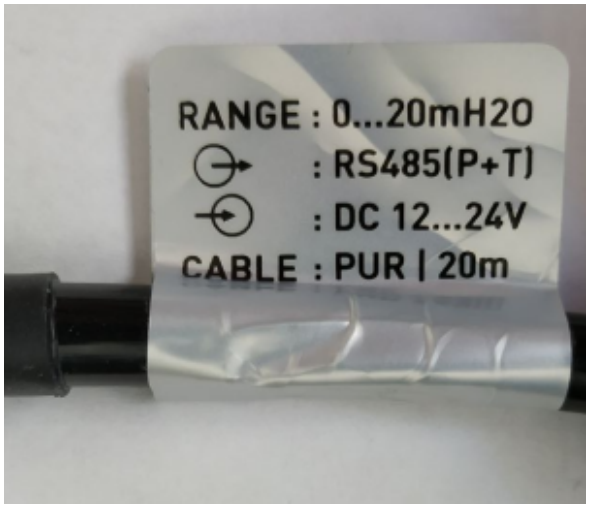
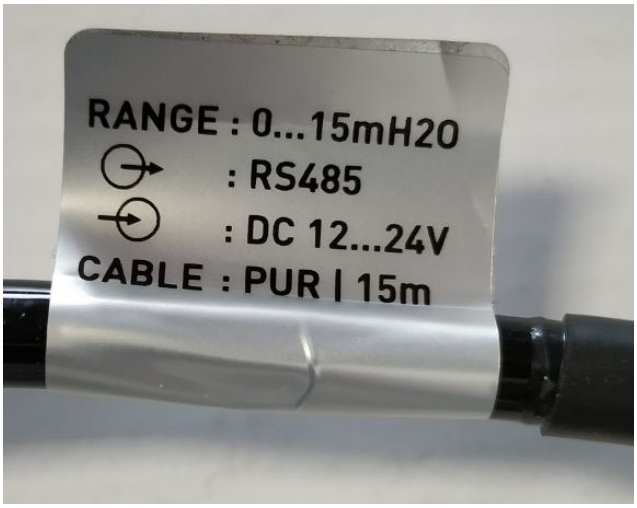
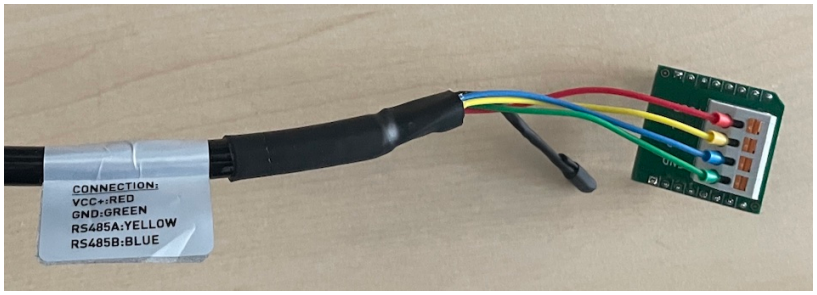
Pressure Sensor Application

Lobaro Pressure or Lobaro Pressure and Temperature Sensor

Lobaro Platform Device Type: LOBARO-PEGELSONDE-HYBRID

Select the correct sensor by checking the cable flag. The Pressure and Temperature sensor got a "P+T" marking.

Lobaro Pressure and Temperature Sensor	**Lobaro Pressure Sensor**	**KELLER S 26X**
Lobaro article: #3000625	Lobaro article: ~~#3000588~~ (discontinued)	Lobaro articl 0701
		

• 0 - 15 mH2O (other lengths on request) • Accuracy: +/- 0.25% FSO (Fully Scale Output) • Drift: +/- 0.3% FSO per year • For water level sensing applications	• 0 - 15 mH2O • Accuracy: 0.5% +/- FSO (Fully Scale Output) • Drift: +/- 0.3% FSO per year • For water level sensing applications	• Typ: "P / 16mH 00001-(• 0 - 16 n • Accurac 0.25% I (Fully S Output) • White=(Black = • Blue=R Yellow= • Water l sensing applicat tank lev sensing • More In eller We
Red = VCC, Green = GND, RS485A = Yellow, RS485B = Blue		
CABLE CONNECTIONS & COLORS 		

Lobaro Pressure and Temperature Sensor

To read out the pressure & temperature probe, the Hybrid Modbus Gateway must be configured as follows:

Parameter	Value	Comment
`WAN`	`lorawan`	For LoRaWAN OTAA usage.
`PlFmt`	**5**	Sets the payload to a short format.
`MbCmd`	0 0/15 `***`**:R,9600,8N1: 010300160002,010300260002,fa0400050001**	• `010300160002` - Read register 22-23 (pressure) • `010300260002` - Read register 38-39 (temperature) • `FA0400050001` - Read internal register 5 (vBat) The CRON Expressions can be adjusted to set time of sensor reading.
`PowerOnDelay`	**3000**	*Battery variant only*. Sets time (in ms) between activating sensor power and reading value (time for sensor to be ready).

Command (since FW v0.10.2)

Starting from v0.10.2 we support a command for LoRaWAN only!

Parameter	Value	Comment
`Cmd`	`0 * * * * *:pressure(port=20)`	Special command to upload in same format but calculating a more stable pressure value

Modbus Register Mapping

The probe is a Modbus slave with the following registers:

Modbus Command: `<Slave Address (1)><Function (1)><Address (2)><Length (2)>`

- Length = Register count -> 1 Register = 2 Bytes
- Function
    - 0x03 = Read Holding Register
    - 0x04 = Read Input Register
    - 0x06 = Write Holding Register

Address	Function	Bytes	Data Scope	Description
0x0000	0x03, 0x06	2	1-255	Slave Address
0x0001	0x03, 0x06	2	• 0-1200 • 1-2400 • 2-4800 • 3-9600 • 4-19200 • 5-38400 • 6-57600 • 7-115200	Baud rate
0x0002	0x03, 0x06	2	• 0 -No • 1- Odd • 2 - Even	• No check • Odd check • Even check
0x0003	0x03	2	• 0- kpa • 1- Mpa • 2- ma • 3- % • 4- Inh2o • 5- FtH2o • 6- mmH2o • 7- mmHg • 8- psi • 9- bar • 10- mBar • 11- g/cm² • 12- kg/cm² • 13- Pa • 14- Torr • 15- Atm • 16- Null • 17- M • 18- cm • 19- mm • 20-InHg • 21- mHg • 22- MH20 • 23- °C	Pressure unit
0x0004	0x03	2	• 0 -#### • 1- ###.# • 2 - ##.## • 3 - #.### • 4 - #.####	Decimal point stands for 0-4 digits decimal points
0x0005	0x03	2	0-30	Filtering Coefficient
0x0016-0017	0x03	4	4 byte float	Measurement Pressure output value
0x0018-0019	0x03, 0x06	4	4 byte float	Master variable of Pressure offset
0x001A-001B	0x03	4	4 byte float	Range minimum of Pressure Transmitter
0x001C-001D	0x03	4	4 byte float	Range maximum of Pressure Transmitter
0x001E-001F	0x03	4	4 byte float	Range minimum of Pressure Sensor
0x0020-0021	0x03	4	4 byte float	Range maximum of Pressure Sensor
0x0026-0027	0x03	4	4 byte float	Measurement Temperature output value
0x00A6-00A7	0x06	4	4 byte float	Zero Clearing value of Pressure Transmitter

0xFFFF	0x06	2	• 0 - save to user area	Save data to user area
0xFFFC	0x06	2	• 0 - factory reset	Restore to factory status (user settings and calibration data)

Data Uplink (Port 20)

```
Bytes | 0 .    | 1 . 2 . 3 . 4 . | 5 . 6 . 7 . 8 . | 9 . 10 . |
------+--------+-----------------+-----------------+----------+
Field | Header | Pressure        | Temperature     | Voltage  |
```

All values are encoded **big-endian**

Field	Type	Value
Header	`uint8`	`0x00` on success, `0x80` if an error occurred
Pressure	`float32`	Pressure in `mH2O`, `ffffffff` on error.
Temperature	`float32`	Temperature in `°C`, `ffffffff` on error.
Voltage	`uint16`	Voltage in mV, `ffff` on error

010300160002

Example

```
# Example of a successful measurement
'003d94ce4541b7a5120e2a'
  '00'       -> Successful readout
  '3d94ce45' -> 0.073 mH2O
  '41b7a512' -> 22.96 °C
  '0e2a'     -> 3626 mV / 3.626 V

# Example
'80ffffffffffffff'
  '80'       -> An error occurred.
  'ffffffff' -> Pressure could not be read.
  'ffffffff' -> Temperature could not be read.
  'ffff'     -> Voltage could not be read.
```

Reference Parser

see below

Lobaro Pressure Sensor

Reading from the Lobaro Pressure Sensor using the Hybrid Gateway can be done by setting the following parameters in the configuration:

Parameter	Value	Comment
`WAN`	`lorawan`	For LoRaWAN OTAA usage.
`PlFmt`	**5**	Sets the payload to a short format.
`MbCmd`	0 0/15 * * * * **:R,9600,8N1:010300040001, fa0400040001,fa0400050001**	• 010300040001 - Read register 4 (pressure) • fa0400040001 - Read internal register 4 (temperature in box) • fa0400050001 - Read internal register 5 (vBat) The CRON Expressions can be adjusted to set time of sensor reading.

`PowerOnDelay`	3000	*Battery variant only*. Sets time (in ms) between activating sensor power and reading value (time for sensor to be ready).

Modbus Register Mapping

The probe is a Modbus slave with the following registers:

Modbus Command: `<Slave Address (1)><Function (1)><Address (2)><Length (2)>`

- Length = Register count -> 1 Register = 2 Bytes
- Function
    - 0x03 = Read Holding Register
    - 0x06 = Write Holding Register

Address	Function	Bytes	Data Scope	Description
0x0000	0x03, 0x06	2	1-255	Slave Address
0x0001	0x03, 0x06	2	• 0-1200 • 1-2400 • 2-4800 • 3-9600 • 4-19200 • 5-38400 • 6-57600 • 7-115200	Baud rate
0x0003	0x03	2	• 0 -#### • 1- ###.# • 2 - ##.## • 3 - #.###	Decimal point stands for 0-3 digits decimal points
0x0002	0x03	2	• 0- Mpa/ • 1- Kpa • 2- Pa • 3- Bar • 4- Mbar • 5- kg/cm2 • 6- psi • 7- mh2o • 8- mmh2o	Pressure unit
0x0004	0x03	2	-32768-32767	Measurement output value
0x0005	0x03	2	-32768-32767	Zero point of transmitter range
0x0006	0x03	2	-32768-32767	Full point of transmitter range
0x000c	0x03, 0x06	2	-32768-32767	Zero point offset value, generally factory sets as 0
0x000F	0x06	2	• 0 - save to user area	
0x0010	0x06	2	• 1 - factory reset	

Data Uplink LoRaWAN (Port 20)

```
Bytes | 0 .    | 1 . 2 .  | 3 . 4 .     | 5 . 6 . |
------+--------+----------+-------------+---------|
Field | Header | Pressure | Temperature | Voltage |
```

All values are encoded **big-endian**

Field	Type	Value
Header	`uint8`	`0x00` on success, `0x80` if an error occurred

Pressure	`int16BE`	Pressure in mmH2O
Temperature	`int16BE`	Temperature in °C inside Bridge
Voltage	`uint16BE`	Voltage in mV, `ffff` on error

Example

```
# Example of a successful measurement
'000211001a0e2a'
  '00'           -> Successful readout
  '0211' -> 529  -> 0.529 mH2O
  '001a' -> 26   -> 26°C (inside Box)
  '0e2a' -> 3626 -> 3626 mV / 3.626 V
```

Keller PR26X

Configuration

Connected pressure sensor probe from Keller Druckmesstechnik PR26X series.

Parameter	Value	Comment
`WAN`	`lorawan`	For LoRaWAN OTAA usage.
`PlFmt`	**5**	Sets the payload to a short format.
`MbCmd`	0 0 * * * *:R,9600,8N1: 010300020002,010300080002,FA0400050001	Reads four Registers: 2 + 3 (Float, Pressure in Bar) and 8 + 9 (Float, Probe Temperature) + Device battery voltage
`PowerOnDelay`	1500	*Battery variant only*. Sets time (in ms) between activating sensor power and reading value (time for sensor to be ready).

Example Modbus response

Hex to float converter: https://gregstoll.com/~gregstoll/floattohex/

Pressure (0x3f75f07b):

P1	1	request	`0x01 0x03 0x00 0x02 0x00 0x02 0x65 0xCB`	0.960701bar
		response	`0x01 0x03 0x04 0x3F 0x75 0xF0 0x7B 0xE3 0xDE`	

Temperature (0x41b5c079):

TOB1	1	request	`0x01 0x03 0x00 0x08 0x00 0x02 0x45 0xC9`	22.719°C
		response	`0x01 0x03 0x04 0x41 0xB5 0xC0 0x79 0x6E 0x0B`	

Data Uplink (Port 20)

```
Bytes | 0 .    | 1 . 2 . 3 . 4 . | 5 . 6 . 7 . 8 . | 9 . 10 . |
------+--------+-----------------+-----------------+----------+
Field | Header | Pressure        | Temperature     | Voltage  |
```

All values are encoded **big-endian**

Field	Type	Value
Header	`uint8`	`0x00` on success, `0x80` if an error occurred
Pressure	`float32`	Pressure in `Bar`, `ffffffff` on error.
Temperature	`float32`	Temperature in `°C`, `ffffffff` on error.

Voltage	`uint16`	Voltage in mV, `ffff` on error

Keller PR46X

Configuration

Connected pressure sensor probe from Keller Druckmesstechnik PR46X series.

Parameter	Value	Comment
`WAN`	`lorawan`	For LoRaWAN OTAA usage.
`PlFmt`	**5**	Sets the payload to a short format.
`MbCmd`	0 0 `****`:R,9600,8N1: 010300020002,010300060002,FA0400050001	Reads four Registers: 2 + 3 (Float, Pressure in Bar) and 6 + 7 (Float, Probe Temperature) + Device battery voltage
`PowerOnDelay`	1500	*Battery variant only*. Sets time (in ms) between activating sensor power and reading value (time for sensor to be ready).

Data Uplink (Port 20)

```
Bytes | 0 .    | 1 . 2 . 3 . 4 . | 5 . 6 . 7 . 8 . | 9 . 10 . |
------+--------+-----------------+-----------------+----------+
Field | Header | Pressure        | Temperature     | Voltage  |
```

All values are encoded **big-endian**

Field	Type	Value
Header	`uint8`	`0x00` on success, `0x80` if an error occurred
Pressure	`float32`	Pressure in `Bar`, `ffffffff` on error.
Temperature	`float32`	Temperature in `°C`, `ffffffff` on error.
Voltage	`uint16`	Voltage in mV, `ffff` on error

LoRaWAN JavaScript Reference Parser (All probe variants)

LORAWAN ONLY WORKS WITH DEFAULT CONFIG FOR MBCMD!

Pressure Probe Parser

```
/**
 * Parser for Lobaro Pressure Probe via LoRaWAN (hybrid gateway).
 * Usable for Pressure Probe as or with Presure+Temperature Probe.
 * Works with TTN, ChirpStack, or the Lobaro Platform.
 */
function signed(val, bits) {
    // max positive value possible for signed int with bits:
    var mx = Math.pow(2, bits-1);
    if (val < mx) {
        // is positive value, just return
        return val;
    } else {
        // is negative value, convert to neg:
        return val - (2 * mx);
    }
}

// Note that MAX_SAFE_INTEGER is 9007199254740991
function toNumber_BE(bytes, len, signed) {
```

```
    var res = 0;
    var isNeg = false;
    if (len == 0) {
        len = bytes.length;
    }
    if (signed) {
        isNeg = (bytes[0] & 0x80) != 0;
    }


    for (var i = 0; i < len ; i++) {
        if (i == 0 && isNeg) {
            // Treat most-significant bit as -2^i instead of 2^i
            res += bytes[i] & 0x7F;
            res -= 0x80;
        } else {
            res *= 256;
            res += bytes[i];
        }
    }

    return res;
}
function int16_BE(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    return signed(bytes[0] << 8 | bytes[1] << 0, 2*8);
}
function int32_BE(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    return toNumber_BE(bytes, 4, true);
}
function uint16_BE(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    return bytes[0] << 8 | bytes[1] << 0;
}
function uint32_BE(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    return bytes[0] << 24 | bytes[1] << 16 | bytes[2] << 8 | bytes[3] << 0;
}
// float32([62, 132, 168, 155]) = 0.305068
function float32(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    bytes = int32_BE(bytes, 0)
    var sign = (bytes >> 31) == 0 ? 1 : -1; // Comparison with 0x80000000 fails on 32 bit systems!
    var exponent = ((bytes >> 23) & 0xFF) - 127;
    var significand = (bytes & ~(-1 << 23));

    if (exponent == 128) {
        // Some systems might have issues with NaN and POSITIVE_INFINITY, e.g. JSON parsing in GoLang
        // return sign * ((significand) ? Number.NaN : Number.POSITIVE_INFINITY);
                return null;
    }

    if (exponent == -127) {
        if (significand == 0) return sign * 0.0;
        exponent = -126;
        significand /= (1 << 22);
    } else {
      significand = (significand | (1 << 23)) / (1 << 23);
    }

    return sign * significand * Math.pow(2, exponent);
}
function float32_BE(bytes, idx) { return float32(bytes, idx); }

/**
 * TTN decoder function.
 */
function Decoder(bytes, port) {
    var vals = {};
    if( port == 20 ){
```

```
        if (bytes.length==5) {
            // Pressure Probe without temperature sensor and Bridges internal Temperature
            vals["error"] = !!(bytes[0]&0x80);
            vals["pressure"] = int16_BE(bytes, 1)/1000;
            vals["temperature"] = int16_BE(bytes, 3);
        }  else if (bytes.length==7) {
            vals["error"] = !!(bytes[0]&0x80);
            vals["pressure"] = int16_BE(bytes, 1)/1000;
            vals["temperature"] = int16_BE(bytes, 3);
            vals["voltage"] = uint16_BE(bytes, 5) / 1000;
        } else if (bytes.length==9) {
            vals["error"] = !!(bytes[0]&0x80);
            // pressure in mH2O
            vals["pressure"] = float32_BE(bytes, 1);
            // temperature in Degree Celsius
            vals["temperature"] = float32_BE(bytes, 5);
        } else if (bytes.length==11) {
            vals["error"] = !!(bytes[0]&0x80);
            // pressure in mH2O or Bar, depending on probe type
            vals["pressure"] = float32_BE(bytes, 1);
            // temperature in Degree Celsius
            vals["temperature"] = float32_BE(bytes, 5);
            vals["voltage"] = uint16_BE(bytes, 9) / 1000;
        }
    }

    if (port === 64 && bytes.length == 13) { // status packet
        vals["Firmware Identifier"] =  String.fromCharCode(bytes[0]) + String.fromCharCode(bytes[1]) + String.
fromCharCode(bytes[2]);
        vals["FirmwareVersion"] = bytes[3] + '.' + bytes[4] + '.' + bytes[5];
        vals["status"] = bytes[6];
        vals["reboot reason"] = bytes[7];
        vals["final words"] = bytes[8];
        vals["voltage"] = uint16_BE(bytes,9)/1000.0
        vals["temperature"] =  int16_BE(bytes,11)/10.0;
    }
    return vals;
}

function NB_ParseModbusQuery(input){
  vals = {};

  for( var i = 0; i< input.d.batch.length; i++ ){
    if (input.d.batch[i].cmd == "AQMAFgAC"){
      vals["pressure"] = float32_BE(bytes(atob(input.d.batch[i].rsp)),3);
    }
    if (input.d.batch[i].cmd == "AQMAJgAC"){
      vals["temperature"] = float32_BE(bytes(atob(input.d.batch[i].rsp)),3);
    }

    // else: keller
    if (input.d.batch[i].cmd == "AQMAAgAC"){
      // convert to mH2O
      vals["pressure"] = float32_BE(bytes(atob(input.d.batch[i].rsp)),3)*10.197442889221;
    }
    if (input.d.batch[i].cmd == "AQMACAAC"){
      vals["temperature"] = float32_BE(bytes(atob(input.d.batch[i].rsp)),3);
    }
      // vbat
   if (input.d.batch[i].cmd == "+gQABQAB"){
      vals["vBat"] = int16_BE(bytes(atob(input.d.batch[i].rsp)),3)/1000.0;
    }

    // internal temperature
     if (input.d.batch[i].cmd == "+gQABAAB"){
      vals["temperatureInt"] = int16_BE(bytes(atob(input.d.batch[i].rsp)),3);
    }
  }

  return vals;
}
```

```
/**
 * TTN V3 Wrapper
 */
function decodeUplink(input) {
   return {
    data: {
      values: Decoder(input.bytes, input.fPort)
    },
    warnings: [],
    errors: []
  };
}

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

/**
 * ChirpStack decoder function.
 */
function Decode(fPort, bytes) {
    // wrap TTN Decoder:
    return Decoder(bytes, fPort);
}

/**
 * Lobaro Platform decoder function.
 */
function Parse(input) {
    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);
        case "device":
          return NB_ParseDeviceQuery(input);
        case "modbus":
```

```
          return NB_ParseModbusQuery(input);
        case "status":
          return NB_ParseStatusQuery(input);
        default:
      }
      return decoded;
    }


    var data = bytes(atob(input.data));
    var port = input.fPort;
    return Decoder(data, port);
}
```