Hybrid Industrial Gateways

Angelegt von Tobias Kaupat, zuletzt geändert am Sept. 24, 2024

Industrial communication Gateways with external power supply or battery.

LTE-M NB-IoT + LoRaWAN®

Hybrid Connectivity

Model	LOB-GW-DINRAIL-HYB-MODBUS
Order number	• 8000171 - Hybrid Gateway V2 (Battery) • 8000159 - Hybrid Gateway (ext. Power, DIN-Rail) • 8000166 - Hybrid Gateway (ext. Power, 230V) • 8000200 - Hybrid Gateway (ext. Power, 230V) + 20x wired M-BUS Converter

Support for:

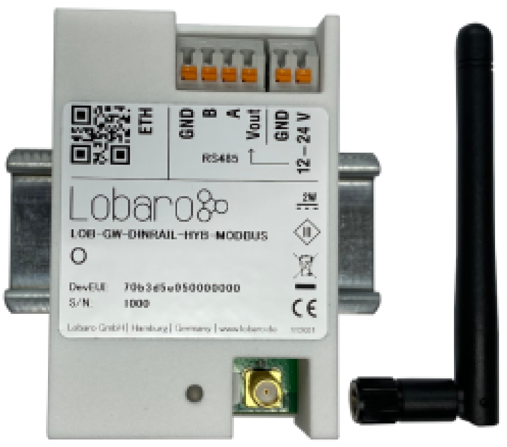
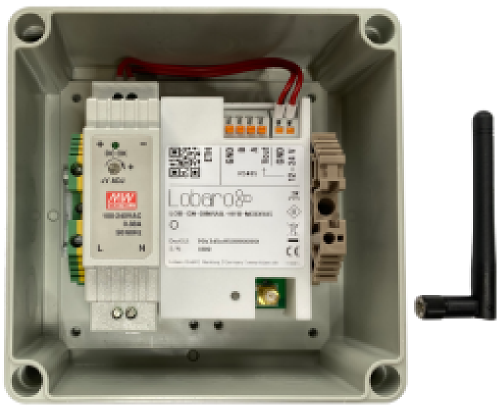
- Modbus ASCII, RTU (RS-485) (all variants)
- Wireless M-BUS (all variants)
- Wired M-BUS (#8000200 only)
- Modbus TCP/IP (Future Firmware on #8000159, #8000166)

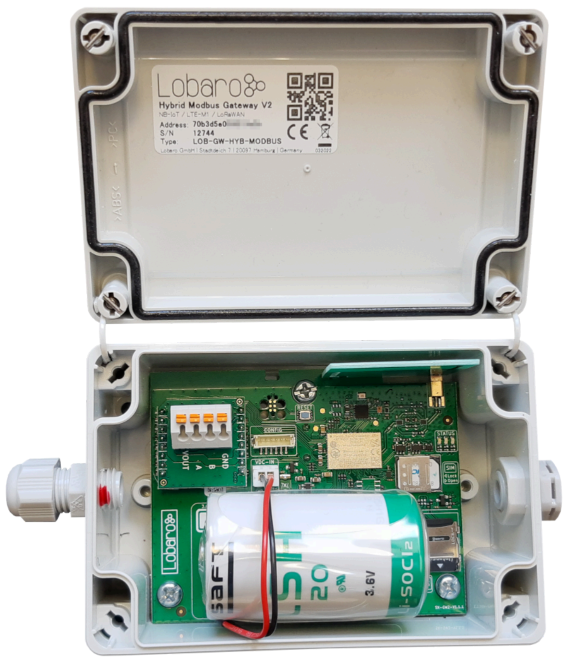
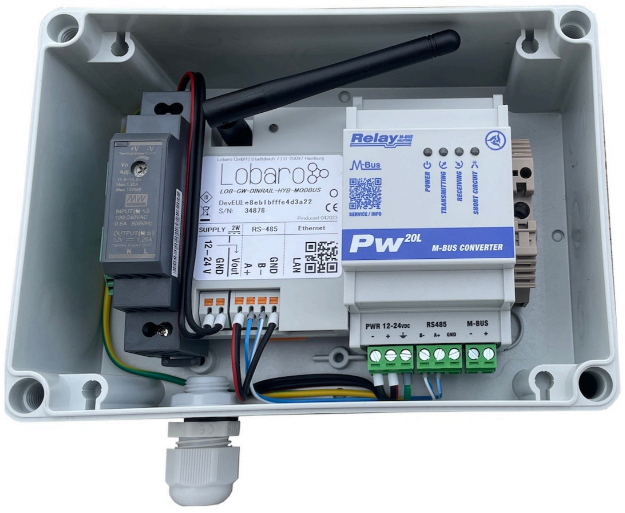
Data communication using:

- LTE-M (all variants)
- NB-IoT (all variants)
- LoRaWAN (all variants)
- LAN (#8000159, #8000166, #8000200)

Current Limitations
The current firmware has some limitations. These features will be implemented in a future release and do not work yet.

- The firmware does **not** support Modbus over TCP, yet.
- Modbus Downlinks are currently **not** supported with LTE-M / NB-IoT - works only with LoRaWAN.

	
8000159 - Hybrid Gateway (ext. Power, Din-Rail) Including: Antenna	8000166 - Hybrid Gateway (ext. Power, 230V) Including: Antenna, 2x cable feedthrough

	
8000171 - Hybrid Gateway V2 (Battery)	8000200 - Hybrid Gateway (230V, 20x M-Bus, NB-IoT, LoRaWAN)

Overview

The *Lobaro Hybrid Modbus Gateway* is a simple to use, cost and energy efficient device that reads, caches and forwards data via Modbus from any number of Modbus enabled devices into the Internet.

The Gateway can be used to communicate with Modbus Slave devices (ASCII/RTU on a RS-485 bus, TCP on over Ethernet) over NB-IoT, LTE-CatM1, LoRaWAN or LAN. Modbus commands can be transmitted via Downlink message to the Gateway and are forwarded by the Bridge to the connected Slave Devices. Received responses are forwarded as Uplink messages to the Lobaro IoT Platform. The Modbus Gateway can also be configured to execute Modbus commands regularly and report the responses at time of execution.

The Modbus Gateway supports reading of all four object types that can be provided by Modbus slave devices: Coil, Discrete Input, Input Register, and Holding Register. It also supports writing values to all writable objects: Coils and Holding Registers. Multiple different slave devices on the Bus can be accessed individually by a single Gateway device, even if the slave devices have a different Modbus configuration. Reading intervals and register definitions can be configured very flexibly to suit individual requirements.

Quick start guide

For details about each Steps please refer to the related detailed sections of the Manual below.

- Make sure the SIM card is inserted correctly when using NB-IoT or LTE-CatM1.
- Connect the Modbus Gateway to your Modbus Slave Devices
    - Via RS485 connector using a twisted pair cable: `A` to `A`, `B` to `B`, and `GND` to `GND` (`GND` is not strictly necessary but enhances the connection. Not all slave devices supply a `GND` connector).
    - Via ETH connector for Modbus TCP using a RJ45 LAN cable(externally powered variants only).
- Connect the Modbus Bridge to a computer using the Lobaro Configuration Adapter and the Lobaro Maintenance Tool.
- Connect power to the device - powering with the configuration adapter does not work.
- Make sure the configuration is correct to connect to the Internet (depends on you connection method: Mobile, LoRaWAN or LAN.
- Make sure the configuration is correct to read out your desired Modbus device (e.g. ASCII/RTU, Baud, Data Length, Stop Bits, Parity and Modbus Command).
- Optionally: Switch to the Log tab of the Lobaro Tool to see if the device is connecting and working as expected.
- Go to The Lobaro Platform and log into your account.
- Go to "Devices" and select your "Hybrid Modbus Gateway".
- If you have several Gateways: the "Address" is printed on the device's case as "DevEUI".
- You should see all Uplinks the Gateway collected sent far. Next step is to configure your individual Device Type to display you slaves data or forward data to your own IT.

Supported Devices

The Lobaro Modbus Gateway works with **all devices that act as a Modbus Client using RTU, ASCII (or, in a future release TCP)**. Some devices that have been used successfully with the Gateways are:

Device	Type	Manufacturer	More information	Sample Implementations
Octave Ultrasonic Meter	Water meter	Arad Group	External Link	
ECL Controller	Heat/Hot Water Regulation	Danfoss	External Link	
UMD 97	Smart Grid Power Meter	PQ Plus	External Link (German)	
DRS458DE	Power Meter	B+G E-Tech GmbH	External Link	
Feuchtemessumformer PCE-P18 Modbus RTU	Humidity / Temperature sensor	PCE-Intruments	External Link (German)	
Lobaro Pressure Sensor	Pressure Sensor	Lobaro		Pressure Sensor Application

Modbus Introduction

For an overview about the Modbus protocol please refer to our documentation page about Modbus.

For a deeper introduction into Modbus please visit https://en.wikipedia.org/wiki/Modbus, which is a good place to start.

Please be aware that to configure the Gateway correctly, you will need to know the details of your installation and the slave devices you are trying to connect. Debugging a failing Modbus setup is quite complex, as there are many potential errors that cause the problems.

Please also note that even when you succeed in reading the data you need from the correct registers, neither the Gateway, nor the Lobaro Platform can know, what that data means. There is no semantic defined in Modbus; without additional information, the data is just bytes, that need to be interpreted by custom software. Registers can hold boolean values, signed or unsigned integers, floating point numbers or anything else. Often values span over multiple registers (each register holds exactly 2 bytes, so that e.g. 32bit integers will need two registers. There is also no dedicated byte order for Modbus, so that is not even trivial to read integers. This is a problem that is intrinsic to using Modbus and cannot be solved easily by a generic bridge device.

If you need support for creating a custom parser that processes the Uplinks and convert it to an easy to use format, please contact us, so that we can send you an offer for your individual use case.

Setting up the device (Din-Rail variants)

Interfaces

Connections as on the label:

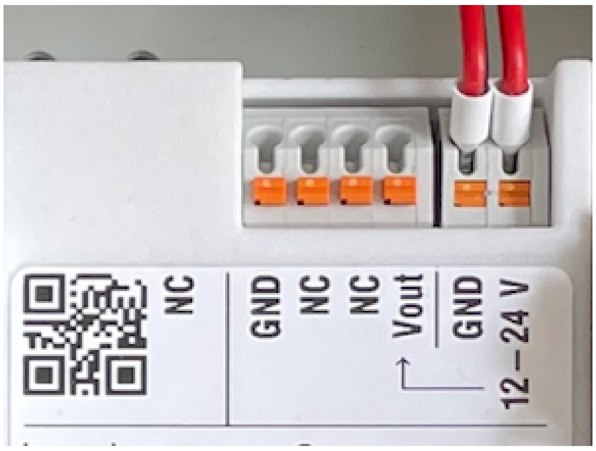
1. Vin - Supply voltage 12 - 24 Volt DC
2. GND - Ground
3. Vout - same as Vin
4. A - Modbus ASCII/RTU line
5. B - Modbus ASCII/RTU line
6. GND - Ground
7. ETH - Ethernet connection for LAN Uplink or Modbus TCP

- Connected cables must be between 0.05 mm² (AWG30) and 1.31 mm² (AWG16).
- Inserted cable length must be between 6mm and 7mm.
- Recommended wire termination:

    - Weidmüller: H0,14/10 GR SV, Article No.: 9005180000, 8mm/6mm, max. AWG26

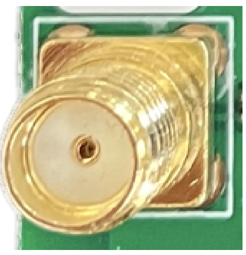
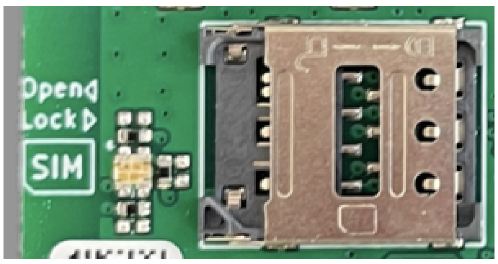
> ⚠
> - Cables must not be connected or disconnected while the device is powered.
> - Vout only supports small consumers like sensors with ≤ 1W. If in doubt, connect sensors to a dedicated power supply according to their manufacturer's instructions.

Power Supply



- Power supply via external mains adapter with 12 - 24 Volt DC
- Power output must be at least 2W and maximum 100W.

Radio (LoRa, FSK, NB-IoT, LTE Cat-M1)

		
Female SMA Connector	Antenna	Nano SIM (4FF) for mobile connectivity

The antenna is used for different radio technologies based on LoRa, FSK including LoRaWAN® and wireless MBUS S1, C1/T1 Modes (868 MHz), OMS v3 & v4.

The same antenna is also used for communication via NB-IoT and LTE Cat-M1. A Nano SIM (4FF) is required for mobile connectivity.

> ⚠ The SIM Card must not be inserted or removed while the device is powered.

Properties of compatible SMA Antennas:

SMA Joint Rod Antenna (LTE, LoRa)	
Frequency range	698-960 / 1710-2700 MHz
Length	108 mm
Antenna Gain	2 dBi
V.S.W.R	<= 2.5
Radiation	Omnidirectional

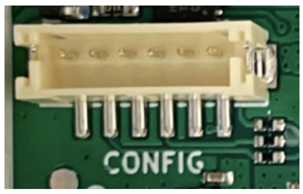
Polarisation	Vertical
Max. Power	5 W
Impedance	50 Ohm
Connector	SMA Male
Material of dome	TPE
Lobaro Article No.	3000413

⚠ The device was only tested with the listed antenna. Lobaro does not take liability for use with different antennas.

RESET-Button

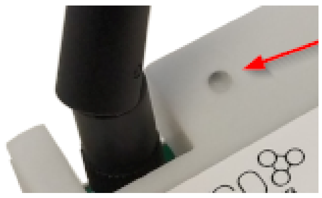
- Press the RESET Button to restart the device
- The RESET Button can be pressed through a small hole in the cover (e.g. with a paper clip). The position is marked with a ring on the label.

Config Port



- The config port can only be reached when the cover is removed
- The configuration port is compatible with our 6-pin Lobaro USB Configuration Adapter (Article No. 8000005).
- A free to use configuration tool can be downloaded from the Lobaro website.

LED



- Status information is visualized via the RGB LED

Mobile operator and LTE band configuration

If you are using a different mobile operator than pre-configured, you should change the mobile operator code set in the Config Parameters `Operator` and (LTE) `Band` Operator codes are 5 digit codes that indicate country and operator.

ⓘ For details about configuration for mobile network operation please refer to our article about LTE / NB-IoT Networks

Configuration

The device is shipped with default configuration parameters. The configuration can be changed via the 6-pin config port using the Lobaro USB Configuration Adapter.

More information about the usage of the configuration tools can be found in our documentation.

ⓘ **Remote Configuration** is also supported after initial network connection.

Networking Parameters (NB-IoT / LoRaWAN)

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 • `ethernet`: use LAN/Ethernet (parallel lte-m or nbiot not possible)
`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 ope
`Band`	NB-IoT Band	`8`	"8", "20", "8,20", Empty = Auto detect (long
`APN`	Mobile operator APN (optional)	`iot.1nce.net`	1nce: iot.1nce.net

			Vodafone Easy Connect: lpwa.vodafone.cc
`PIN`	SIM PIN (optional)		Empty or 4 digits (e.g. `1234`)

LoRaWAN Parameters (WAN = "lorawan", "lorawan-abp")

Name	Description	Default Value	Value Description & Examples	Since
`DevEUI`	DevEUI used to identify the Device	Device's own DevEUI as printed on label	8 bytes = 16 hex digits, e.g. `0123456789abcdef`	
`JoinEUI`	EUI used for OTAA (aka `AppEUI`)	Individual default value for each device	8 bytes = 16 hex digits, e.g. `0123456789abcdef`	
`AppKey`	AES Key used for LoRaWAN	Individual default value for each device	16 bytes = 32 hex digits, e.g. `0123456789abcdef00112233445566 7788`	
`SF`	Minimal Spreading Factor used	`12`	`7-12`, used after reset, can be decreased by ADR during operation (but not increased)	
`OpMode`	LoRaWAN Operation Mode	`A`	A = Class A, C = Class C	v0.2.1

Keep the value of `AppKey` secret. If you change it, make sure you are using a good random source.

OTAA - Over the Air Activation

The preferred method to use LoRaWAN is Over The Air Activation (OTAA). When `WAN="lorawan"` the device uses the values to perform an OTAA Join with the LoRaWAN Network Server. Make sure the values for DevEUI, JoinEUI, and AppKey match.

The device is manufactured with a globally unique EUI64 that is used as DevEui. This EUI is printed on the devices label and can be used to identify a device. You can change the DevEUI used for LoRaWAN by changing the configuration parameter `DevEUI`. The device will still keep it's unique EUI64 it was delivered with. You can see it in the Log output during booting, even if a different value is used for the DevEUI.

Each Device will be configured with a unique JoinEUI and AppKey that are generated using a cryptographic hashing algorithm. Those values will seem random and are very likely to be unique for each device. These values are known to Lobaro but will not be made public. You can change the `AppKey` if you prefere to have Keys that are known only to you. *Be sure to use a good random source when generating keys.*

ABP - Activation by personalisation

Our devices support activation by personalisation (ABP) when `WAN="lorawan-abp"`. This mode is useful for devices that have a bad reception. You will have to synchronise session keys by hand between the device and your Network Server when using ABP.

When using ABP, the device will use the parameters `DevEUI` and `AppKey` for generating the session parameters:

- `DevAddr` will consist of the last for bytes of `DevEUI`
- `NetSKey` will be cryptographically derived from `AppKey`
- `AppSKey` will be cryptographically derived from `AppKey`

The Values will be printed out in the Log after boot, so you can copy them to the configuration in your Network Server. Do change the `DevAddr`, alter the `DevEUI` value in the configuration. To create a different pair of session keys, change the value of `AppKey` in the configuration. Best practise is, to change it to randomly generated bytes coming from a good random source. The generated Session Keys will be deterministic for a given value of `AppKey`, even if used on different devices.

LAN Parameters (WAN = "ethernet")

Connection via LAN/Ethernet is not supported, yet.

⚠ Coming soon!

Modbus related Parameters

Name	Description	Default Value	Values Description & Examples
`MbCmd`	List of Modbus Commands with Cron and Modbus parameters (see below).	`0 0/5 * * * *:R,9600,8N1:010300000003`	Zero or more entries of Modbus commands to be executed by the device. Each entry starts with a Cron expression defining when to execute the commands followed by the bus parameters used to address the Modbus slave devices. Each entry can contain multiple commands. See description below for a detailed explanation. †

† See also our Introduction to Cron expressions.

`MbCmd` defines, what Modbus commands are executed on the Bus, when, and what Modbus Configuration to use for them. The configuration is very flexible and allows complex setups, that include executing different commands at individual intervals or times or using multiple different Modbus parameters to address incompatible Slaves on the same installation. Any Modbus command can be sent, including writing registers or diagnostic messages.

The parameter consists of up to 32 entries, separated by semicolons. Each entry consists of three parts, separated by colons: its individual Cron expression, the Modbus configuration used, and the Modbus commands to be executed. Each entry can have multiple Modbus commands to execute on activation, separated by comma. A Modbus command must be written as the Bytes to be sent on the bus in Hex notation. The check sums will be added by the device according to the protocol used.

The command is entered in Hex and without any check sums and is 6 bytes long (12 hexdigits). The default value is `010300000003`, it consists of 4 hex parts: `01`, `03`, `0000`, `0003`

0x01

Address of the Slave Device. 1 byte: often 01 new devices

0x03

What kind of Modbus Register to read. 1 byte. 03 stand for Holding Register.

0x0000

Number/address of the first register to read. 2 bytes. Many devices have some value to read out at 0000.

0x0003

Number of consecutive registers to read from the first register. 2 bytes. This would read the registers #0, #1, and #2 in one command.

The default value of `0 0/5 * * * *:R,9600,8N1:010300000003` shows a very basic example with a single entry executed every 5 minutes using Modbus RTU to read 3 consecutive holding registers from a single slave device.

Complete Syntax of MbCmd

```
MbCmd        = "<Entry1>;<Entry2>;...;<Entry32>"
Entry        = "<Cron>:<MbParm>:<Command1>,<Command2>,...,<CommandN>"
MbParm       = "<Protocol>,<Baud>,<SymbolCfg>"
Protocol     = "R" for Modbus RTC, "A" for Modbus ASCII
Baud         = Baud rate, any of: 2400, 4800, 9600, 19200, 38400, 57600, 115200
SymbolConfig = Token string defining Data Length, Parity, and Stop Bits. Any of: "7E1", "7E2"
Command      = "<bytes to be sent in hex without checksum>"
```

Examples

```
Example A:
"0 0/5 * * * *:R,9600,8N1:010300000003"
  Entry 1:
    Cron:   "0 0/5 * * * *": Execute entry every 5 minutes, on minutes 0, 5, 10, 15, ..., 55
    Config: "R,9600,8N1": Use Modbus RTU on 9600 Baud, Datalength: 8, Parity: None, 1 stop bi
    Commands:
      "010300000003": Read 3 holding registers of Slave 1, starting at register 0

Example B:
"0 * * * * *:A,9600,7E1:0e0400100004,0f400100004;0 0 * * * *:A,9600,7E1:0e0400200020"
  Entry 1:
    Cron:   "0 * * * * *": Execute entry every full minute
    Config: "A,9600,7E1": Use Modbus ASCII on 9600 Baud, Datalength:7, Parity: Even, 1 stop b
    Commands:
      "0e0400100004": Read 4 input registers of Slave 14, starting at register 16
      "0f0400100004": Read 4 input registers of Slave 15, starting at register 16
  Entry 2:
    Cron:   "0 0 * * * *": Execute entry every full hour
    Config: "A,9600,7E1": Use Modbus ASCII on 9600 Baud, Datalength:7, Parity: Even, 1 stop b
      "0e040a800020": Read 32 input registers of Slave 14, starting at register 2688
```

Maximum registers / salves

The maximum amount of registers / slaves that can be setup is only limited by the available configuration memory, which is about 4 kB in total. In doubt try out a long MbCmd using the configuration adapter and see if it fits into the device. If not the device log would indicate it and the device enter a reset loop. If the config will be set remotely the previous config would be restored.

Wireless M-Bus related Parameters

Subject to change! Many feature of this device are still under development; wireless M-Bus support is still experimental. When updating firmware, check the changelog for breaking changes.

Name	Description
`wMbusCmd`	List of wM-Bus collection entries with cron, mode, and duration. Multiple entries can be added, separated by "`;`". See below for examples.
`mFilter`	wMBus manufacturer filter sep. by "`,`" e.g. "`dme,itw`" (Comma separated list WITHOUT spaces)
`typFilter`	wMBus device type filter e.g. `08,07` for Heat Cost and Water
`devFilter`	meter id filter e.g. • cmode, smode (wmbus): `06198833` (up to 8 digits with leading 0) • xmode (Sensus RF): `10121335300` (exactly 11 digits, no "-"!) **`(Comma separated list WITHOUT spaces: "88009035,13456035,56268931")`** Up to 500 wMBus-IDs or 400 Sensus-RF-IDs are supported.
`ciFilter`	Collect only telegrams with specific values in the ci-Field, must we written as 2 hex digits (with leading zeros). **`(Comma separated list WITHOUT spaces, e.g.: "8a,07,71")`**
`maxTelegrams`	Set hard limit on how many telegrams will be collected and uploaded. The bridge will stop collection, once this number has been collected, regardless of the passed time. Can be used save battery / data volume, should the device be in an area with a large number of meters. Set to `0` for no limit.

Example commands

wMbusCmd	Description
`0 0 12 * * *:C=180`	Each day at 12h UTC collect C/T-Mode for 180 seconds.
`0 0 5,17 * * *:X=100,U=120`	Each day at 5h and 17h UTC collect Sensus-RF for 100 seconds and Müller-Funk for 120 seconds.
`0 0 * * * *:C=120;0 0 10 * * *:S=120`	Collect C/T-Mode for 120 seconds every hour. Collect S-Mode every day at 10h UTC for 120 seconds.

Wired M-Bus related Parameters

Subject to change! Many feature of this device are still under development; wired M-Bus support is still experimental. When updating firmware, check the changelog for breaking changes.

You need an M-Bus converter attached to the RS-485 connector of the Industrial Gateway to use wired M-Bus. Lobaro offers a complete solution including a converter under order number 8000200.

Name	Description
`MbusCmd`	List of M-Bus reading entries with cron, baud rate and list of slave adresses. Multiple entries can be added, separated by `";"`.

Entries

Entries in `MbusCmd` must be of the form `"<cron>:<baud rate>:<address>,<address>,..."`. Only primary addresses are supported.

Scanning

The Gateway supports scanning for slave devices on the bus. This can be activated by using `"*"` instead of an address list. The Gateway will run a scan process once after booting. Scanning will take about 1 minute on Baud rate 2400. A special discovery message will be uploaded after booting that lists all discovered devices.

Examples

MBusCmd	Explanation
`0 0 * * * *:2400:*`	Run slave discovery with 2400 Baud on boot. Readout discovered Slaves once every hour and upload telegrams.
`0 0/15 * * * *:4800:1,2,4,5`	Read slaves with addresses 1, 2, 4, and 5 with 4800 Baud every 15 minutes.
`0 0 12 1W * *:9600:*;0 0 * * * *:9600:1`	Two separate entries that will be executed using different cron expressions (both use 9600 Baud for communication): • `"0 0 12 1W * *:9600:*"`: The Gateway will run the scan process on boot to find all devices attached to the bus. The devices will all be read on the first weekday of each month at midday (UTC). • `"0 0 * * * *:9600:1"`: The Gateway will read the slave with address 1 once on every full hour.

Modes of operation (work cycle)

Subject to change! This product is still very young and experience might lead to adjustments in the future.

This chapter explains how the device starts and works to collect and upload data.

Startup process

The starting process of the device is linear and executed the following steps in the order given here. The startup is triggered on power on or after a reset was triggered; this can happen over the reset button, by using the Lobaro tool over the config adapter, by sending a reboot command via Downlink, of if a fatal error occurs during operation of the device.

- On power on (or after a reset), the device will start by verifying the signature of the installed firmware. It will only continue the boot process if the signature is valid.

- The device will then activate the Arm TrustZone of its central processor, to prepare an isolated environment for the actual firmware to run. This activates hardware security mechanisms that protect the device from a wide range of errors and manipulations while it is running.
- Only then, the application program is started, having restricted access to the hardware.
- The application reads and outputs the configuration parameters programmed into it and verifies the configuration does not contain any obvious errors (like invalid syntax or impossible value combinations). If any fatal errors exist, the device will output information about it in the Log and then reboot, so that you spot invalid configurations directly when you set them.
- At this point the RGB-LED starts to continuously output the state of the Modbus connection and the connection to the backend.
- The device will try and execute every Modbus command from every entry in the configuration parameter `MbCmd`, using the Modbus configuration of the entry (writing operations are skipped to avoid side effects). This allows you to check your Modbus configuration and the connection to the slave devices early on boot and even without a config adapter attached. The device will continue execution, even if the Modbus Commands fail, so that you can check the connection to the backend even when not connected to the Modbus slaves.
- After testing Modbus, the device will try to connect to the network for connection to the backend. The exact action will depend on your configuration:
    - For **NB-IoT** or **LTE-M** the Gateway will activate its Modem and try to attach to the mobile provider. If that succeeds, it will try to connect to the backend configured (normally an instance of the Lobaro Platform). If that succeeds, the device will upload some information about itself to the backend (including its configuration) and than synchronise its internal clock over the connection. Only if all this succeeds, the device will move on.
    - For **LoRaWAN** with **OTAA**, the device will perform an OTAA Join operation with the LoRaWAN Network Server. If that succeeds, the device will upload a Status Message that will also be used to synchronise its internal clock with the network. Only if that succeeds, the device will move on. On failure, the device will retry with increasing timeouts to perform the join operation and time sync, until it succeeds.
    - For **LoRaWAN** with **ABP** there is no explicit joining operation, as the session must already exist between Network Server and Device. It simply uploads a Status Message that will also be used to synchronise the internal clock. If the synching fails multiple times, the device will skip synchronisation and just move on to start normal operations. *Be aware that this can lead to the Device operating with a clock that does not match real time.* The Modbus Commands will be executed according to that clock, which will most likely not be consistent with what you expect. This exception is introduced on purpose, as ABP is meant to make the device usable in locations, where there is poor Downlink reception from the Network, where an OTAA Join cannot be performed, but Uplink messages still might be coming through to the Network. The device will continue to try and synchronise its clock every day and might succeed at some point in the future. The changing of the clock will be compensated as well as possible, but the exact times when commands are executed is most likely to change at that point.
- When the connection to the backend has been established (this is not certain to have succeeded when using LoRaWAN ABP), the device will start its internal scheduler and will from this moment on be running in normal operation mode.

Normal operation

The actions executed by the device during normal operation are controlled by a scheduler that executes a list of jobs whenever it is their time to run. Only a single job will be executed at any time, so if a job is running for some time, other jobs will be executed delayed (but the execution will not be skipped). Each entry you add to the `MbCmd` config parameter will have its own job in the scheduler. The Cron Expression of the entry will control, how often and when the job will be executed. In addition to that there will be a Status job which runs once every day and triggers the upload of a status message which will also perform a clock synchronisation. When your configuration has multiple entries that are scheduled for the same time, they will be executed in the order you put them in the configuration.

The jobs will generate Uplink messages that need to be uploaded. Those will be queued and upload in the order they are generated. The device will continue to execute jobs while handling uploads. When Uplinks are generated faster than they can be sent, the queue will run full and new Uplinks will be dropped silently. This is most likely to happen when using LoRaWAN on higher spreading factors, as the Gateway can read data over Modbus much faster than it can be sent over LoRaWAN.

After each Uplink sent, the device will look for Downlinks coming from the Network (this is done for both, LoRaWAN and LTE configurations). Downlinks can contain remote commands controlling the device (like configuration changes, reboot requests, or (for LTE only) remote firmware updates). There can also be Modbus commands sent via Downlink that will be executed on the bus by the Gateway directly (the response from the slave will be sent as Uplink). *Downlink Modbus commands are currently only supported for LoRaWAN.*

The daily Status message Uplink makes sure that the device can be reached for remote configuration within 24 hours, independent of the current configuration.

Changing configuration or performing a firmware update will result in the Gateway rebooting. We try our best to keep our devices from ever reaching a state that makes them unreachable. A new configuration set via Downlink will be temporary until a connection to the Network can be established again. If the new configuration fails to connect to the Network, the previous configuration is restored.

LED patterns

DIN-Rail Variant

The DIN-Rail Hybrid Modbus Gateway has an RGB LED that can be seen with closed or open box. It signals the state of the device.

- special patterns:
  - constant MAGENTA: Device is in Bootloader Mode (not actively running, remove config adapter and press reset to leave Bootloader Mode).
  - GREEN on for ~2 s and off again: Device just booted, either after power on, reset b button or software, or after a hard failure.
  - CYAN flashes on and off once a second: Device is is Dialog Mode. To leave the Dialog Mode, change the configuration using the config adapter.

During normal operation (when the device finished booting and is not running any special modes), the device will continuously flash its LED twice, each flash giving a different piece of information:

- The first flash indicates the state of the connected sensors/Modbus slaves. This state is first set when the device tests all Modbus commands after boot. It will be updated during operations whenever Modbus commands are executed.
  - BLUE: State of the sensors has not been determined. This is the state after booting, while the device is trying to execute all Modbus commands in its configuration.
  - GREEN: All Modbus commands from config have been executed successfully.
  - RED: None of the Modbus commands could where successful. There is most likely a problem in the connection (wires) or in the configuration.
  - YELLOW: At least one Modbus command has been successful, but at least one Modbus command failed. Will only happen when multiple Modbus commands and/or entries are present in `MbCmd`. This can happen though a miss-configuration, when one of multiple slaves could not be reached, or if only one entry has failed and one succeeded - maybe they are executed during different times and the disruption only lasted for a short while.
- The second flash indicates the state of the connection to the backend. This can be a connection to the Lobaro Platform via NB-IoT/LTE-M or Ethernet, or a LoRaWAN connection to a Network Server.
  - BLUE: State of connection has not been determined, yet. This is the situation after boot, when the device has not yet finished connecting.
  - GREEN: Connection to the Platform/Network Server has been established successfully.

Battery Variant

Since v0.8.5: The Battery Hybrid Modbus Gateway has three separate LEDs, Blue, Green, and Red (left to right) in a square labeled "STATUS" 🔵 🟢 🔴 . The patterns look different from the DIN-Rail variant, as the colours are not combined:

- special patterns:
  - 🔵 ⚪ 🔴 constant BLUE and RED: Device is in Bootloader Mode (not actively running, remove config adapter and press reset to leave Bootloader Mode).
  - ⚪ 🟢 ⚪ GREEN on for ~2 s and off again: Device just booted, either after power on, reset b button or software, or after a hard failure.
  - 🔵 🟢 ⚪ BLUE and GREEN flashes on and off once a second: Device is is Dialog Mode. To leave the Dialog Mode, change the configuration using the config adapter.

During normal operation (when the device finished booting and is not running any special modes), the device will continuously flash its LEDs twice, each flash giving a different piece of information. The information is repeated every second for the first 15 minutes. After 15 minutes, the interval is changed to only give the information once per minute, to save battery power.

- The first flash indicates the state of the connected sensors/Modbus slaves. This state is first set when the device tests all Modbus commands after boot. It will be updated during operations whenever Modbus commands are executed.
  - 🔵 ⚪ ⚪ BLUE: State of the sensors has not been determined. This is the state after booting, while the device is trying to execute all Modbus commands in its configuration.
  - ⚪ 🟢 ⚪ GREEN: All Modbus commands from config have been executed successfully.
  - ⚪ ⚪ 🔴 RED: None of the Modbus commands could where successful. There is most likely a problem in the connection (wires) or in the configuration.
  - ⚪ 🟢 🔴 RED and GREEN: At least one Modbus command has been successful, but at least one Modbus command failed. Will only happen when multiple Modbus commands and/or entries are present in `MbCmd`. This can happen though a miss-configuration, when one of multiple slaves could not be reached, or if only one entry

has failed and one succeeded - maybe they are executed during different times and the disruption only lasted for a short while.

- The second flash indicates the state of the connection to the backend. This can be a connection to the Lobaro Platform via NB-IoT/LTE-M or Ethernet, or a LoRaWAN connection to a Network Server.
    - 🔵 ● ● BLUE: State of connection has not been determined, yet. This is the situation after boot, when the device has not yet finished connecting.
    - ● 🟢 ● GREEN: Connection to the Platform/Network Server has been established successfully.

NB-IoT / LTE-M

The Lobaro Platform

The easiest way to work with the *Lobaro Modbus Gateway* is the *Lobaro Platform*. You can find it under https://platform.lobaro.com – Log in with the credentials provided by Lobaro.

Your Gateways should be listed under "Devices". If you have multiple devices in your account, you can distinguish them by the field "Address". The Address is printed on the box of the Gateway (it is the DevEUI that is also used for LoRaWAN).

The Industrial Gateway uses CoAP(s) to communicate with the Lobaro Platform. This documentation shows only the Payload of the CoAP messages. The data is sent using CBOR, which is a binary format that works a lot like JSON, but uses less bytes. In this documentation the messages are written as JSON, so that they can be read by humans. CBOR supports binary data. In JSON, binary data is represented by strings containing the bytes encoded in Base64.

Envelope

Each uplink is enclosed in an envelope that contains 4 values:

Uplink envelope

```
{
    "i": "70b3d5e050010001",
    "n": 19,
    "q": "...",
    "d": { ... }
}
```

Key	Meaning	Explanation
"i"	Identification	The device's address (DevEUI), as printed on the device. A string.
"n"	Number	Uplink Frame Counter, incementing number of uplink, starting with 1 on device boot. A number.
"q"	Query	The type of the message sent in the uplink, e.g. "status", "mbus". A string.
"d"	Data	The payload of the message. An object.

The payload (key "d") is different for each message type (key "q"). The list that follows only shows the payload for those message types.

Upload type "device"

The first uplink sent to the platform after boot. It contains information about the device that don't change during execution, like addresses and firmware names and versions. It also contains information about why and how the device booted that can help when analysing problems.

Payload for "device" uplink Quelle erweitern

Upload type "config"

The second uplink sent after boot is the complete configuration of the device. Any change of the device's configuration is only applied after a reboot, so this configuration is valid for the complete runtime of the device until the next reboot (which will upload the configuration again).

Payload for "config" uplink Quelle erweitern

Upload type "status"

The "status" uplink is sent at boot after the config and once every 6 hours. It contains health information about the device and information about the connection.

The message is sent 4 times a day, so that problems with the devices will be noticed quickly. It also allows always reaching the device via downlink within 6 hours.

Payload for "status" uplink Quelle erweitern

Upload type "modbus"

The "modbus" uplink is sent for each entry in `MbCmd` on execution of the entry, after boot and when the cron triggers. It contains a list of the executed commands (excuted in their order in the entry). For each command the command itself, the response to it, the error code (0 for no error), and the time of the execution is included. When an entry has a lot of data and the capacity of a single uplink is exceeded, entries will be split into multiple uplinks.

Payload for "modbus" uplink Quelle erweitern

Upload type "wmbus"

The "wmbus" uplink sends collected wireless M-Bus telegrams as batch. The telegrams are sent raw and without CRCs along with some Metadata (time of reception, reception signal quality, and format and mode of the telegram). The Gateway will fit as many telegrams in a single uplink as possible. If there are too many telegrams for a single uplink, multiple uplinks will be sent.

Payload for "wmbus" uplink Quelle erweitern

Upload type "mbus-scan"

The "mbus-scan" uplink sends the result of a wired M-Bus device scan. It is only sent once after boot, as the scan is only executed once after boot.

Payload for "mbus-scan" uplink Quelle erweitern

Upload type "mbus"

The "mbus" uplink sends wired M-Bus telegrams after they have been send requested via the Bus.

Payload for "mbus" uplink Quelle erweitern

LoRaWAN Payload formats

The Industrial Gateway has a lot of different functions and therefore uses a lot of different ports and formats for LoRaWAN. They are all listed in the following table for quick reference. They are described in detail, grouped by their functions.

Any numerical data that is added by Lobaro's format will be sent using Big Endian (aka Network Byte Order). Data contained inside M-Bus telegrams or data read from Modbus slaves is sent as it is read; the byte order used there is dependent on the devices that generate the data. Timestamps will be uploaded as signed Big Endian 40 bit integers holding a UNIX timestamp.

Where payload formats are described, byte positions are counted beginning with 0. Values are often written in hex.

Overview

Direction	Function	Port	PlFmt Value	Message
Uplink	Modbus	3	1	Modbus Responses triggered by configuration.
Uplink	Modbus	4	any	Modbus Responses triggered by Downlinks.
Uplink	Modbus	5	any	Continuation of Responses that do not fit in a single Uplink.
Uplink	Modbus	20-59	4	Compact payload format with timestamp
Uplink	Modbus	20-59	5	Compact payload format without timestamp
Uplink	Status	64	any	Status messages
Uplink	wM-Bus	102		wM-Bus telegrams with meta data. Split for long telegrams
Uplink	M-Bus	110		Discoverd M-Bus slave addresses
Uplink	M-Bus	112		M-Bus telegrams with meta data. Split for long telegrams
Uplink	Config	128	any	Remote configuration response
Uplink	Config	129-131	any	Remote configuration long response 129 = start, 130 =

				middle, 131 = last
Downlink	Modbus	4	any	Modbus Commands to be forwarded by the Bridge.
Downlink	Config	128	any	Remote configuration

Status Message (Up, Port 64)

The Industrial Gateway sends a status message several times a day. From this you will quickly know, if there is a problem with the device in the field. It will also allow you to reach the device via downlink on a short notice, whatever its configuration is. The status message will be sent every 6 hours, so 4 times a day. The exact time is different for each individual device, but will be consistent for each specific device.

Status messages are transmitted on port 64 and have a fixed length of **13 bytes** (might get longer in future versions).

Structure of a message on port 64

```
Bytes | 0 . 1 . 2 | 3 . 4 . 5 | 6 . 7 . 8 | 9 . 10  | 11 . 12     |
      +-----------+-----------+-----------+---------+-------------+
Data  | firmware  | version   | rfu       | voltage | temperature |
```

Firmware

Bytes 0 to 2 contain three ASCII chars that identify the firmware running on the device. Values are `MBB` for the Battery Gateway and `MBD` for the DIN-rail Gateway.

Version

Bytes 3 to 5 hold the version of the firmware running on the device, encoded in 3 independent unsigned 8 bit integers. Example: 0x00 0x09 0x1a stands for v0.9.26.

RFU

Bytes 6 to 8 are reserved for future use.

Voltage

Bytes 9 and 10 hold the voltage supplied by the power source in `mV`, encoded as an unsigned 16 bit big endian integer. Example: `0x0DDB` = `3547` = `3.547 V`.

Temperature

Bytes 11 and 12 hold the Gateway's internal temperature (measured inside the µC), encoded as a signed 16 bit big endian integer, holding the temperature in tenth of °C. Examample: `0x00F6` = `246` = `24.6 °C`.

Modbus - two different formats

The Industrial Gateway supports two very different formats for uploading Modbus data via LoRaWAN.

The verbose format (`PlFmt = 1`) includes all information needed, so that the Modbus command and the response received can be read from it. It will also include error codes in case the communication did not work. It is more complex to parse, and it will not use the datarate given by LoRaWAN efficiently, because every uplink will contain the request used.

The compact format (`PlFmt = 4` and `PlFmt = 5`) only upload data from the response received, so they will be using LoRaWAN's limited datarate more efficiently. To understand the data in the uplinks, the backend will need to know the exact configuration of the commands. Errors in communication can be detected, but not as exactly as in the verbose format.

You need a solid understanding of how Modbus works, if you want to understand the formats explained here. Explaining Modbus is outside the scope of this document. You can find a good introduction on the English Wikipedia: https://en.wikipedia.org/wiki/Modbus.

Modbus Verbose Format (Up, Ports 3 to 5)

When `PlFmt` is set to `1`, the Gateway will send Modbus uplinks in the verbose format on port 3. The format contains the complete response to each command. It has all additional information to know what command was sent, even if it runs into an error. The format addapts to changing Spreading Factors and tries to use the available payload capacity. If a response is too long for the payload capacity, messages are split into parts. For split uplinks, the first one will also be sent on port 3, following parts (that need to be re-attached) will be sent on port 5. It is possible to write configurations that will never need to split a message. Just make sure that no response will be longer than the minimal available uplink size (respecting the additinal bytes of this format). The device will try to fit as many responses as possible into a single uplink, without changing the order in which they are sent. If the next response will not fit, a new uplink will be started for it. Only responses that do not fit in a single uplink will ever be split. Responses from separate entries to `MbCmd` (with their own cron) will never be put into the same uplink.

Commands that are executed from downlinks sent on port 4 will also use the Verbose Format. The responses to downlinks will be sent on port 4 instead of 3, the rest stays the same. If those responses are split over multiple uplinks, the additional parts will also be sent on port 5.

Take a look at the example uplinks and at the reference parser in this document, to get a better understanding on how this format works.

Structure of a message on port 3

```
Byte | 0 . 1 . 2 . 3 . 4 | 5 ...      | ...        | ... | ...        |
     +-------------------+------------+------------+-----+------------+
Data | timestamp         | response 1 | response 2 | ... | response n |
```

Structure of a response part on port 3

```
Byte | 0      | 1 .. len-3      | len-2 . len-1  | len   |
     +--------+-----------------+----------------+-------+
Data | length | Modbus response | start register | count |
```

Timestamp

Bytes 0 to 4 will hold the timestamp when the execution of the entry started, the timestamp on which the entry's cron triggered. That might not be the exact time at which the commands of this uplink have been executed. The commands of the entry will be executed one ofter another, and communication over Modbus takes time. That time is much longer if the communication fails (as the Gateway waits for a timeout and executes retries for failed commands). The commands of this uplink could have been executed several seconds later than the timestamp. The messages are built like this on purpose. It allowes the backend to know, which uplinks are from the same execution of the same entry.

The timestamp is sent as a signed 40 bit integer containing the number of seconds passed since 1970-01-01T00:00:00 UTC (a UNIX timestamp).

Response

Each uplink will contain one or more responses. Each response starts with its length, so that the parser can find out, how many responses are in an uplink.

Length

Each uplink starts with an unsigned 8 bit integer that holds the length of the response (including the additional 3 bytes for start register and count). If there are multiple responses in an uplink, the length can be used to find out where they end. If a response has been split over multiple uplinks, this can be identified by the length as well.

Modbus response

The bytes sent as response by the Modbus slave device as they were sent on the wire, excluding the checksum. This will normally include the data you need, as well as the slave's address and the method used to read the data. Keep in mind, that in case of an error, the response will be shorter then on successful execution. Error responses are 3 bytes long and contain an error code as the third byte. If the error code is `11` (= `0x0B`), the error response is actually created by the Gateway. It is an error code for Modbus gateways to indicate that no response could be received. An error condition is indicated in the response by highest bit of the function byte. This is part of the Modbus spec.

Start register

The address of the first register/coil that was affected, encoded as an unsigned big endian 16 bit integer. This information is needed on error conditions to know which command was executed. For Modbus commands that do not have a starting register/coil (e.g. function 7, reading exeption status), the value of this field is undefined.

Count

The number of registers/coils that have been affected, encoded as an unsigned 8 bit integer. This information is sometimes needed to know which command was executed. For Modbus commands that do not have a register/coil value (e.g. funtion 5, forcing a single coil), this value is undefined. When a command is executed that affects more than 255 coils, this value is undefined.

Split uplinks (Port 5)

If a Modbus response does not fit into a single uplink, it will be split into multiple parts. The first part will be put into an uplink on port 3 as described above. There will be only a single response in an uplink if splitting happens. You can see by the length field, and the uplinks size, that the response has been split. There will follow as many uplinks as needed to fit the hole response sent on port 5. Use the length byte and the LoRaWAN frame counter to fit all the parts together. The payloads must all be appended and can than be handled as a single large uplink on port 3.

For responses to downlinks on port 4, the splitting process is the same, only that the first uplink will be sent on port 4 instead of port 3.

It is possible to write configurations that avoid splitting completely. This is often easier then handling split uploads.

Responsed to Downlinks (Port 4)

Uplinks triggered by downlinks on port 4 are always using the Verbose Format. They are sent on port 4 instead of port 3. The timestamp in port 4 uplinks holds the time when the downlink with the command was received.

Modbus Compact Format (Up, Ports 20 to 59)

When `PlFmt` is set to `4` or `5`, the Gateway will send Modbus uplinks in compact format. All Modbus commands in all entries in `MbCmd` will be processed in the order they appear. The Gateway will create an uplink format that holds all data that is read from any Modbus slaves. It will be a fixed format, where every byte from every register or coil will have its fixed position.

Look at the Log when you test the configuration for the compact format. The Gateway will output detailed information on the format it creates from the configuration. It will list register number and slave address for each byte. If there is a problem in the configuration and no format can be created, that will also be reported.

The fixed upload format created will start using port 20 and use as many ports needed (up to port 59) to fit all the bytes from your configuration. No message will be longer then the number of bytes set in `PlMax`. The bytes of the responses will simply be attached in the order they appear in the command definition. Whenever a response wont fit in the message, a new uplink definition on a new port will be startet. If a response is too long for the payload size set in `PlMax` then the configuration is invalid. Longer commands must be split into multiple shorter commands, so that the responses will fit. The order of responses will never be changed, even if a different order would fit into less different uplink formats. It is up to the person configuring the Industrial Gateway to find a configuration that uses this format efficiently. When there exist multiple entries in `MbCmd` each new entry will always start a new uplink format.

It is possible to build very efficient formats with this feature. You can group data points together that you need in a single uplink. If you need help with configuring the Gateway for your installation, please contact sales@lobaro.de for an offer.

Each uplink format will have a single header byte that holds an error flag and a format id (0-127) that you can set freely for each of your devices. `PlFmt = 4` will include a 5 byte timestamp in each uplink. `PlFmt = 5` will not. Appart from that, the formats work identically.

The payload in the compact format is influenced by the config parameters `MbCmd`, `PlFmt`, `PlMax`, and `PlId`.

Uplinks responding to downlinks on port 4 will always respond in the verbose format on port 4.

Structure of an uplink using PlFmt = 4

```
Byte | 0      | 1 . 2 . 3 . 4 . 5 | 6 ...      | ...        | ... | ...          |
     +--------+-------------------+------------+------------+-----+--------------+
Data | header | timestamp         | response 1 | response 2 | ... | response n |
```

Structure of an uplink using PlFmt = 5

```
Byte | 0      | 1 ...      | ...        | ... | ...        |
     +--------+------------+------------+-----+------------+
Data | header | response 1 | response 2 | ... | response n |
```

Header

Byte 0 of each uplink holds the header byte. The most significant bit will indicate if an error occurred during any of the commands that were executed to collect the data for this uplink. Be aware that this format will neither report the error code, nor will it explicitly state, which commands resulted in an error. The bytes comming from the failed modbus command will all be set to `0xff`. This can also be a legitimate value returned by a Modbus slave device.

The lower seven bits of the header byte contain the value of `PlId`. This can be useful if you have a large number of devices in the field where many share a configuration. You can create a parser for your formats that identifies a specific format by this ID. If you don't need that feature, just keep `PlId = 0`.

If you do not use `PlId`, the header byte will be `0x00` if all commands for that format were executed successfully. If there has an error on at least one of the commands, it will be `0x80`.

Timestamp

On `PlFmt = 4`, bytes 1 to 5 will hold the timestamp when the execution of the entry started. That is the timestamp on which the entry's cron triggered. That might not be the exact time at which the commands of this uplink have been executed. The commands of the entry will be executed one ofter another, and communication over Modbus takes time. That time is much longer if the communication fails (as the Gateway waits for a timeout and executes retries for failed commands). The commands of this uplink could have been executed several seconds later than the timestamp. The messages are built like this on purpose. It allowes the backend to know, which uplinks are from the same execution of the same entry.

The timestamp is sent as a signed 40 bit integer containing the number of seconds passed since 1970-01-01T00:00:00 UTC (a UNIX timestamp).

Wireless M-Bus (Up, Port 102)

The payload format for uploading wireless M-Bus telegrams is the same that is used by Lobaro's Wireless M-Bus when set to PayloadFormat 2. Telegrams are uploaded on port 102 with some metadata prefixing it. Because wM-Bus telegrams are often longer than the available payload size in LoRaWAN, telegrams can be split over several uplinks. The first byte in each uplink on port 102 indicates if that uplink is part of a split message, and if there are uplinks comming before and/or after it. With this information and the frame number of the LoRaWAN uplink, split messages can be put back together again. The payload will always use as many bytes as possible for the current spreading factor used.

If you are using the Lobaro Platform for integrating your devices, it will take care of putting the parts back together. It can also decrypt encrypted wM-Bus telegrams if you provide the keys to the platform. It also has a parser for M-Bus telegrams, that will present the contained data in a JSON-Format that is easier to read than a raw M-Bus telegram.

Structure of a message on port 102

```
Start of a wireless M-Bus new telegram uploaded:
```

```
Byte | 0    | 1 . 2 . 3 . 4 . 5 | 6    | 7 ...                    |
     +------+-------------------+------+--------------------------+
Data | part | timestamp         | rssi | raw telegram (beginning) |

Continuation of a wireless M-Bus telegram upload:

Byte | 0    | 1 ...                    |
     +------+--------------------------+
Data | part | raw telegram (continued) |
```

Part byte

The byte at position 0 (part) indicates if the uplink containes the first and/or final part of a telegram:

Part byte	Meaning
`0x03`	This uplink contains both, the first and final part of a telegram. Bytes 1 to 6 contain the timestamp of reception and the RSSI. The bytes from position 7 to the end of this uplink are the complete telegram.
`0x02`	This uplink contains the final part of the telegram, but not its start. There has been at least one uplink before this for the telegram. The telegram bytes start from position 1; there is no meta data in this uplink.
`0x01`	This uplink contains the first part of the telegram, but not the final part. There will be at least one more uplink containing bytes of the telegram. Bytes 1 to 6 contain the timestamp of reception and the RSSI. The bytes from position 7 to the end of this uplink contain the first part of the telegram.
`0x00`	This uplink contains neither the first nor the final part of a telegram. There has been at least one uplink before this, and there will be at least one uplink after this, for the current telegram. There is no meta data in this uplink. The bytes from position 1 to the end of the uplink contain a part of the raw telegram.

Timestamp

The bytes 1 to 5 contain the time of reception of the w-MBus telegram, measured by the internal clock of the Industrial Gateway. It is sent as a signed 40 bit number containing the number of seconds passed since 1970-01-01T00:00:00 UTC (a UNIX timestamp). The timestamp is only included in uplinks that contain the first part of a telegram (Part byte = `0x03` or Part byte = `0x01`).

RSSI

The byte at position 6 holds the reception quality of the wireless M-Bus telegram, as it was received by the Industrial Gateway. This can be useful as an indicator of how far away a received meter is from the Gateway. It also gives you an estimation of how likely it is that the reception of the meter's telegram by the Gateway will be disrupted by radio disturbances. The byte contains an unsigned integer that holds the negative RSSI measured on reception in dBm. So a value of `0x31` means an RSSI of `-49 dBm`. The RSSI is only included in uplinks that contain the first part of a telegram (Part byte = `0x03` or Part byte = `0x01`).

Raw Telegram

The remaining payload of an uplink is used to send as many bytes of the telegram as possible. If a telegram is split, it must be reattached in your backend (or in the Lobaro Platform). Use the part byte and the LoRaWAN frame counter to fit the parts together. If there are skips in the frame counter, than part of the telegram has been lost. If an uplink holds the beginning of a telegram (Part byte = `0x03` or Part byte = `0x01`), the telegram's data starts at byte 7. For continued uplinks (Part byte = `0x02` or Part byte = `0x00`), te telegram's data starts at byte 1.

Wired M-Bus (Up, Ports 110 and 112)

The payload format used to upload wired M-Bus telegrams is similiar to the format used for wireless M-Bus telegrams. Because M-Bus telegrams are often longer than the available payload size in LoRaWAN, telegrams can be split ofer several uplinks. The first byte in each uplink on port 112 indicates if that uplink is part of a split message, and if there are uplinks comming before and/or after it. With this information and the frame number of the LoRaWAN uplink, split messages can be put back together again. The payload will always use as many bytes as possible for the current spreading factor used.

To use wired M-Bus with the Industrial Gateway, you will need a M-Bus converter between the Gateway and your slave devices. Lobaro provides a combined solution under order number 8000200, that allows to attach up to 20 wired M-Bus devices.

If you are using the discovery feature, the device will scan for attached M-Bus slaves after booting. It will upload the result of the scanning process on port 110. There will be discovery uplink for each entry in `MbusCmd` that has `*` as the device list. If there are more devices discovered than fit in a LoRaWAN uplink, the information is split over mutliple uplinks.

If you are using the Lobaro Platform for integrating your devices, it will take care of putting the parts back together. It also has a parser for M-Bus telegrams, that will present the contained data in a JSON-Format that is easier to read than a raw M-Bus telegram.

Structure of a message on port 110

```
Byte | 0 . 1 . 2 . 3 | 4            | 5              |
     +---------------+--------------+----------------+
Data | Baud rate     | slaves found | addresses sent |
```

Baud rate

The Baud rate used for the discovery, encoded as an unsigned big endian 32 bit integer.

Slaves found

The number of slaves fond during scanning, encoded as an unsigned 8 bit integer.

Addresses sent

Number of addresses already sent in uplinks before this one, encoded as an unsigned 8 bit integer. This will only have a value > 0 when there have been more slaves discovered than can be fit in a single uplink. For this to happen, you will need to use a different M-Bus converter than the one normally provided by Lobaro (the Relay M-Bus PW20L/RS485), as that one only supports up to 20 slave devices.

Structure of a message on port 112

```
Start of a new wired M-Bus telegram uploaded:

Byte | 0    | 1 . 2 . 3 . 4 . 5 | 6         | 7 . 8 . 9 | 10 ...                   |
     +------+-------------------+-----------+-----------+--------------------------+
Data | part | timestamp         | address   | request   | raw telegram (beginning) |

Continuation of a wired M-Bus telegram upload:

Byte | 0    | 1 ...                     |
     +------+---------------------------+
Data | part | raw telegram (continued)  |
```

Part byte

The byte at position 0 (part) indicates if the uplink containes the first and/or final part of a telegram:

Part byte	Meaning
`0x03`	This uplink contains both, the first and final part of a telegram. Bytes 1 to 9 contain meta data. The bytes from position 10 to the end of this uplink are the complete telegram.

0x02	This uplink contains the final part of the telegram, but not its start. There has been at least one uplink before this for the telegram. The telegram bytes start from position 1; there is no meta data in this uplink.
0x01	This uplink contains the first part of the telegram, but not the final part. There will be at least one more uplink containing bytes of the telegram. Bytes 1 to 9 contain metadata. The bytes from position 10 to the end of this uplink contain the first part of the telegram.
0x00	This uplink contains neither the first nor the final part of a telegram. There has been at least one uplink before this, and there will be at least one uplink after this, for the current telegram. There is no meta data in this uplink. The bytes from position 1 to the end of the uplink contain a part of the raw telegram.

Timestamp

The bytes 1 to 5 contain the time of reception of the w-MBus telegram, measured by the internal clock of the Industrial Gateway. It is sent as an signed 40 bit number containing the number of seconds passed since 1970-01-01T00:00:00 UTC (a UNIX timestamp). The timestamp is only included in uplinks that contain the first part of a telegram (Part byte = `0x03` or Part byte = `0x01`).

Address

Byte 6 holds the (primary) M-Bus address of the slave this telegram was received from. It is encoded as an unsigned 8 bit integer. The address is only included in uplinks that contain the first part of a telegram (Part byte = `0x03` or Part byte = `0x01`).

Request

Bytes 7 to 9 contain the request this telegram is a response to, encoded as three ascii characters. This will normally be `"UD2"`. The request is only included in uplinks that contain the first part of a telegram (Part byte = `0x03` or Part byte = `0x01`).

Raw Telegram

The remaining payload of an uplink is used to send as many bytes of the telegram as possible. If a telegram is split, it must be reattached in your backend (or in the Lobaro Platform). Use the part byte and the LoRaWAN frame counter to fit the parts together. If there are skips in the frame counter, than part of the telegram has been lost. If an uplink holds the beginning of a telegram (Part byte = `0x03` or Part byte = `0x01`), the telegram's data starts at byte 10. For continued uplinks (Part byte = `0x02` or Part byte = `0x00`), te telegram's data starts at byte 1.

Remote Configuration (Down & Up, Ports 128 to 131)

The Industrial Gateway supports remote configuration by downlinks on port 128. It can be used to read and write config parameters and execute commands, e.g. a reboot. Responses to these downlinks are sent on port 128. If a response is too long for a single uplink, it will be split using the ports 129 - 131.

For more information, see LoRaWAN Downlink Config.

On devices running a firmware version <= 0.5.1, the config changed by LoRaWAN downlinks will not be persisted in the device. After a reboot, the original configuration will be restored.

LoRaWAN reference Decoder

This is a decoder written in JavaScript that can be used to parse the device's LoRaWAN messages. It can be used as is in The Things Network.

› Klicken Sie hier, um zu erweitern...

```
// Version 0.0.2
// Changelog
```

```
//
// 0.0.2 - 2024-09-24
// - Added support for Coils and do not return NaN in some cases.

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

function int40_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 int16_BE(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    return bytes[0] << 8 | bytes[1] << 0;
}

function uint16_BE(bytes, idx) {
    bytes = bytes.slice(idx || 0);
    return bytes[0] << 8 | bytes[1] << 0;
}

function port1(bytes) {
    return {
        "port":1,
        "version":readVersion(bytes),
        "flags":bytes[3],
        "temp": int16_BE(bytes, 4) / 10,
        "vBat": int16_BE(bytes, 6) / 1000,
        "timestamp": int40_BE(bytes, 8),
        "operationMode": bytes[13],
        "noData": !!(bytes[3] & 0x01)
    };
}

function port2(bytes) {
    var regs = [];
    if (bytes.length > 5) {
        // loop through data packs
        var b = bytes.slice(5);
        while (b.length>=4) {
            var r = {
                "device":b[0],
                "register":int16_BE(b, 1),
                "count":b[3] & 0x3f,
                "error":!!(b[3]>>7),
                "data":null
            };
            var dataLen = r["count"]*2;
            if (b.length >= dataLen+4) {
                r["data"] = b.slice(4, 4 + dataLen);
            }
            regs.push(r);
            b = b.slice(4+dataLen);
        }
    }
    return {
        "port":2,
        "timestamp": int40_BE(bytes, 0),
        "registers": regs
    };
}

function modbusErrorString(code) {
    // Modbus exception codes
    // see https://en.wikipedia.org/wiki/Modbus#Exception_responses
    switch (code) {
        case 1:
            return "Illegal Function";
        case 2:
            return "Illegal Data Address";
        case 3:
            return "Illegal Data Value";
        case 4:
            return "Slave Device Failure";
        case 5:
            return "Acknowledge";
        case 6:
            return "Slave Device Busy";
        case 7:
            return "Negative Acknowledge";
        case 8:
            return "Memory Parity Error";
        case 10:
            return "Gateway Path Unavailable";
        case 11:
            return "Gateway Target Device Failed to Respond";
        default:
            return "Unknown error code";
    }
}

function parseModbusPayloadRegisters(payload) {
    if (payload.length < 1) {
        return null;
    }
    var byteCnt = payload[0];
    if (payload.length !== byteCnt + 1) {
        return null;
    }
    var fun = payload[1] & 0xf;
    var vals = [];

    if (fun == 0x01) {
      // Coils
      for (var i=0; i<byteCnt; i++) {
        vals.push(+payload[i+1])
      }
    } else {
      // 2 Byte Registers
      for (var i=0; i<byteCnt; i+=2) {
        vals.push([+payload[i+1], +payload[i+2]])
      }
    }

    return vals;

}
function parseModbusResponse(raw) {
    var resp = {};
    if (raw.length >= 6) {
        var fun = raw[1] & 0xf;
        var error = !!(raw[1] & 0x80);
        var rawResp = raw.slice(0, raw.length - 3);
        resp["slave"] = raw[0];
        resp["function"] = fun;
        resp["error"] = error;
        resp["start"] = uint16_BE(raw, raw.length - 3);
        resp["cnt"] = raw[raw.length - 1];
        resp["raw"] = rawResp;
        if (error) {
            resp["errorCode"] = raw[2];
            resp["errorText"] = modbusErrorString(raw[2]);
        } else {
           resp["values"] = parseModbusPayloadRegisters(rawResp.slice(2))
        }
    }
    return resp;
}

function FullResponses(bytes, port) {
    var timestamp = int40_BE(bytes);
    var pos = 5;
    var resps = [];
    while (pos < bytes.length) {
        var respLen = bytes[pos++];
        if (bytes.length >= pos + respLen) {
            var rawResponse = bytes.slice(pos, pos + respLen);
            resps.push(parseModbusResponse(rawResponse));
            pos += respLen;
        } else {
            break;
        }
    }
    return {
        "port": port,
        "timestamp" : timestamp,
        "responses": resps
    };
}

function bin2String(array) {
    var result = "";
    for (var i = 0; i < array.length; i++) {
        result += String.fromCharCode(array[i]);
    }
    return result;
}

function ConfigResponse(data) {
    var t = bin2String(data);
    return {
        "response" : t,
        "error" : (t.length === 0) || (t[0] === '!')
    }
}

/**
 * TTN decoder function.
 */
function Decoder(bytes, port) {
    switch (port) {
        case 1:
            // Status message:
            return port1(bytes);
        case 2:
            // not legacy format:
            return port2(bytes);
        case 3:
        case 4:
            // v1.0.0 format, full modbus responses:
            return FullResponses(bytes, port);
        case 5:
            // continuation of previous response:
            return {};
        case 6:
            // dense format with prefixed timestamp:
            return {};
        case 7:
            // dense format without timestamp:
            return {};
        case 128:
            return ConfigResponse(bytes);
    }
    return {"error":"invalid port", "port":port};
}
```

```
/**
 * LoRaServer decoder function.
 */
function Decode(fPort, bytes) {
    // wrap TTN Decoder:
    return Decoder(bytes, fPort);
}
function Parse(input) {
    var data = bytes(atob(input.data));
    var port = input.fPort;
    var fcnt = input.fCnt;
    var vals =  Decoder(data, port);
    vals["port"] = port;
    vals["data"] = data;
    vals["fnct"] = fcnt;
    var lastFcnt = Device.getProperty("lastFcnt");
    vals["reset"] = fcnt <= lastFcnt;
    Device.setProperty("lastFcnt", fcnt);
    return vals;
}
```

LoRaWAN Examples

This chapter illustrates with some examples, how working with the Industrial Gateway looks like. The bytes that are sent via LoRaWAN are presented here as hex strings, while on the air they are sent as raw bytes. Modbus Commands and Responses are broken down to their parts in the explanations, but explaining the format used by Modbus in detail is beyond the scope of this manual. You can find a short explanation on Modbus on Wikipedia: https://en.wikipedia.org/wiki/Modbus.

› The Examples section contains an illustration of a split up Response

Uplinks Modbus in Verbose Payload Format (PlFmt=1)

The following shows some examples of configuration for the automated reading and what the generated Uplinks for that could look like.

Example A1: Read Holding Registers 0, 1, and 2 of device with address 1

Verbose Payload Format (Port 3, PlFmt=1)

```
MbCmd = '010300000003'

# Example resulting Uplink after successful readout
Up, Port 3: '005d1698fd0c0103061234567890ab000003'
 '005d1698fd' -> timestamp = 1561762045 -> 2019-06-28T22:47:25 UTC
 '0c'        -> first Response is 12 bytes long
 '0103061234567890ab000003' 12 bytes modbus response:
   '01' -> slave device with address 1
   '03' -> function 3 = read Holding Register, success
   '06' -> 6 bytes of data in Response following
   '1234567890ab' -> 6 bytes of data
   '0000' -> start reading at register 0
   '03' -> read 3 consecutive registers

# Example resulting Uplink after failing readout
Up, Port 3: '005d1698fd0601830b000003'
 '005d1698fd' -> timestamp = 1561762045 -> 2019-06-28T22:47:25 UTC
 '06'        -> first Response is 6 bytes long
 '01830b000003' 3 bytes modbus response:
   '01' -> slave device with address 1
   '83' -> function 3 with error indicator 80 = read Holding Register, failed
   '0b' -> error code 11: "Gateway Target Device Failed to Respond"
   '0000' -> start reading at register 0
   '03' -> read 3 consecutive registers
```

Example A2: Read coils 1000-1019 of device 32

Verbose Payload Format (Port 3, PlFmt=1)

```
MbCmd = '200103e80014'
```

```
# Example resulting Uplink
Up, Port 3: '005d1698fd 09 200103f1041a03e814'
 '005d1698fd' -> timestamp = 1561762045 -> 2019-06-28T22:47:25 UTC
 '09' -> first Response is 9 bytes long
 '200103f1041a03e814' 9 bytes of response:
  '20' -> slave device with address 32
  '01' -> read coils, success
  '03' -> 3 bytes of data
  'f1041a' -> 20 bits of data packed into 3 bytes
  '03e8' -> start reading at coil 1000
  '14' -> read 20 consecutive coils
```

Example A3: Read registers from two devices

Verbose Payload Format (Port 3, PlFmt=1)

```
MbCmd = '0a0300010005,3001ea600020'
```

```
# Example resulting Uplink
Up, Port 3: '005d1698fd100a030a11112222333344445555000105 0a30010412345678ea6020'
 '005d1698fd' -> timestamp = 1561762045 -> 2019-06-28T22:47:25 UTC
 '10' -> first Response is 16 bytes long
 '0a030a11112222333344445555000105' 16 bytes of Response
  '0a' -> slave device with address 10
  '03' -> read Holding Registers, success
  '0a' -> 10 bytes of data following
  '11112222333344445555' 10 bytes of data
  '0001' -> start reading at register 1
  '05' -> read 5 registers
 '0a' -> second Response is 10 bytes long
 '30010412345678ea6020' 10 bytes of Response
  '30' -> slave device with address 48
  '01' -> read Coils, success
  '04' -> 4 bytes of data following
  '12345678' -> 32 bits of data packed in 4 bytes
  'ea60' -> start at coil 60000
  '20' -> read 32 coils
```

Example A4: Split uplink message

Verbose Payload Format (Port 3, PlFmt=1)

```
MbCmd = '010300010020'
# Command reads 32 consecutive registers resulting in 64 bytes payload
```

```
# Example resulting Uplinks for a Spreading Factor of 12 with 51 bytes of payload per
message
Up 1, Port 3:
'005d1698fd46010340000100020003000400050006000700080009000a000b000c000d000e000f0010001100
12001300140015'
  '005d1698fd' -> timestamp = 1561762045 -> 2019-06-28T22:47:25 UTC
  '46' -> first Response is 70 bytes long since the remainder of the message does not
contain 70 bytes,
  you know there must be an additional part coming
Up 2, Port 5: '0016001700180019001a001b001c001d001e001f00200120'
  This contains the rest of the message. Appended to the previous message, it adds up to
the correct number of bytes.
```

Uplinks triggered by Downlink Commands

Example B1: Read single Input Register by Downlink

Verbose Payload Format (Port 4, Downlink Response)

```
Down, Port 4: '06180401000001'
 '06' -> first Command is 6 bytes long
 '180401000001' 6 bytes of Modbus Command
  '18' -> slave device with address 24
  '04' -> function 4, read Input Register
  '0100' -> start at register 256 '0001' -> read 1 register

# Example resulting Uplink
Up, Port4: '004b3dd67508180402abcd010001'
 '004b3dd675' -> timestamp = 1262343797 -> 2010-01-01T11:03:17 UTC
 '08' -> first Response is 8 bytes long
 '180404abcd010001' 8 bytes of Response
  '18' -> slave device with address 24
  '04' -> read Input Register, success
  '02' -> 2 bytes of data following
  'abcd' -> 2 bytes of data
  '0100' -> start at register 256
  '01' -> read 1 register
```

Example B2: Writing holding registers on multiple devices

Verbose Payload Format (Port 4, Downlink Response)

```
Down, Port 4: '06a106aabb12340fa210a00100040811223344556677888'
 '06' -> first Command is 6 bytes long
 'a106aabb1234' 6 bytes of Modbus Command
  'a1' -> slave device with address 161
  '06' -> function 6, write single Holding Register
  'aabb' -> address of Register to write = 43707
  '1234' -> two bytes of data
 '0f' -> second Command is 15 bytes long
 'a210a0010004081122334455667788' 15 byte of Modbus Command
  'a2' -> slave device with address 162
  '10' -> function 16, write multiple Holding Registers
  'a001' -> start at register 40961
  '0004' -> 4 registers to write
  '08' -> 8 bytes of data follow
  '1122334455667788' -> 8 bytes of data

# Example resulting Uplink Up, Port 4: '004b3dd67506a1860200000006a210a0010004'
 '004b3dd675' -> timestamp = 1262343797 -> 2010-01-01T11:03:17 UTC
 '06' -> first Response is 3 bytes long
 'a18602000000' 3 bytes of Modbus Response
  'a1' -> slave device address 161 '86' -> write single Holding Regsiter, failed
  '02' -> error code 2: "Illegal Data Address"
  '0000' -> start register not used (undefined)
  '00' -> count not used (undefined)
 '06' - second Response is 6 bytes long
 'a210a0010004' 6 bytes of Modbus Response
  'a2' -> slave device address 162
  '10' -> write multiple Holding Registers, success
  'a001' -> start at register 40961
  '0004' -> 4 registers to write
```

Uplinks in Compact Payload Format

Example C1: Single Modbus Command, PlFmt 4

```
# Attached device: B+G E-Tech power Meter

# Config:
MbCmd = 010300000003  -> Read registers 0 to 3 from Slave 1
PlFmt = 4             -> Compact Format with Timestamp
PlMax = 51            -> Max 51 Bytes per Uplink
PlId  = 0             -> Payload Id = 0

# Info from Log
APP| Number of commands to be executed on cron: 1
APP|   01 03 00 00 00 03
APP| Compact format definition, id=0, max size=51
APP|   Port 20:
APP|     000-000=error&fmt-id
APP|     001-005=timestamp
APP|     006-011=resp(010300000003)

# Successful readout, leading to Uplink on Port 20: '00005fd8bf08000000010033'
  '00' -> '0' no error, '00' -> PlId = 0
  '005fd8bf08' -> timestamp = 1608040200 -> 2020-12-15T13:50:00 UTC
  '000000010033' -> 6 Bytes data from 3 Registers -> 3.07 kWh

# Failed readout, leading to Uplink on Port 20: '80005fd8c7caffffffffffff'
  '80' -> '1' error in any of the responses, '00' -> PlId = 0
  '005fd8c7ca' -> timestamp = 1608042442 -> 2020-12-15T14:27:22 UTC
  'ffffffffffff' -> 6 Bytes set to 0xff, indicating error
```

Example C2: Multiple Commands, PlFmt 5

```
# Attached devices: Multiple B+G E-Tech power Meters

# Config:
MbCmd = 010300000003,020300000003 -> Read registers 0 to 3 from Slave 1
PlFmt = 5                         -> Compact Format with Timestamp
PlMax = 51                        -> Max 51 Bytes per Uplink
PlId  = 13                        -> Payload Id = 13

# Info from Log
APP| Number of commands to be executed on cron: 2
APP|   01 03 00 00 00 03
APP|   02 03 00 00 00 03
APP| Compact format definition, id=29, max size=51
APP|   Port 20:
APP|     000-000=error&fmt-id
APP|     001-006=resp(010300000003)
APP|     007-012=resp(020300000003)

# Successful readout of both, leading to Uplink on Port 20:
'1d0000000100330000001a0040'
  '1d' -> '0' no error, '1d' -> PlId = 13
  '000000010033' -> 6 Bytes data from 3 Registers, Meter 1 ->  3.07 kWh
  '0000001a0040' -> 6 Bytes data from 3 Registers, Meter 2 -> 67.20 kWh

# Partly successful readout, leading to Uplink on Port 20:
'9d000000010033ffffffffffff'
  '9d' -> '1' error (in any of the values in message), '1d' -> PlId = 13
  '000000010033' -> 6 Bytes data from 3 Registers, Meter 1 -> 3.07 kWh
  'ffffffffffff' -> 6 Bytes set to 0xff to indicate error, Meter 2
```

```
# Failed readout, leading to Uplink on Port 20: '9dffffffffffffffffffffffff'
  '9d' -> '1' error (in any of the values in message), '1d' -> PlId = 13
  'ffffffffffff' -> 6 Bytes set to 0xff to indicate error, Meter 1
  'ffffffffffff' -> 6 Bytes set to 0xff to indicate error, Meter 2
```

Example C3: Multimple Commands, leading in multiple Upoads, PlFmt 5

```
# Config
MbCmd = 010300000010,010301000004,0103020a000c,010300800008
PlFmt = 5                          -> Compact Format with Timestamp
PlMax = 40                         -> Max 40 Bytes per Uplink
PlId  = 10                         -> Payload Id = 10

# Info from Log
APP| Number of commands to be executed on cron: 4
APP|   01 03 00 00 00 10
APP|   01 03 01 00 00 04
APP|   01 03 02 0A 00 0C
APP|   01 03 00 80 00 08
APP| Compact format definition, id=10, max size=40
APP|   Port 20:
APP|     000-000=error&fmt-id
APP|     001-032=resp(010300000010)
APP|   Port 21:
APP|     000-000=error&fmt-id
APP|     001-008=resp(010301000004)
APP|     009-032=resp(0103020a000c)
APP|   Port 22:
APP|     000-000=error&fmt-id
APP|     001-016=resp(010300800008)

# Resulting in 3 consecutive uploads on Port 20-22:
# Port 20: 0a000000010002000300040005000600070008000900 0a000b000c000d000e000f
  '0a' -> '0' no error, '0a' -> PlId 10
  '000000010002000300040005000600070008000900 0a000b000c000d000e000f' -> 32 Bytes of
Data from Registers 0x0000-0x000f
# Port 21: 8affffffffffffffff111122223333444455556666777788889999aaaabbbbcccc
  '8a' -> '1' error occured, '0a' -> PlId 10
  'ffffffffffffffff' -> 8 Bytes set to 0xff indicating error reading Registers 0x0100-
0x0103
  '111122223333444455556666777788889999aaaabbbbcccc' -> 24 Bytes of Data from Registers
0x020a-0x0215
# Port 22: 0a01010202030304040505060607070808
  '0a' -> '0' no error, '0a' -> PlId 10
  '01010202030304040505060607070808' -> 16 Bytes of Data from Registers 0x0080-0x0087
```

Mobile data consumption

Uploading one Uplink with 400 bytes including all metadata (might be less, depending on the configuration).

Telegram upload interval	Monthly NB-IoT data usage
1 each Day	~12 kB
8 each Day (every 3h)	~100 kB
400 each Week	~700 kB
250 each Day	~3 MB

All calculations are estimations and might vary depending on the configuration.

CE Declaration of Conformity

CE-LOB-GW-DINRAIL-HYB_11_01_22.pdf

Keine Stichwörter