duurzaam
energie
perspectief

White Paper Firmware Upload over Modbus

Alliander Standard for Modbus Devices

DEP - Duurzaam Energie Perspectief
Authors: Patrick Vermeulen
Rob Kopmeiners

Summary

The Modbus standard (modbus.org) is often used for interfacing with digital equipment in the primary and secondary substations of European electricity grids. Alliander uses the Modbus RTU serial interface as a standard for digitization of secondary substations. There is one feature though that is lacking for large scale deployment and that is a standardized method for firmware upgrade over Modbus RTU of the digital equipment.

In this white paper a general concept for a practical implementation of firmware upgrade over Modbus RTU is described. The functioning of this method is proven in an operational district heating network in the Netherlands.

Contents

1 Introduction

Alliander controls and supervises its grid with SCADA, using digital equipment for information and control. Currently more and more digital equipment is introduced in the medium-voltage grid as part of Alliander's strategy for optimized utilization of the grid. The digital equipment that is commercially available and used in the secondary substations, has in many cases a Modbus RTU serial interface to connect with the local controller. The local controller communicates to Alliander's backend or SCADA systems via a mobile communication network.

One important aspect for managing digital equipment at scale is firmware upgrade. Currently, such a feature is not generally available for the digital equipment with a Modbus RTU interface by lack of a standard.

This white paper proposes a general concept that makes it feasible to implement a method for firmware upgrade in all digital equipment with a Modbus RTU serial interface. The method is actually in use in a operational district heating network. The control of the firmware process is inspired by the Dutch smart meter standard, eg. DSMR 4.3 [1], that is used extensively in the Netherlands. A recent paper shows the feasibility of file transfer with the Modbus protocol, although the Ethernet version is used (Modbus TCP) [2].

The concept for firmware upgrade over Modbus RTU in this white paper is the *de facto* Alliander standard and freely available for all manufacturers to implement and to use. An open source strategy is considered, enabling collaborating and verifiable code by the community.

Section 2 of this document describes the general concept of the firmware upgrade and Section 3 the details. The experience in a district heating network with this method is elaborated in Section 4.

2 Concept of Firmware Upgrade

The case described here is a local central device, named a gateway, that wants to upgrade the firmware of a connected Modbus device. The firmware upgrade process uses the basic Modbus RTU master/slave communication over a serial RS-485 interface. The gateway is the Modbus master and the Modbus device is the slave. The firmware of the Modbus device is a binary file, created by the manufacturer, that is available at the gateway. The gateway needs to deliver the binary file to the Modbus device. After receiving the binary file, the Modbus device will subsequent verify and activate the new firmware. Figure 1 shows the concept in an UML diagram.

The Modbus protocol is register-based, and a file transfer by the Modbus protocol is actually a data transfer by writing to the holding registers. Since a binary firmware file can be quite large, the transfer of the file will be done in cycles, block by block. Each block is a chunk of the file, limited by the maximum 246 bytes payload of a Modbus frame. The gateway will write blocks of data to the registers and the Modbus device will assemble them to create the original binary file.

The whole firmware upgrade process is controlled via the Firmware Upgrade Control Record and the Firmware Upgrade Status Record to which the gateway writes control commands and the Modbus device reports its status. Notice that since the Modbus is a master/slave protocol, the gateway has to request the Modbus device for its status, hence reading out the Firmware Upgrade Status Register at the Modbus device.

Data integrity of the binary file is crucial and is verified at two levels. The Modbus protocol provides a block writing command with function code 16 (hex: 0x10) with an integrity check by means of a 2 bytes CRC per frame. So each frame is checked. For the Modbus device a final integrity check on the whole binary file is highly recommended and may also include verification on authenticity by means of a digital signature. The implementation of this integrity feature is left to the manufacturers of the Modbus device.

In this concept it assumed that an extended validation check of the transferred binary file – including the mentioned integrity check – is performed in the Modbus device. After validation, the Modbus device will activate the new firmware, in most cases requiring an automatic reboot of the device. Communication with the gateway will be temporary lost. After reestablishing the Modbus connection, the gateway will verify the successful firmware activation, for instance by reading the status of the firmware upgrade process and the new firmware version of the Modbus device.

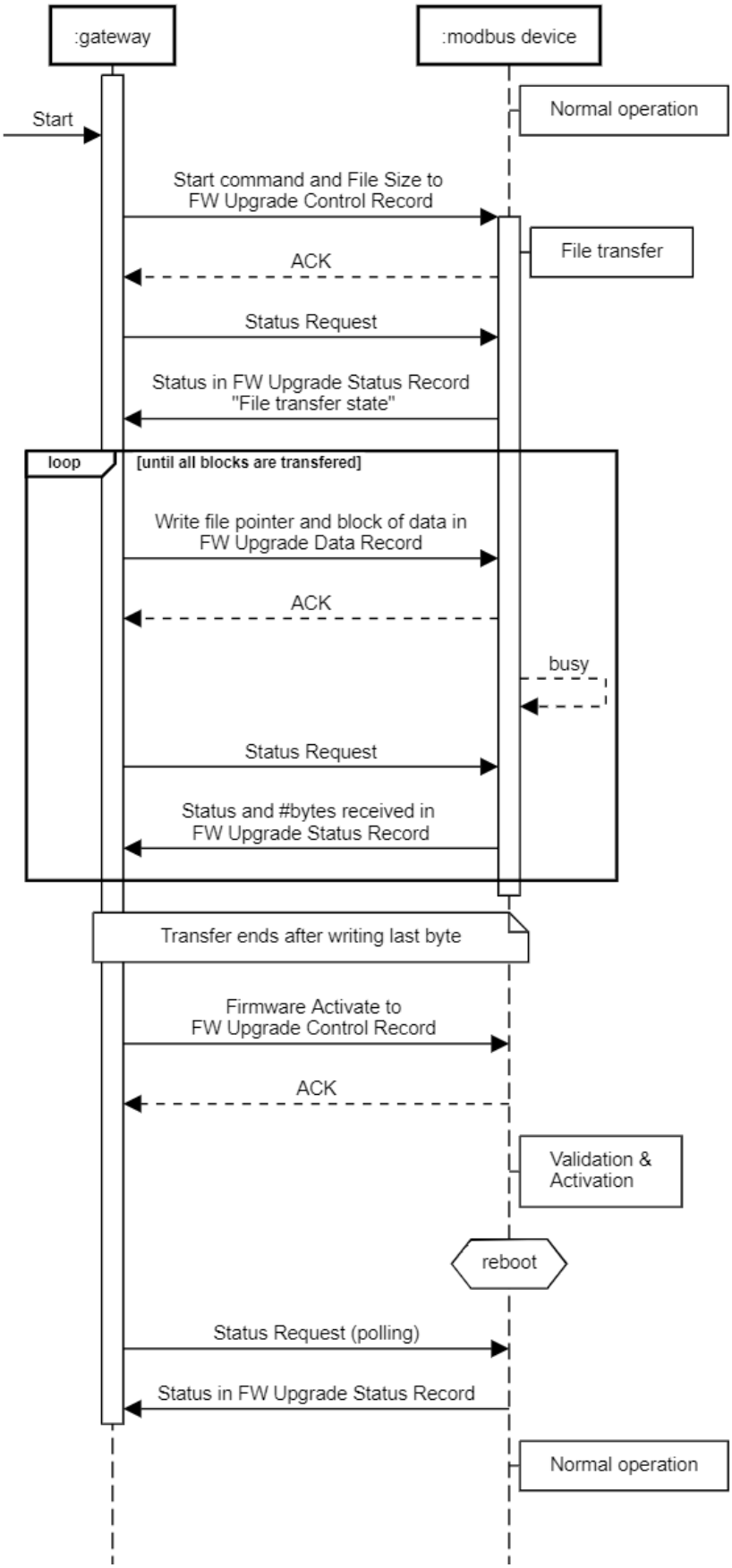


Figure 1: Concept of the firmware upgrade process over Modbus.

3 Bits and Bytes

This concept uses general Modbus ADU's (Application Date Unit) for serial lines [3] with PDU's (Protocol Data Unit) using the two function codes 03 (0x03) for reading multiple holding registers and 16 (0x10) for writing multiple holding registers [4]. In the Modbus specifications [3] and [4] a Modbus frame for serial lines is defined as presented in Figure 2 and is equal for master and slave.

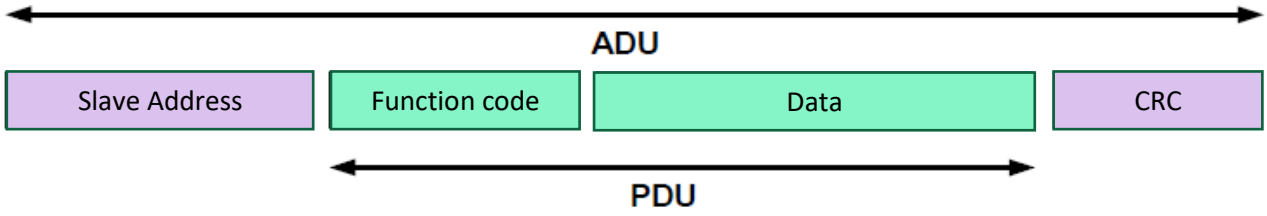


Figure 2: Modbus frame over a serial line

3.1 Firmware Upgrade Control Record

The gateway controls the process at the Modbus device with the Firmware Upgrade Control Record. The Firmware Upgrade Control Record consists of writing to three Modbus holding registers at the Modbus device starting at the predefined address <fw_ctl_writing_address>. It has the structure shown below.

Register Address	High byte	Low byte
<fw_ctl_writing_address>	Control code (write)	0x00 or Block size in bytes (write)
<fw_ctl_writing_address>+1	File size in number of bytes MSB (16 bits)	
<fw_ctl_writing_address>+2	File size in number of bytes LSB (16 bits)	

Note that sending a command to Firmware Upgrade Control Register is actually writing to the three registers with function code 16 (0x10) starting at address <fw_ctl_writing_address>. In this concept the default address is 0x4200, but could be configured, depending on the capabilities of the Modbus device.

3.2 Firmware Upgrade Status Record

The gateway verifies the status of the Modbus device with the Firmware Upgrade Status Record, reading three Modbus holding registers at the Modbus device starting at the predefined address <fw_ctl_reading_address>. It has the structure shown below.

Register Address	High byte	Low byte
<fw_ctl_reading_address>	FW upgrade state (read)	Error code (read)
<fw_ctl_reading_address>+1	Received number of bytes MSB (16 bits)	
<fw_ctl_reading_address>+2	Received number of bytes LSB (16 bits)	

Requesting the status via Firmware Upgrade Control Register is actually requesting a read-out of three registers with function code 3 (0x03) at address <fw_ctl_reading_address>. In this concept the default address is 0x4210, but could be configured, depending on the capabilities of the Modbus device.

Sidenote: Register Numbers and Register Addresses

Many publications on Modbus discuss alignment of Modbus addresses. The confusion is already created in the Modbus standard: “Registers are addressed starting at zero. Therefore register numbered 1 is addressed as 0.”

So, while Modbus registers count from 1 to N per category (Discrete inputs, Coils, Input registers, Holding registers), the addressing of registers in the PDU counts from 0 to N-1.

How exactly Modbus registers are mapped on internal memory is vendor specific and irrelevant for the concept in this white paper. In this paper Modbus registers are only referred to in addresses and not in register numbers. Hence in this white paper the addressing starts from 0.

3.3 Control codes

The following table shows the control codes that are available, but are not always needed. In general, the implemented process could be quite simple and once started the process forwards automatically step by step. If finer control of the process is needed, the other control codes can be used.

Control code	Meaning
0	Firmware upgrade start
1	Firmware upgrade send data
2	*Reserved*
3	*Reserved*
4	Firmware upgrade validate
5	*Reserved*
6	*Reserved*
7	Firmware upgrade activate
8	*Reserved*
9	*Reserved*
10	Firmware upgrade cancel

The default process starts with control code 0. So writing for instance 0x0000 and the file size in number of bytes as 32-bits unsigned integer to the address space starting at <fw_ctl_writing_address> (default: 0x4200), will start off the firmware upgrade process.

3.4 Firmware upgrade states

The firmware upgrade process in the Modbus device steps through several states. The gateway will frequently request the status during the firmware upgrade process. The state of the firmware process and the accompanying error code is the status. The state codes are listed in the following table.

State code	Meaning
0	IDLE state
1	DATA RECEIVE state
2	DATA RECEIVED state
3	DATA RECEIVING failed state
4	VALIDATING state
5	VALIDATED state
6	VALIDATION failed state
7	ACTIVATING state
8	ACTIVATED state
9	ACTIVATION failed state

Figure 3 shows in detail how the Modbus device steps through the state of the firmware upgrade process. Certain states may be very short, being combined or non-existent. For instance, after a very short (automatic) validation at the end of the DATA RECEIVE state, the device may reboot and return in ACTIVATED state or IDLE state (normal operation). The best strategy will be vendor specific.

3.5 Error codes

While the gateway frequently requests the status during the firmware upgrade process, it expects the state of the firmware process and the accompanying error code. Normally, if everything goes well, the error code is 0, meaning no error or exception condition.

Error codes and diagnostics can be very helpful in batch processing and large scale deployment of devices with centralized device management.

Error code	Meaning	Modbus device condition
0	No error	
1	Block size not supported	'Firmware upgrade start' command
2	Image size too big	'Firmware upgrade start' command
3	Modbus device busy / processing data	DATA RECEIVE state
4	Invalid file offset	DATA RECEIVE/DATA RECEIVING failed state
5	Data receive error	DATA RECEIVE/DATA RECEIVING failed state
6	Image not complete error	VALIDATION/ACTIVATION failed state
7	Invalid security error	VALIDATION/ACTIVATION failed state
8	Invalid firmware for this Modbus device	VALIDATION/ACTIVATION failed state

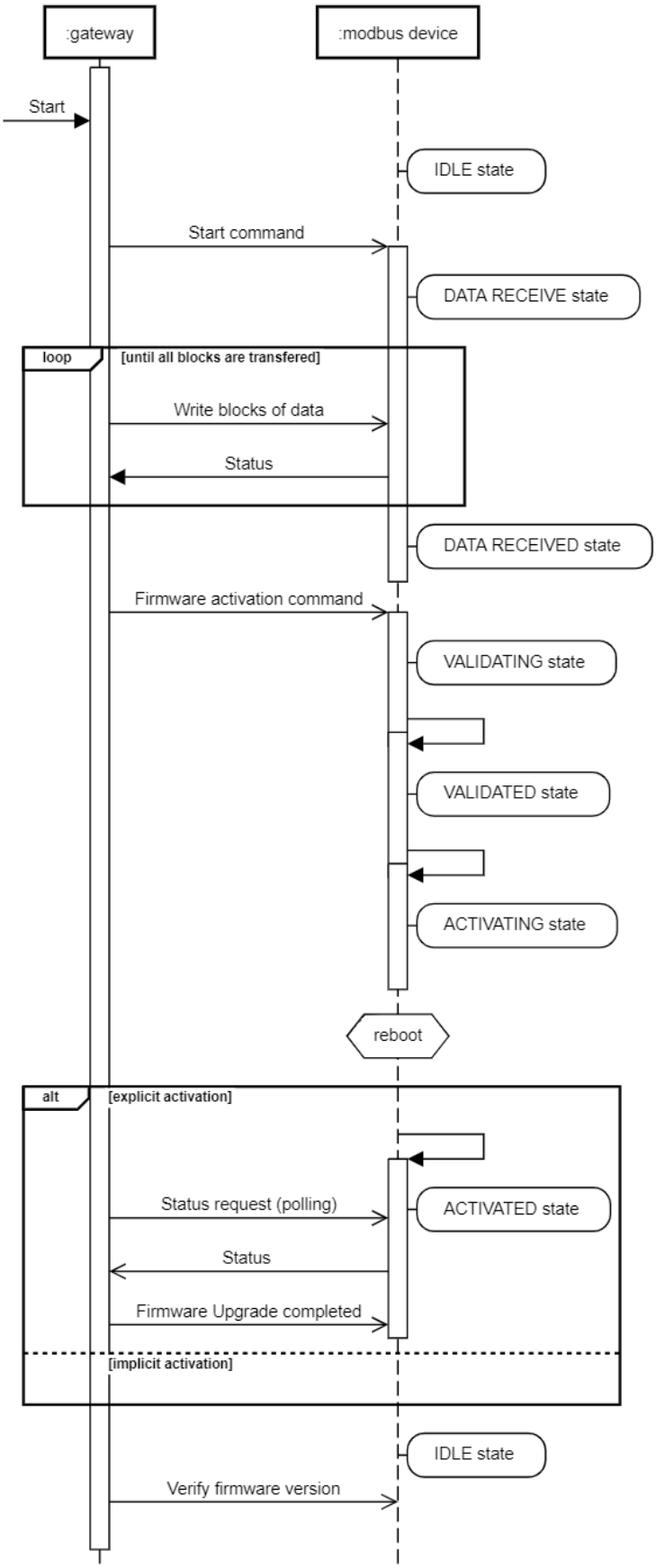


Figure 3: Stepping through states in the firmware upgrade process.

3.6 Firmware Upgrade Data Record

The Firmware Upgrade Data Record is used by the gateway to write chunks of the binary file as data blocks to the Modbus device. It is defined as a block of Modbus holding registers at the Modbus device with the starting address <fw_data_address> (predefined default: 0x4300). It has the structure shown below and it only used to write data from the gateway to the Modbus device with function code 16 (0x10).

Register Address	Holding Register value of 2 bytes
<fw_data_address>	File pointer in bytes MSB (16 bits)
<fw_data_address>+1	File pointer in bytes LSB (16 bits)
<fw_data_address>+2	data word 1
…	data word *n*
<fw_data_address>+2+N	data word N

The whole data block of N data words is transferred as one Modbus frame. The Modbus PDU contains the function code (0x10), the starting address <fw_data_address>, the number Modbus holding registers N+2, the size in bytes (2N+4), followed by the file pointer and the data block, see table above. The maximum size for N is 121, so 242 bytes.

The file pointer is the byte counter (32-bits unsigned integer) of the binary file at the side of the gateway and also serves as unique packet identifier at the Modbus device. It start at 0x0000 and is incremented by 2N after each successful transfer of a data block. The size of the data block in bytes is always 2N, except the last one. If the file size is an odd number of bytes, the last register of the PDU contains one byte of meaningful data (MSB) and one byte of dummy data, i.e. 0x00. The Modbus device is aware of the size and ignores this byte.

After writing to the Firmware Upgrade Data Record, and after receiving the errorfree acknowledgement (ACK) from the Modbus device, the gateway must request the status from the Modbus device using the Firmware Upgrade Status Record, see section 3.2. The response of the Modbus device to this request is the current state, see section 3.4, and the updated file pointer, or equally the received number of bytes.

3.7 Diagnostics and error codes

The whole firmware upgrade process is prone to errors. An error of one single bit during the transfer may result in failure of the firmware upgrade. It is therefore crucial to monitor the data integrity.

Comfortably, the Modbus protocol has a reliable error checksum for each Modbus frame. This is the first line of defence. Errors in the checksum, both at the gateway and the Modbus device, must be handled according to the Modbus protocol. If a CRC error is detected and the gateway is notified (including the check with received number of bytes in the Firmware Upgrade Status Record), it is recommended in this concept that the gateway repeats the frame it has sent before.

One important feature must be mentioned here. It is crucial that the Firmware Upgrade Status Record always provides correct information. Since the Modbus may be an autonomous and asynchronous process at the Modbus device, the Firmware Upgrade Status Record may not be updated fast enough, and a status request

results in false information. To prevent such an harmful event, the Modbus device may set error code 3 (Modbus device busy / processing data) while processing the received data and not being able to update the Firmware Upgrade Status Record with the correct information.

Further, the Modbus protocol requires that each frame sent by the master results in a response from the slave. If such a response is not returned within the time boundaries of the Modbus implementation, a time-out so to say, it is recommended in this concept that the gateway checks the received number of bytes at the side of Modbus device and acts accordingly.

How many repetitions are allowed in case time-outs or CRC failures occur more than once in a row, is implementation specific and may be configurable. It is recommended to have one repetition as default.

This concept ensures that essential information is made available to gateway by means of the requested status report in exception conditions. The gateway may act accordingly, as complex or as simple as is needed.

3.8 Physical interface aspects

The highest possible baud rate is recommended taking into account transmission line length and quality. In theory 10 Mbit/s for a RS-485 should be possible, but the transceivers (UART's) at both ends must be capable to handle asynchronous bitstreams at that speed (UART).

Since Modbus is a bus with multiple transceivers (slaves) and all must understand the signalling and protocol, the quite common baud rate (bit-rate) of 115200 bit/s is recommended.

The overall transfer rate depends on several practical factors. An 1 MB binary may take up to 10 minutes to be transferred over Modbus RTU, based on calculations from [2].

3.9 Time outs

Each protocol has to take into account exception conditions like lost connection. Also the Modbus RTU specifications [3] and [4] recommends time-outs to prevent stalled processes. The time-outs are implementation specific. And since time-outs results in implicit exception handling, alignment of time-outs and the exception handling in the firmware upgrade process at both sides of the line is very important and requires special attention. Time-outs and exception handling is recognized in the following situations.

Protocol step	Time-out condition
Data block transfer	Slave: no more data blocks received & binary file incomplete
Data block ACK response	Master: no confirmation received
Command response	Master: no confirmation received
Reboot	Master: no confirmation received

The exception handling will vary and is vendor-specific. An effective implementation is that the gateway requests status reports after time-outs and checks what the Modbus device actually expects. The gateway may then repeat the previous command (write block, etc.). If the failure is persistent, the gateway may for example restart the whole firmware upgrade process or may abort the process and reports back the central server.

4 Real Use Case of Managed Devices

In a real-life pilot, Alliander developed a district heating network with a modular energy system (MES). The network distributes heated water from a central heat source to 222 houses. The heat source is created with a central heat pump module in this case. In the future the network can be expanded modularly, for instance by adding multiple heat sources.

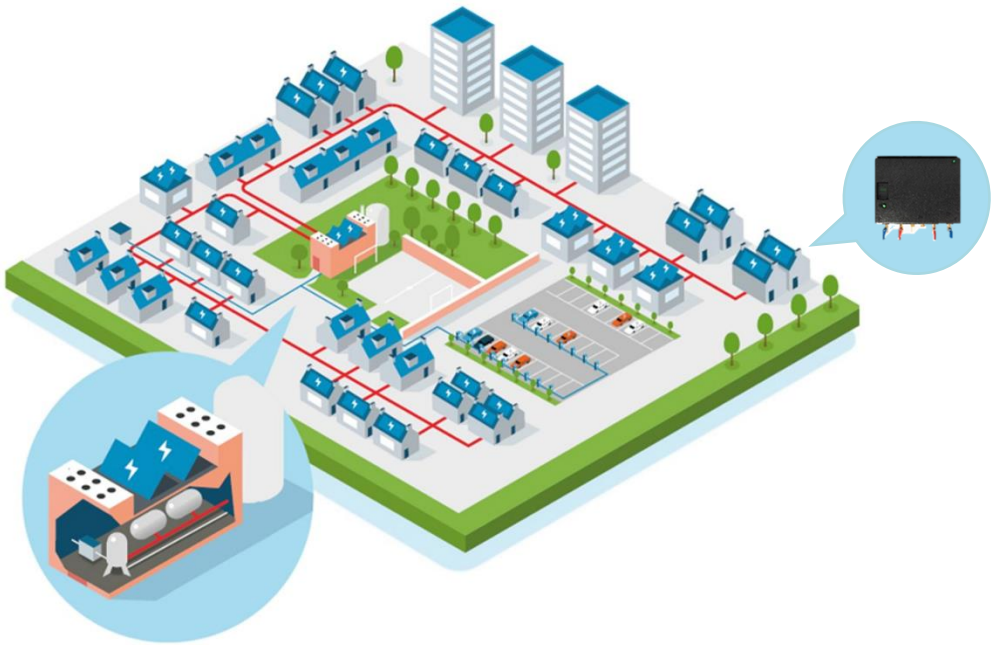


Figure 4: Impression of the experimental district heating network with a modular energy system..

The households fully rely on the district heating network and the local control runs normally autonomously. Since this is an innovative project and Alliander needs to evaluate many parameters, the possibility for remote operational control of the residential heat exchanging unit is key. And an important part of the remote operational control is the firmware upgrade of the heat exchanging unit.

The heat exchanging unit is locally connected via a Modbus RTU serial interface to a local gateway. The gateway uses mobile communication to connect with the backend systems, hosted in cloud environment. Operators of the backend systems monitor the parameters and gather measurement data (pressure, temperature, flow, valve position, etc.) And if needed, the operators adapt control, configure the device and occasionally upgrade the firmware of the gateway and the firmware of the heat exchanging unit.

Changing the firmware of the heat exchanging unit requires a firmware upgrade process over the Modbus RTU interface, from gateway to heat exchanging unit. And this is live and working successfully for more than two years!

The insert below describes the process in detail. It is a simplified version of the firmware upgrade process described in this document, optimized for this project. The gateway is the Modbus master and the software is written by Alliander. The heat exchanging unit is the Modbus slave device for which the software is written by KZ. Alliander and KZ tuned several settings during development and a very robust firmware process is established.

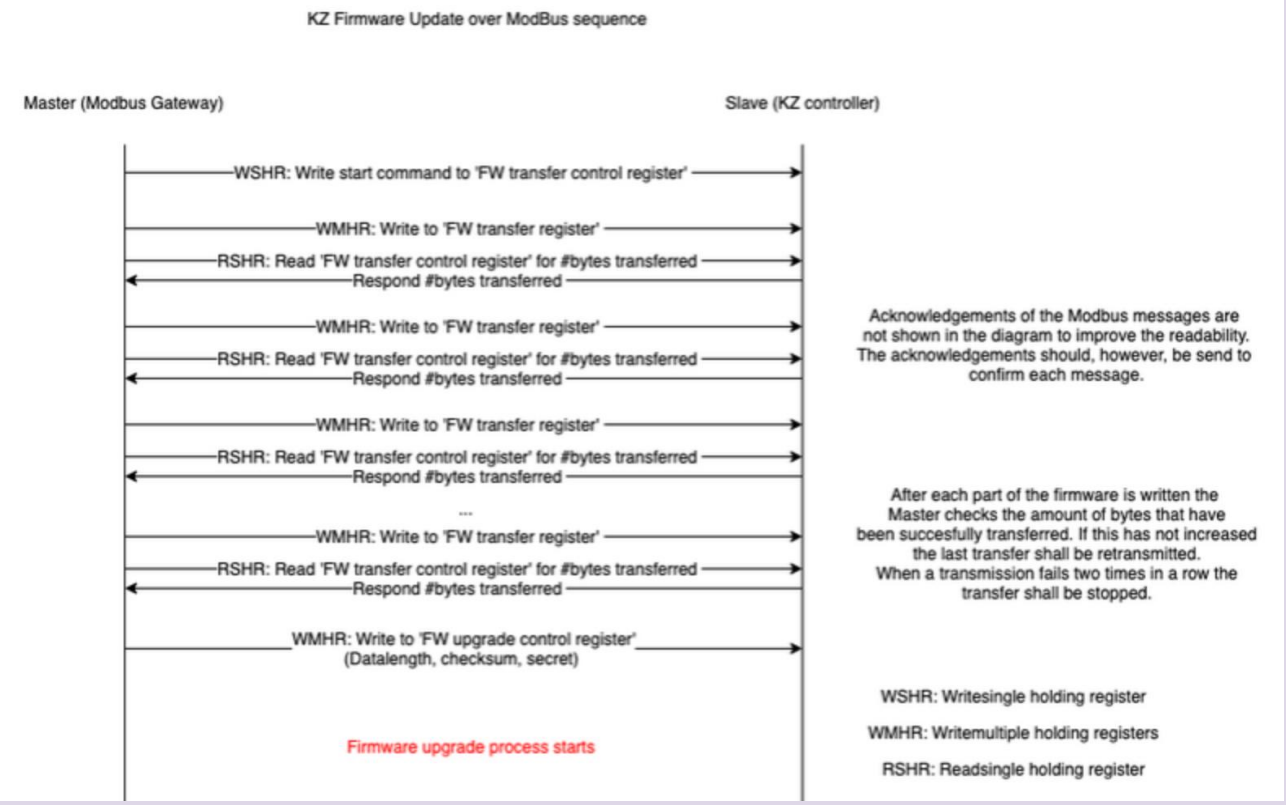
Use Case: Firmware Upgrade of Heat Exchanging Unit

Alliander uses a small router with a Linux based operating system as gateway. KZ configured an ARM Cortex-M33 based microcontroller in the heat exchanging unit as Modbus device. Together they determined the following settings:

- File size maximum: 250 kB due to memory restrictions; actual size is 180 kB
- Serial line: baud rate 19200, 8 bits, no parity
- Data block size: 64 byte (fixed)
- Wait time master / processing time data slave: 200 ms before read request after ACK writing block
- Time-out on response or ACK: 2 ms
- Number of retries: 30 in case of misalignment, direct abort in case of no response

Note that retries are hardly seen in practice. When after the retries the error condition persists, the firmware upgrade process is automatically aborted. The last step is an activation command. The implementation of the normal data transfer process is shown below.

With this implementation a firmware file of 180 kB is transferred over an 1.5 meter unshielded cable in 6 to 10 minutes. In the pilot, several times all devices were updated with new firmware. For 98% of the devices the transfer was successful at the first attempt. With a second pass, 100% of the 222 devices were upgraded successfully.



5 Conclusion

This white paper constructs a versatile firmware upgrade process using standard Modbus RTU protocol elements and by defining control, status and data records. Developers may optimize the process, like more or less robustness, more or less throughput, more or less automated steps, etcetera, by adding or leaving out certain elements.

A basic implementation is successfully deployed in a district heating network and runs reliable for several years.

Note that both sides, developers of the gateway and developers of the Modbus device, must agree on this process. A companion specification and a thorough integration and verification test is highly recommended.

6 References

[1] Netbeheer Nederland, „Dutch Smart Meter Requirements 4.2.3 - Main Document," 2019.

[2] J. Ding, „A File Transfer Method Based on Modbus Protocol," in *8th International Conference on Power and Renewable Energy (ICPRE)* , Shanghai, China, 2023.

[3] Modbus Organization, „MODBUS over serial line specification and implementation guide V1.02," Modbus.org, 2006.

[4] Modbus Organization, „MODBUS Application Protocol Specification V1.1b3," Modbus.org, 2012.

Duurzaam Energie Perspectief | Energie Management

Duurzaam Energie Perspectief
Bezoekadres
Basisweg 10, 1043 AP Amsterdam
Utrechtseweg 68, 6812 AH Arnhem
Dijkgraaf 4, 6921 RL Duiven
Postadres
Postbus 50, 6920 AB Duiven
KvK 09119276
Telefoon: (088) – 191 15 61
www.dep.nl
contact@dep.nl