire DS18B20 Temp. sensor RJ45 (3m cable)
• 3000793 1-Wire Hub 8x RJ45	• 3000784 - DS18B20 Temp.-Sensor RJ45
	Pinout (Front-side)
	blocked URL
	Pin 2: 3V3 provided by Lobaro Box **Pin 4: 1-Wire Data** **Pin 5: 1-Wire GND**

Configuration

The configuration is done using Lobaro Maintenance Tool and the Lobaro USB PC adapter.

Networking Parameters (NB-IoT / LoRaWAN)

Name	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`	Hostname / IP of the Lobaro Platform API *Not used for LoRaWAN uplink*	`94.130.20.37`	94.130.20.37 = platform.lobaro.com ⚠ DNS is not supported yet
`Port`	Port number of the Lobaro Platform API *Not used for LoRaWAN uplink*	`5683`	

NB-IoT Parameters (WAN = "lte", "nbiot", "ltem")

The LTE functionality is enabled if the `WAN` parameter is set to `lte`, `nbiot`, or `ltem`. Using this mode requires an appropriate SIM-Card to be inserted.

Name	Description	Default Value	Value Description & Examples
`Operator`	Mobile Operator Code (optional)	`26201`	`26201` (=Deutsche Telekom), for other operators, see above.
`Band`	NB-IoT Band	8	"8", "20", "8,20", Empty = Auto detect (longer connecting time)
`APN`	Mobile operator APN (optional)	`iot.1nce.net`	1nce: iot.1nce.net Vodafone Easy Connect: lpwa.vodafone.com (l = littel L)
`PIN`	SIM PIN (optional)		Empty or 4 digits (e.g. `1234`)

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.

> ⚠ Downlink configuration via LoRaWAN is not supported yet.

Name	Description	Type	Default Value	Values
`DevEUI`	DevEUI used to identify the Device	`byte[8]`		e.g. `0123456789abcdef`
`JoinEUI`	Used for OTAA (called AppEUI in v1.0)	`byte[8]`		e.g. `0123456789abcdef`
`AppKey`	Key used for OTAA (v1.0 and v1.1)	`byte[16]`		
`SF`	Initial / maximum Spreading Factor	`int`	12	`7 - 12`
`OpMode`	Operation Mode	`string`	A	A= Class A, C= Class C
`LostReboot`	Days without downlink before reboot	`int`	5	days, `0`=don't reboot

Operation

Without configuration the sensors will be transmitted ordered by the 48 Bit id, ignoring the Sensorfamily prefix and the Checksum.

name	description	example value
`TestMode`	Run device in Testing Mode to help identify sensors. Must be `false` for normal operations.	`true` or `false`
`MeasureCron`	Cron expression† defining when to read out sensors	`0 0/15 * * * *`(every 15 minutes)
`SendInternalTemp`	Toggle output of internal sensor. Will always be sent first.	`true` or `false`
`SendSensorId`	Include Sensor ID in upload. Changes Payload format and Port.	`true` or `false`
`SensorIdOrder`	Semicolon separated list of 48 Bit IDs in hex (up to 25)	`22ffffff0000;44ffffff0000; 11ffffff0000`

† See also our Introduction to Cron expressions.

Temperature and Error values

Temperature is transmitted in 10th of degrees Centigrade (d°C), to avoid having to deal with floating point numbers. Error conditions for individual sensors are transmitted as temperatures below -300°C.

temperature	hex value	error condition
-899.0°C	`0xdce2`	Sensor not found (for sensors set in `SensorIdOrder`).
-997.0°C	`0xd90e`	Communication timed out.
-998.0°C	`0xd904`	Temperature read out error.
-999.0°C	`0xd8fa`	No temperature value.

Sensor IDs

A complete sensor ID consists of 8 bytes:

```
byte    0: family code
bytes 1-6: serial number
byte    7: crc cecksum
```

Only the serial number is used by the device. Family code and checksum are omitted for uploads and in configuration.

Internal Sensor

There is a temperature sensor included in the board of the device. If `SendInternalTemp` is set to `true`, that sensor will be included in uploads. It will always be the first sensor in the list.

Sensor order

The sensors will be ordered by their IDs (without the leading byte indicating the sensor type). You can fix the upload order by listing sensors in `SensorIdOrder`. Sensors included here will always be included in upload messages, whether the sensor is found or not. If the sensor cannot be found, a temperature value of -899.0°C (`=0xdce2`) is sent.

If the internal sensor is included (by `SendInternalTemp=true`), the internal sensor will still be sent first, before the first sensor listed in `SensorIdOrder`.

Any sensors that are found and are neither internal nor listed in `SensorIdOrder` will be appended after the last sensor from `SensorIdOrder`, ordered by their IDs.

Payload

Example payloads for each port:

Status Message (Port 1)

Only for Firmware < v0.2.0

Payload: `00040001070ce3`

Decoded:

```
      {
  "temp": 26.3,
  "vBat": 3.299,
  "version": "v0.4.0"
}
```

Data Message (Port 2)

Payload with temperature measurements when Sensor-ID is included.

Structure:

name	len	type	description
success	1	`uint8`	0 = Read error, 1 = Success
sensor id	6	`uint8[6]`	6-Byte 1-Wire Sensor Id
temperature	2	`int16 BE`	Temperature in 1/10 °C
...			... sensor id and temperature fields repeat ...

When the total length of data exceeds 50 bytes, the message will be split and uploaded in multiple packets using this format.

Example Payload: `01551e46920d0200da96b446920c0200d7dafc46920d0200d5202e4692050200dc`

Decoded:

```
      {
  "sensors": [
    {
      "id": "551e46920d02",
      "temp": 21.8
    },
    {
      "id": "96b446920c02",
      "temp": 21.5
    },
    {
      "id": "dafc46920d02",
      "temp": 21.3
    },
    {
      "id": "202e46920502",
      "temp": 22
    }
  ],
  "success": true
}
```

Data Message (Port 3)

Payload with temperature measurements when Sensor-ID is omitted.

Before v0.4.0 this format is sent on Port 2.

Structure:

name	len	type	description
success	1	`uint8`	0 = Read error, 1 = Success
temperature	2	`int16 BE`	Temperature in 1/10 °C
...			... temperature field repeats ...

When the total length of data exceeds 50 bytes, the message will be split and uploaded in multiple packets using this format.

Example Payload: `0100f500f500f800f500f300f8`

Decoded

```
      {
   "sensors": [
     24.5,
     24.5,
     24.8,
     24.5,
     24.3,
     24.8
   ],
   "success": true
}
```

Error Message (Port 4)

Error Message containing a single 0 Byte to indicate that not a single temperature sensor could be found and therefore no data exists.

Status Message (Port 64)

Since v0.2.1

Example Payload: `545031 000202 000000 0e6c 010e`

name	len	type	description
Firmware Identifier	3	`String`	3 Charcter FW Identifier `TP1`
Firmware Version	3	`uint8[3]`	Version: <major>.<minor>.<patch>
Status	1	`uint8`	RFU - always 0
Reboot reason	1	`uint8`	RFU - always 0
Final words	1	`uint8`	RFU - always 0
vBat	2	int16	battery Voltage in mV
Temperature	2	int16	Internal temperature from controller in 1/10 °C

Parser

Lobaro Platform / TheThingsNetwork (TTN) / ChirpStack

```
        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 parse_hex(bytes, idx, end) {
    var chars = "0123456789abcdef";
    bytes = bytes.slice(idx || 0, end || null);
    var s = "";
    for (var i=0; i<bytes.length; i++) {
        var byte = bytes[i];
        s += chars.charAt(byte>>4);
        s += chars.charAt(byte & 0xf);
    }
    return s;
}

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

function DecoderPort1(bytes) {
    return {
        "version":readVersion(bytes),
        "temp": parse_sint16(bytes, 3) / 10,
        "vBat": parse_uint16(bytes, 5) / 1000,
```

```
    };
}

function DecoderPort2(bytes) {
    // Decode an uplink message from a buffer
    // (array) of bytes to an object of fields.
    var sensors = [];
    var success = false;

    var pos = 0;
    if ( bytes.length ) {
        pos+=1;
        success = bytes[0] !== 0;
    }
    var left = bytes.length - pos;
    while (left>=8) {
        var sensor = {
            //'id_': bytes.slice(pos, pos+6),
            'id': parse_hex(bytes, pos, pos+6),
            'temp': parse_sint16(bytes, pos+6) / 10.0
        };
        sensors.push(sensor);
        pos += 8;
        left = bytes.length - pos;
    }
    // if (port === 1) decoded.led = bytes[0];

    var decoded = {};
    decoded['success'] = success;
    decoded['sensors'] = sensors;
    return decoded;
}

function DecoderPort3(bytes) {
    // Decode an uplink message from a buffer
    // (array) of bytes to an object of fields.
    var sensors = [];
    var success = false;

    var pos = 0;
    if ( bytes.length ) {
        pos+=1;
        success = bytes[0] !== 0;
    }
    var left = bytes.length - pos;
    var nr=0;
    while (left>=2) {
      var temp = parse_sint16(bytes, pos) / 10.0
        sensors.push(temp);
        pos += 2;
        left = bytes.length - pos;
    }
    // if (port === 1) decoded.led = bytes[0];

    var decoded = {};
    decoded['success'] = success;
    decoded['sensors'] = sensors;
    return decoded;
}

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

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 app_data = bytes.slice(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("app_data", app_data);
  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,
    "app_data": app_data
  };
}

function Decoder(bytes, port) {
    switch (port) {
        case 1:
            return DecoderPort1(bytes);
```

```
        case 2:
            return DecoderPort2(bytes);
        case 3:
            return DecoderPort3(bytes);
        case 4:
            return "error: No sensors found";
        case 64:
            return DecoderPort64(bytes);
    }
    return {
        "error": "Invalid port",
        "port": port
    };
}

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

    if (input.i && input.d) {
      // NB-IoT
      var decoded = {};
      decoded = input.d;
      decoded.address = input.i;
      decoded.fCnt = input.n;

      var query = input.q || "data";

      switch (query) {
        case "config":
            return NB_ParseConfigQuery(input);
        default:
      }
      return decoded;
    }

    var decoded = Decoder(b, input.fPort);

    decoded.rssi = input.rssi;
    decoded.fCnt = input.fCnt;

    return decoded;
}

// Wrapper for Loraserver / ChirpStack
function Decode(fPort, bytes) {
    return Decoder(bytes, fPort);
}
```

CE Declaration of Conformity

CE Declaration of Conformity (pdf).