SPBUS PROTOCOL SPECIFICATION

TABLE OF CONTENTS 1
PURPOSE 3
PRELIMINARIES 4
Abbreviations 4
Numeric notations 4
INTRODUCTION 5
SPBUS network 6
SPBUS network architectures 6
Timing considerations 7
MESSAGE FRAME 9
Field descriptions 9
Packet_start 10
Command 10
Hardware_id 11
Address 11
(Transaction) Identifier 12
Status 12
Register_start 13
Register_length 13
Data 14
Reserved 15
CRC 15
Packet_end 16
Commands 16
Read_register_layer_1 16
Read_register_layer_2 17
Write_register_layer_1 17
Write_register_layer_2 18
Set_address 18
Ignore 19
Broadcast_displays_on 19
Broadcast_displays_off 19
Broadcast_scan 19
Broadcast_status 19
Broadcast_write_register 20
Direct addressing message propagation 20
Broadcast (message propagation) 20
Without reply 21
With reply 21
Message frame construction examples 23
Read firmware version of PDU with unit address 42, transaction identifier 1, and FW version 233 (2.33) 23

Scan SPBUS network which consists of 2 SPBDs, one with unit address 8, and the other with unit address 87 23
FIRMWARE UPGRADE **24**
Firmware upload preparations 24
Firmware data upload 24
SPBUS PACKET OVER ETHERNET (Client <-> Master) **25**
Connection **25**
Arc4 **25**
Packet construction 25
Example packet construction 26

2. PURPOSE

The purpose of this document is to provide a concrete and clear description for developers who intend to integrate the SPBUS (Schleifenbauer Products BUS) protocol into their (custom) software solutions. This document will provide information on:

- Which communication protocols are used by SPBUS
- How to use said communication protocols with SPBUS
- SPBUS network architecture
- SPBUS message frame
- Available commands for SPBUS networks

With the information given in this document a developer should be able to:

- Create SPBUS network configurations
- Communicate with Schleifenbauer devices within an SPBUS network
- Incorporate SPBUS network communications within custom software solutions

3. PRELIMINARIES

3.1. Abbreviations

SPBUS	Schleifenbauer Products BUS
PDU	Power Distribution Unit
PC	Personal Computer
SPBD	SPBUS Device
SPDM	Schleifenbauer Products Data Model
TCP/IP	Transmission Communication Protocol / Internet Protocol
IPAPI	Internet Protocol Application Programming Interface

3.2. Numeric notations

- Hexadecimal numbers are denoted by a "0x" prefix
- Octal number are denoted by a "0" prefix
- Bits numbers are denoted by a "b" prefix
- The remainder of standalone numbers within this document are base-10 decimal numbers

4. INTRODUCTION

SPBUS protocol is an application communication protocol developed by Schleifenbauer Products as message protocol used with their product line of SPBDs (SPBUS Devices) such as PDUs (Power Distribution Units) and the DPM3. This message protocol allows SPBD commands and data to be either sent or requested externally, therefore allowing users to change or display SPBD data.

An example of sending SPBD configuration data over SPBUS is configuring a new name for a SPBD, or alternatively, retrieving the current name of a SPBD over SPBUS to display (e.g. to monitor the status of the SPBD).

An SPBUS network is a master-slave network of which a client (e.g. PC (Personal Computer)) is in communication with the master of the SPBUS network, or acts as the master itself in a SPBUS network. The client device ought to initiate any requests and can either address a single device or can use broadcasts to address multiple devices within the SPBUS network.

The SPBUS protocol uses RS485 (with a baud rate of 115200) and, optionally, Ethernet communication for sending and receiving SPBD commands. This specification will describe the SPBUS protocol such that a developer can create own tool(s) to communicate with a SPBD over either RS485, or Ethernet.

5. SPBUS network

5.1. SPBUS network architectures

A typical SPBUS network architecture consists of:

- 0 or 1 Client(s) (e.g. PC, server, etc.)
- 1 Master SPBD
- *n* Slave SPBDs, with 0 <= *n* <= 65535

The communication from a client to a master SPBD can only go over Ethernet (note that only new versions of PDUs support an Ethernet connection). The communication between the SPBDs themselves is currently done using only RS485, in which the SPBDs are connected using a daisy chain of RS485 connections. Within such a daisy chain network a SPBD part of the network has either one or two active RS485 connections to other SPBDs. Every SPBD to SPBD connection is a standalone connection but creates a network as more SPBDs get daisy-chained to other SPBDs using an SPBD's RS485 connections. All commands are forwarded from one RS485 connection to another.

The SPBDs within a SPBUS network configuration can have different configurations. Namely, an open ring configuration (Figure 1, Open ring SPBUS network) or a closed ring configuration (Figure 2, Closed ring SPBUS network). Whether a configuration is open or closed is defined by whether a SPBD has a (daisy-chained) connection between its own two RS485 connections, meaning that if a message is sent over a SPBD's RS485 connection 1 and the message keeps getting forwarded by the slaves, then it will be received on RS485 connection 2 of the same SPBD. The purpose of a closed ring network is to increase robustness, as open ring configurations are prone to disconnections between SPBDs within the SPBUS network in case of an RS485 connection failure. An RS485 connection failure in a closed ring would merely revert the SPBUS network back to an open ring configuration whilst the disconnection problem can be addressed and is therefore more robust against RS485 connection failure.

Another option is to configure a client as a master by making a client use a RS485 connection to communicate with a SPBD. In such cases a SPBUS network architecture might look as shown in Figure 3, closed ring where client is also the master. The open and closed ring configuration options still apply in such a case but the client/master has to consider whether it's in a closed or open configuration. In case of a closed ring configuration it should ignore its own messages when they're received on its 2nd RS485 connection. In case of an open ring, the client/master must ensure that messages are sent over both RS485 connections. Checking whether a configuration is open or closed can be done using the 'Ignore' command described in the chapter 'Commands'.

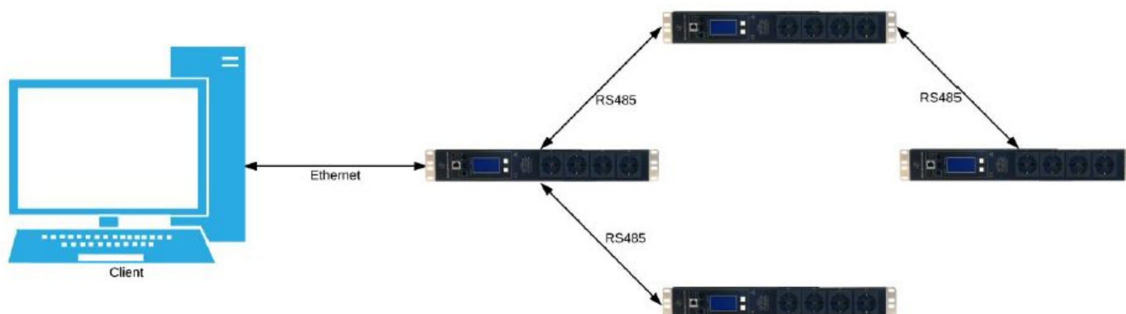


Figure 1, Open ring SPBUS network

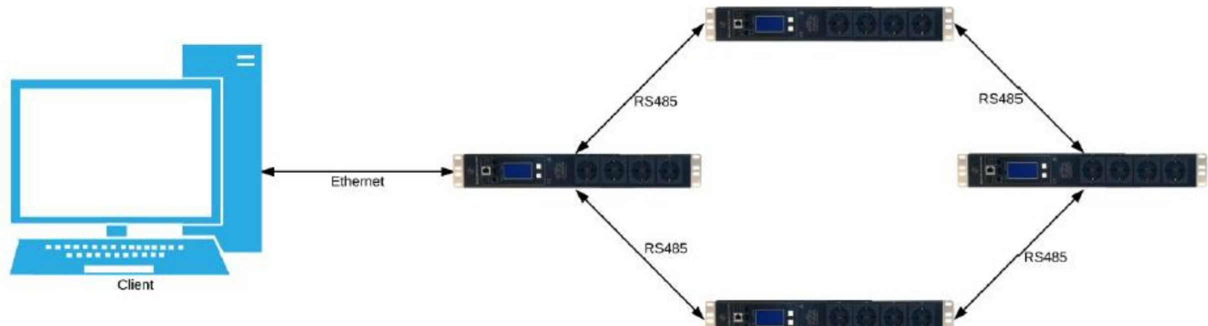


Figure 2, Closed ring SPBUS network

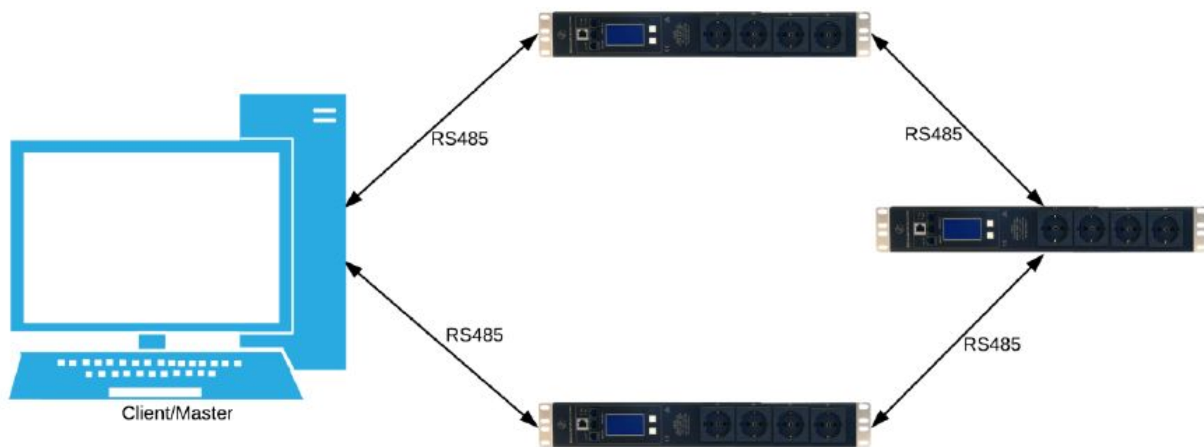


Figure 3, Closed ring where client is also the master

5.2. Timing considerations

Messages on the SPBUS network's SPBDs are sent on initiative of either the client, or the master SPBD. The request initiator sends a message frame to a SPBD through the SPBUS' daisy-chain network. Every SPBD forwards the message to the other RS485 port until the addressed slave receives the message. If the addressed SPBD does not exist within the SPBUS network, then there will either be no reply, or in a closed ring configuration, the message will go around until it is received by the master again. However, because open loops cannot rely on a message returning at the master it is necessary to implement a timeout of 0.2s (200ms) when expecting a reply.
In other words, if 0.2s has elapsed after a request where a reply is expected but no reply has been received, then assume the request failed and discard any reply if received after the timeout. It may be the case that the SPBD exists but the timeout happened regardless. In such cases another request might be preferred instead of assuming the SPBD does not exist.
In case a broadcast with expected reply is send by an initiator, every response resets the master's 0.2s timeout until no responses are received within the 0.2s timeout. See the chapter 'Broadcast (message propagation)' for more information on broadcasts and how messages ought to be forwarded for broadcasts.
Replies of directly addressed requests, i.e. requests addressed for a particular SPBD, respond after 50ms (see Figure 4, Directly addressed Read register command with reply). This delay is in place to support backwards compatibility with older SPBDs.

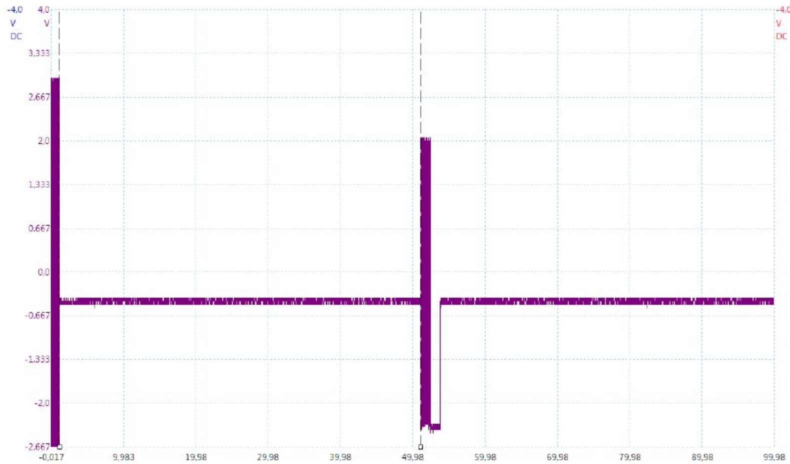


Figure 4, Directly addressed Read_register command with reply

Similarly, SPBDs receiving broadcasts with replies will respond after a delay; However, the delays for broadcasts with replies is 25ms between any two messages on the SPBUS and will therefore cause the broadcast replies to have intervals of 25ms between each reply and between the request and first reply (see Figure 5, Broadcast scan command with 4 replies and chapter 'Broadcast (message propagation)').

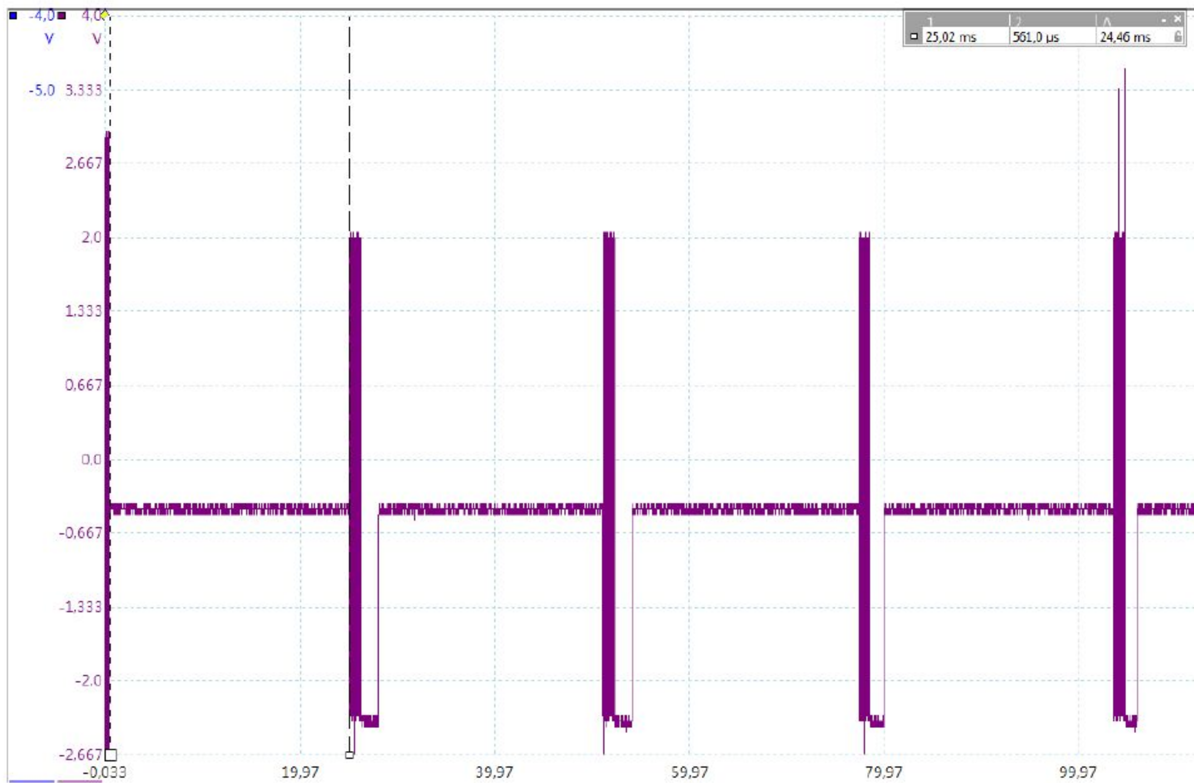


Figure 5, Broadcast_scan command with 4 replies

6. MESSAGE FRAME

A message frame consists of multiple fields. Some fields are mandatory for all message frames and other fields are dependent on the type of message. The following table offers a brief description of these fields. 'Field' is the message frame field that is to be described, 'Mandatory' denotes whether the field occur in every message frame regardless of what's sent, 'Size' denotes the size of the field in bytes, and 'Summary' briefly describes the purpose of the field. If a field does occur within a message frame, then the order of the fields are as they occur in Table 1, Brief message frame field descriptions **with the exception of the 'Address', and 'Hardware_id' fields** (refer to the chapter for more information)

Field	Mandatory	Size	Summary
Packet_start	Yes	1	Message start byte
Command	Yes	1	Command to execute
Hardware_id	No	6	SPBD's unique hardware identification number
Address	No	2	Address of SPBD to execute the command
Identifier	No	2	Transaction identifier (generated by master)
Status	No	6	Status of SPBD
Register_start	No	2	Register start address (for reading/writing to and from a register)
Register_length	No	2	Number of bytes to read/write starting from Register start
Data	No	x	Register data.
Reserved	No	1	Reserved.
CRC	Yes	2	16-bit Cyclic Redundancy Check
Packet_end	Yes	1	Message end byte

Table 1, Brief message frame field descriptions

Notes:

1. Value 'x' should be exactly equal to 'Register length'
2. Order of values consisting of bytes >= 2 are in little-endian (low byte – high byte order). I.e. address 0x1234 becomes 0x3412 on the SPBUS. However, the field order remains the same.
3. The **message frame size limit is 512**. Any messages that exceed this limit are discarded.

6.1. Field descriptions

The chapter's introduction described some message frame fields and whether said fields are included in the message frame or depend on the 'Command' field value for inclusion. This chapter describes every field in detail and denotes whether a field is included in a message frame or not, depending on the 'Command' field's value. The types of messages are not only dependent on the command type, but also the direction of the message (i.e. whether it's a master request or slave reply).

The 'Command' field requires a more detailed explanation and is therefore described in the chapter 'Commands'. Because the 'Command' field is the deciding factor in the message frame's construction, the 'Commands' chapter also details the construction of all message frames using the fields.

The following format is used for the field inclusion table; command, direction (master to slave, M->S), direction (slave to master, S->M). A 'Yes' in the inclusion table denotes inclusion of the field to the message frame; a 'No' denotes that the field is not included to the command's message frame.

The construction field describes the format of the field where the following notations are used:

- X = a single hexadecimal digit
- C = a single ASCII character

- N = any of the above notation options, any amount of times

6.1.1. Packet_start

- Description: Prefix for every message on the SPBUS denoting the start of a request message, or a successful/unsuccessful slave reply. Can only be one of the following values:
  - STX: Start TX, the start byte denoting the start of a message. Can only occur in messages sent by the master to one of the slave SPBDs. If packets from master to slave do not contain this as start byte, the packet will be ignored by the SPBDs.
    - Value: 0x02
  - ACK: ACKnowledgement of a message frame. Only occurs in messages sent by a slave SPBD when said slave successfully received and processed the master's command.
    - Value: 0x06
  - NAK: No-AcKnowledgement of a message frame. Only occurs in messages sent by a slave when said slave could not successfully process the master's command.
    - Value: 0x0F
- Size: 1 byte
- Construction: XX
- Inclusion table: This field is mandatory in every message

	M->S	S->M
Read_register_layer_1	Yes	Yes
Read_register_layer_2	Yes	Yes
Write_register_layer_1	Yes	Yes
Write_register_layer_2	Yes	Yes
Set_address	Yes	Yes
Ignore	Yes	Yes
Broadcast_displays_on	Yes	Yes
Broadcast_displays_off	Yes	Yes
Broadcast_ignore	Yes	Yes
Broadcast_scan	Yes	Yes
Broadcast_status	Yes	Yes
Broadcast_write_registers	Yes	Yes

6.1.2. Command

- Description: Refer to the chapter 'Commands' for full specification commands.
- Size: 1 byte
- Construction: XX
- Inclusion table: This field is mandatory in every message

	M->S	S->M
Read_register_layer_1	Yes	Yes
Read_register_layer_2	Yes	Yes
Write_register_layer_1	Yes	Yes

Write_register_layer_2	Yes	Yes
Set_address	Yes	Yes
Ignore	Yes	Yes
Broadcast_displays_on	Yes	Yes
Broadcast_displays_off	Yes	Yes
Broadcast_ignore	Yes	Yes
Broadcast_scan	Yes	Yes
Broadcast_status	Yes	Yes
Broadcast_write_registers	Yes	Yes

6.1.3. Hardware_id

- Description: Hardware identification number of a SPBD. Can be obtained from the 'ABOUT' tab on the SPBD's display as the 'ID' value. Defined as a 3-tuple, each represented by a 2-byte unsigned integer (e.g. 44013-05345-00000). The construction is simply a concatenation of the three 2-byte unsigned integer values.
- Size: 6 bytes
- Construction: XXXXXXXXXXXX
- Inclusion table:

	M->S	S->M
Read_register_layer_1	No	No
Read_register_layer_2	No	No
Write_register_layer_1	No	No
Write_register_layer_2	No	No
Set_address	Yes	Yes
Ignore	No	No
Broadcast_displays_on	No	No
Broadcast_displays_off	No	No
Broadcast_ignore	No	No
Broadcast_scan	No	Yes
Broadcast_status	No	No
Broadcast_write_registers	No	No

6.1.4. Address

- Description: Unit address of a SPBD. Can refer to either a master, or slave SPBD. Used to address SPBDs for command execution. A 'Broadcast_scan' command will reveal all the addresses of the units currently on the SPBUS network (see 'Broadcast (message propagation)').
- Size: 2 bytes
- Construction: XXXX
- Inclusion table:

	M->S	S->M
Read_register_layer_1	Yes	Yes
Read_register_layer_2	Yes	Yes
Write_register_layer_1	Yes	Yes
Write_register_layer_2	Yes	Yes
Set_address	Yes	Yes
Ignore	Yes	No
Broadcast_displays_on	No	No
Broadcast_displays_off	No	No
Broadcast_ignore	No	No
Broadcast_scan	No	Yes
Broadcast_status	No	Yes
Broadcast_write_registers	No	No

6.1.5. (Transaction) Identifier

- Description: Transaction identifier to recognize reply messages with the corresponding request messages. The initiator of a message ought to set this field. To implement this identifier, the first transaction should initiate on 0 and increment by 1 at every new command sent by the initiator. Integer overflows are allowed (unsigned) when the 2-byte limit is reached.
- Size: 2 bytes
- Construction: XXXX
- Inclusion table:

	M->S	S->M
Read_register_layer_1	Yes	Yes
Read_register_layer_2	Yes	Yes
Write_register_layer_1	Yes	Yes
Write_register_layer_2	Yes	Yes
Set_address	No	No
Ignore	Yes	No
Broadcast_displays_on	No	No
Broadcast_displays_off	No	No
Broadcast_ignore	No	No
Broadcast_scan	No	No
Broadcast_status	No	No
Broadcast_write_registers	No	No

6.1.6. Status

- Description: The status bytes of an SPBD. The first byte contains the 'ssstat' value of the SPBD. The remaining 5 bytes are reserved.
- Size: 6 bytes
- Construction: XXXXXXXXXXXX
- Inclusion table:

	M->S	S->M
Read_register_layer_1	No	No
Read_register_layer_2	No	No
Write_register_layer_1	No	No
Write_register_layer_2	No	No
Set_address	No	No
Ignore	No	No
Broadcast_displays_on	No	No
Broadcast_displays_off	No	No
Broadcast_ignore	No	No
Broadcast_scan	No	No
Broadcast_status	No	Yes
Broadcast_write_registers	No	No

6.1.7. Register_start

- Description: Start address of a register. Refer to the SPDM document for all available registers, their purpose, and accessibility (Read-Only, Read/Write, Write-Only).
- Size: 2 bytes
- Construction: XX
- Inclusion table:

	M->S	S->M
Read_register_layer_1	Yes	Yes
Read_register_layer_2	Yes	Yes
Write_register_layer_1	Yes	No
Write_register_layer_2	Yes	No
Set_address	No	No
Ignore	No	No
Broadcast_displays_on	No	No
Broadcast_displays_off	No	No
Broadcast_ignore	No	No
Broadcast_scan	No	No
Broadcast_status	No	No
Broadcast_write_registers	Yes	No

6.1.8. Register_length

- Description: Size of the data to be written or the size of data requested. 'Register_start + Register_length' = end address of what's to be written/read.
  - E.g. a 'Write_register_layer_1' command using fields 'Register_start' = 158, 'Register_length' = 2, and 'Data' = 56 = 0x38 (i.e writing a new unit address to a device), then '0x3800' (note: little-endian) will be written at address 158, until address 'Register_start' + 'Register_length' = 158 + 2 = 160
- Size: 2 byes
- Construction: XXXX
- Inclusion table:

	M->S	S->M
Read_register_layer_1	Yes	Yes
Read_register_layer_2	Yes	Yes
Write_register_layer_1	Yes	No
Write_register_layer_2	Yes	No
Set_address	No	No
Ignore	No	No
Broadcast_displays_on	No	No
Broadcast_displays_off	No	No
Broadcast_ignore	No	No
Broadcast_scan	No	No
Broadcast_status	No	No
Broadcast_write_registers	Yes	No

6.1.9. Data

- Description: Data field that denotes the data sent with 'Register_write', and received with 'Register_read' replies. Refer to the chapter 'Commands' for more information.
- Size: Register_length
- Construction: N
- Inclusion table:

	M->S	S->M
Read_register_layer_1	No	Yes
Read_register_layer_2	No	Yes
Write_register_layer_1	Yes	No
Write_register_layer_2	Yes	No
Set_address	No	No
Ignore	No	No
Broadcast_displays_on	No	No
Broadcast_displays_off	No	No
Broadcast_ignore	No	No
Broadcast_scan	No	No
Broadcast_status	No	No
Broadcast_write_registers	Yes	No

6.1.10. Reserved

- Description: Reserved space. Only included in replies with the 'Packet_start' field set to 'NAK'.
- Size: 1
- Construction: XX
- Inclusion table:

	M->S	S->M
Read_register_layer_1	No	Yes*
Read_register_layer_2	No	Yes*
Write_register_layer_1	No	Yes*
Write_register_layer_2	No	Yes*
Set_address	No	Yes*
Ignore	No	No
Broadcast_displays_on	No	No
Broadcast_displays_off	No	No
Broadcast_ignore	No	No
Broadcast_scan	No	No
Broadcast_status	No	No
Broadcast_write_registers	No	No

*= Only applies if the 'Packet_start' field is set to 'NAK'.

6.1.11. CRC

- Description: Cyclic Redundancy Check validation on the fields between field 'Packet_start', and 'CRC' (**excluding 'Packet_start', and 'CRC' in the validation data**)
  - The CRC field is the checksum result of CRC16-CCITT with the fields between 'Packet_start' and 'CRC', excluding the 'Packet_start', and 'CRC' fields in the calculation. In the example, the fields used for the CRC checksum are highlighted.

Example: [Packet_start][Command][...][CRC][Packet_end]

- Size: 2
- Construction: XXXX
- Inclusion table: This field is mandatory in every message

	M->S	S->M
Read_register_layer_1	Yes	Yes
Read_register_layer_2	Yes	Yes
Write_register_layer_1	Yes	Yes
Write_register_layer_2	Yes	Yes
Set_address	Yes	Yes
Ignore	Yes	Yes
Broadcast_displays_on	Yes	Yes
Broadcast_displays_off	Yes	Yes
Broadcast_ignore	Yes	Yes
Broadcast_scan	Yes	Yes
Broadcast_status	Yes	Yes
Broadcast_write_registers	Yes	Yes

6.1.12. Packet_end

- Description: Suffix for every message on the SPBUS. Can only be the following value:
  - ETX: End TX
    - Value: 0x03
- Size: 1
- Construction: XX
- Inclusion table: This field is mandatory in every message

	M->S	S->M
Read_register_layer_1	Yes	Yes
Read_register_layer_2	Yes	Yes
Write_register_layer_1	Yes	Yes
Write_register_layer_2	Yes	Yes
Set_address	Yes	Yes
Ignore	Yes	Yes
Broadcast_displays_on	Yes	Yes
Broadcast_displays_off	Yes	Yes
Broadcast_ignore	Yes	Yes
Broadcast_scan	Yes	Yes
Broadcast_status	Yes	Yes
Broadcast_write_registers	Yes	Yes

6.2. Commands

A message frame's construction depends heavily on the command of the master, and the direction it's going (i.e. whether it's a master's command or a slave's reply). This chapter describes both the commands in detail, and the construction of the message frame when using said command.

The construction explanation works as follows: '[' and ']' delimits the message frame's fields. Some fields use one of the predefined values of said field. The used predefined value is denoted by a suffix of a dot, plus the name of said field option (e.g. Packet_start.STX). Information on these options can be found in the 'Field descriptions' chapter.

6.2.1. Read_register_layer_1

- Description: Request from master to a slave SPBD to read a register and reply with the contents of said register. If a register has an extension, then this command will allow reads to channels 1-27. Refer to the SPDM (Schleifenbauer Products Data Model) to find all registers and their purpose. Note that some registers are not readable (i.e. write-only). In case a 'Read_register_layer_1' command is sent to one of the SPBDs write-only registers, the SPBD will discard the message and reply with a 'NAK' message frame.
- Value: 0x01 = b00000001 = 1
- Note that the size of the 'Data' field is exactly equal to 'Register_length'.
- This command also supports reading from multiple adjacent registers at once. This can be done by increasing the value of the 'Register_length' field to include all desired and adjacent registers.
- Construction:
  - Request:
    - [Packet_start.STX][Command.Read_register_layer_1][Address][Identifier][Register_start][Register_length][CRC][Packet_end.ETX]

- Total size: 13 bytes

- Reply:
  - Success:
    - [Packet_start.ACK][Command.Read_register_layer_1][Address][Identifier][Register_start][Register_length][Data][CRC][Packet_end.ETX]
      - Total size: 13 bytes + 'Register_length'
  - Failure:
    - [Packet_start.NAK][Command.Read_register_layer_1][Address][Identifier][Reserved][CRC][Packet_end.ETX]
      - Total size: 10 bytes

6.2.2. Read_register_layer_2

- Description: Request from master to a slave SPBD to read a register and reply with the contents of said register. If a register has an extension, then this command will allow reads to channels 28-54 (addressed as 1-27). Refer to the SPDM (Schleifenbauer Products Data Model) to find all registers and their purpose. Note that some registers are not readable (i.e. write-only). In case a 'Read_register_layer_2' command is sent to one of the SPBD's write-only registers, the SPBD will discard the message and reply with a 'NAK' message frame.
- Value: 0x02 = b00000010 = 2
- Note that the size of the 'Data' field is exactly equal to 'Register_length'.
- This command also supports reading from multiple adjacent registers at once. This can be done by increasing the value of the 'Register_length' field to include all desired and adjacent registers.
- Construction:
  - Request:
    - [Packet_start.STX][Command.Read_register_layer_2][Address][Identifier][Register_start][Register_length][CRC][Packet_end.ETX]
      - Total size: 13 bytes
  - Reply:
    - Success:
      - [Packet_start.ACK][Command.Read_register_layer_2][Address][Identifier][Register_start][Register_length][Data][CRC][Packet_end.ETX]
        - Total size: 13 bytes + 'Register_length'
    - Failure:
      - [Packet_start.NAK][Command.Read_register_layer_2][Address][Identifier][Reserved][CRC][Packet_end.ETX]
        - Total size: 10 bytes

6.2.3. Write_register_layer_1

- Description: Request from a master to a slave SPBD to write what's in the 'Data' field to a register. Refer to the SPDM to find all registers and their purpose. If a register has an extension, then this command will allow writes to channels 1-27. Note that some registers are not writable (i.e. read-only). In case a 'Write_register_layer_1' command is sent to one of the SPBDs read-only registers, the SPBD will discard the message and reply with a 'NAK' message frame.

- Value: 0x10 = b00010000 = 16

- Note that the size of the 'Data' field is exactly equal to 'Register_length'.

- This command also supports writing to multiple adjacent registers at once. This can be done by increasing the value of the 'Register_length' field to include all desired and adjacent registers and adjusting the 'data' field to support the 'Register_length' field's resize.

- Construction:
  - Request:
    - [Packet_start.STX][Command.Write_register_layer_1][Address][Identifier][Register_start][Register_length][Data][CRC][Packet_end.ETX]
      - Total size: 13 bytes + Register_length
  - Reply:
    - Success:
      - [Packet_start.ACK][Command.Write_register_layer_1][Address][Identifier][CRC][Packet_end.ETX]
        - Total size: 9 bytes
    - Failure:
      - [Packet_start.NAK][Command.Write_register_layer_1][Address][Identifier][Reserved][CRC][Packet_end.ETX]
        - Total size: 10 bytes

6.2.4. Write_register_layer_2

- Description: Request from a master to a slave SPBD to write what's in the 'Data' field to a register. Refer to the SPDM to find all registers and their purpose. If a register has an extension, then this command will allow writes to channels 28-54 (addressed as 1-27). Note that some registers are not writable (i.e. read-only). In case a 'Write_register_layer_2' command is sent to one of the SPBDs read-only registers, the SPBD will discard the message and reply with a 'NAK' message frame.
- Value: 0x11 = b00010001 = 17
- Note that the size of the 'Data' field is exactly equal to 'Register_length'.
- This command also supports writing to multiple adjacent registers at once. This can be done by increasing the value of the 'Register_length' field to include all desired and adjacent registers and adjusting the 'data' field to support the 'Register_length' field's resize.
- Construction:
  - Request:
    - [Packet_start.STX][Command.Write_register_layer_2][Address][Identifier][Register_start][Register_length][Data][CRC][Packet_end.ETX]
      - Total size: 13 bytes + Register_length
  - Reply:
    - Success:
      - [Packet_start.ACK][Command.Write_register_layer_2][Address][Identifier][CRC][Packet_end.ETX]
        - Total size: 9 bytes
    - Failure:
      - [Packet_start.NAK][Command.Write_register_layer_2][Address][Identifier][Reserved][CRC][Packet_end.ETX]
        - Total size: 10 bytes

6.2.5. Set_address

- Description: Sets the unit address of a SPBD to [Address]. Uses the hardware identifier of said SPBD to address it. Note that message frames with this command have the 'Hardware_id' field prior to the 'Address' field during master send/slave receive.
- Value: 0x20 = b00100000 = 32
- Construction:
  - Request:
    - [Packet_start.STX][Command.Set_address][Hardware_id][Address][CRC][Packet_end.ETX]

- Total size: 13 bytes
- Reply:
  - Success:
    - [Packet_start.ACK][Command.Set_address][Hardware_id][Address][CRC][Packet_end.ETX]
      - Total size: 13 bytes
  - Failure:
    - [Packet_start.NAK][Command.Set_address][Hardware_id][Address][Reserved][CRC][Packet_end.ETX]
      - Total size: 14 bytes

6.2.6. Ignore

- Description: Slave SPBDs that receive this command will ignore it and forward the message to the next SPBD. This command is typically used to determine whether the network is configured with a closed, or open ring configuration.
  In other words, if this command is sent by the master over SPBUS connection 1 and received on SPBUS connection 2, then there is a closed ring configuration. If this command is sent by the master over SPBUS connection 1 and not received on SPBUS connection 2 within 200ms, then there is an open ring configuration.

- Value: 0x40 = b01000000 = 64
- Construction:
  - Request:
    - [Packet_start.STX][Command.Ignore][Address][Identifier][CRC][Packet_end.ETX]
      - Total size: 9 bytes
  - Reply: No reply.

6.2.7. Broadcast_displays_on

- Description: Enables the displays of all SPBD devices on the SPBUS.
- Value: 0x80 = b10000000 = 128
- Construction:
  - Request:
    - [Packet_start.STX][Command.Broadcast_displays_on][CRC][Packet_end.ETX]
      - Total size: 5 bytes
  - Reply: No reply.

6.2.8. Broadcast_displays_off

- Description: Disables the displays of all SPBD devices on the SPBUS.
- Value: 0x81 = b10000001 = 129
- Construction:
  - Request:
    - [Packet_start.STX][Command.Broadcast_displays_off][CRC][Packet_end.ETX]
      - Total size: 5 bytes
  - Reply: No reply.

6.2.9. Broadcast_scan

- Description: Request to all SPBDs on the SPBUS network to identify themselves. Note that with this command, the master expects a reply from every device. Refer to the 'Broadcast (message propagation)' chapter for more information. **Note that the message frames with this command have the 'Address' field prior to the 'Hardware_id' field during slave send/master receive.**
- Value: 0x90 = b10010000 = 144
- Construction:
  - Request:
    - [Packet_start.STX][Command.Broadcast_scan][CRC][Packet_end.ETX]
      - Total size: 5 bytes

- Reply:
  - [Packet_start.ACK][Command.Broadcast_scan][Address][Hardware_id][CRC][Packet_end.ETX]
    - Total size: 13 bytes

6.2.10. Broadcast_status

- Description: Request to all SPBDs on the SPBUS network to reply with their current status. Typically used to quickly gather all events and alerts. Note that with this command, the master expects a reply from every device. Refer to the 'Broadcast (message propagation)' chapter for more information on replies.
- Value: 0x91 = b10010001 = 145
- Construction:
  - Request:
    - [Packet_start.STX][Command.Broadcast_status][CRC][Packet_end.ETX]
      - Total size: 5 bytes
  - Reply:
    - [Packet_start.ACK][Command.Broadcast_status][Address][Status][CRC][Packet_end.ETX]
      - Total size: 13 bytes

6.2.11. Broadcast_write_register

- Description: Write to one or more registers of all slaves currently on the SPBUS network. Note that with this command, the master does not expect a reply from any device. Refer to the 'Broadcast (message propagation)' chapter for more information on broadcasts.
- Value: 0xA0 = b10100000 = 160
- Construction:
  - Request:
    - [Packet_start.STX][Command.Broadcast_write_register][Register_start][Register_length][Data][CRC][Packet_end.ETX]
      - Total size: 9 bytes + 'Register_length'
  - Reply: No reply.

6.3. Direct addressing message propagation

Figure SEQ "Figure" * ARABIC 6, Direct addressing message propagation

A message addressed to a single SPBD will be forwarded until the end of the SPBUS network, regardless whether the addressed SPBD is found during the forwarding of the command. This means that every SPBD is ought to forward the

message before detecting whether the message was addressed to it and process further if necessary.

I.e. if the addressed SPBD exists within the SPBUS network, then it will forward the message, process the message/command, and finally, reply to the master (see Figure 6, Direct addressing message propagation).

6.4. Broadcast (message propagation)

Broadcasts are messages with more than 1 recipient. As mentioned in previous chapters, some broadcast request messages expect a response of every device on the SPBUS to be send to the initiator of the broadcast and some do not reply back at the broadcast at all.

6.4.1. Without reply

If a broadcast request is sent on the SPBUS and no reply is expected, then every slave that receives the broadcast will forward it to the next SPBD in the network before processing the command (see Figure 7, Broadcast without replies). An example of this is Figure 8, Broadcast_displays_on command forwarding where a 'Broadcast_display_on' command is send on the SPBUS. We see the data "BEFORE", denoting the SPBUS connection before an SPBD and the data "AFTER", denoting the SPBUS connection after that same SPBD.

Figure SEQ "Figure" * ARABIC 7, Broadcast without replies

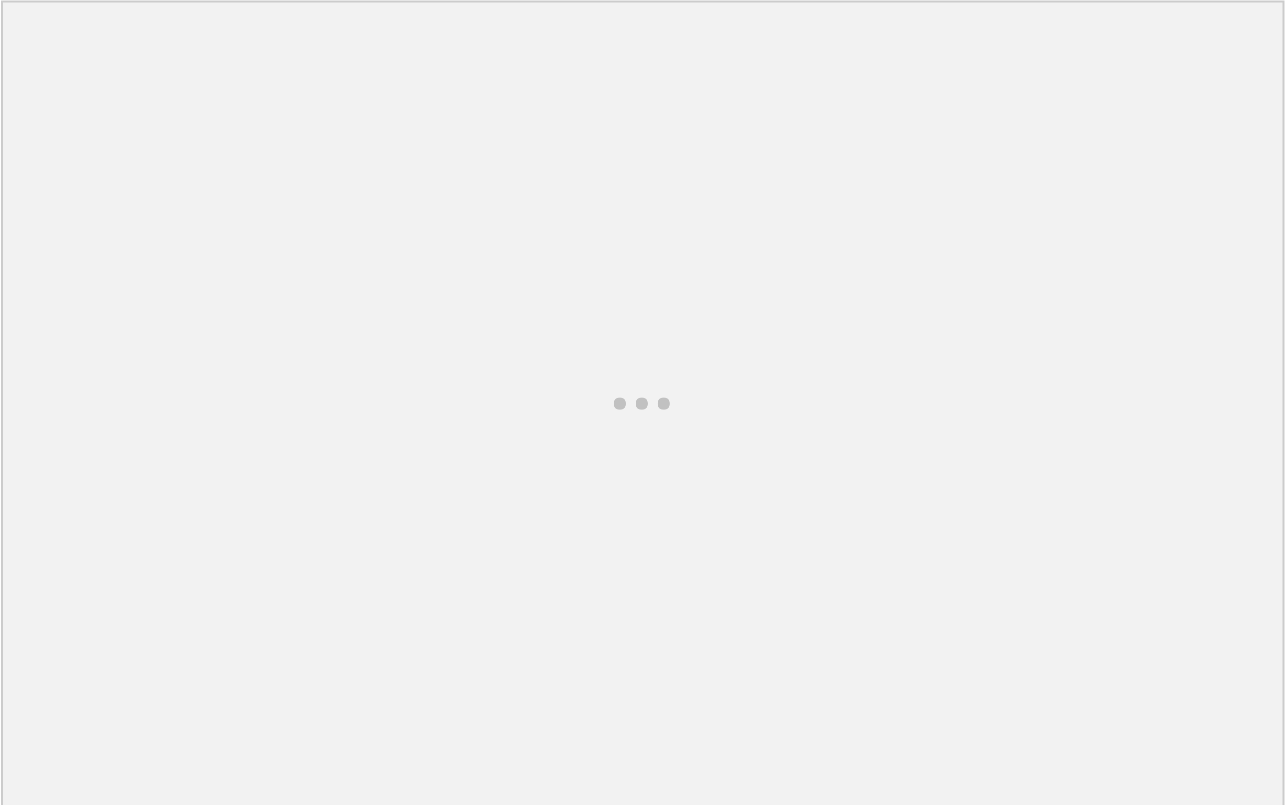
Figure SEQ "Figure" * ARABIC 8, Broadcast_displays_on command forwarding

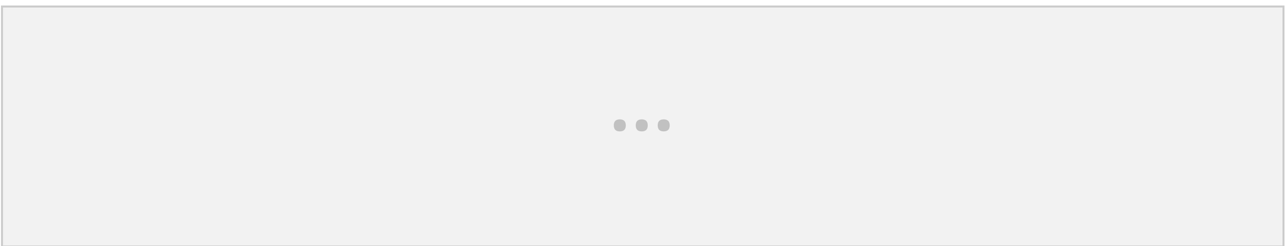
6.4.2. With reply

If a broadcast request is sent out on the SPBUS and replies are expected, then every slave will **first** process the request and send back a reply **before** sending the broadcast request to the next SPBD in the daisy-chain (Figure 9, Broadcast with replies).

A more specific example is shown in figures 10 to 12 where Figure 10 shows a PDU receiving the 'Broadcast_scan' command from SPBUS connection 1, Figure 11 shows reply back over connection 1 and propagation of 'Broadcast_scan' to next PDU over connection 2, and Figure 12 shows the reply of the next PDU propagated back to the master.

The SPBD that requested the broadcast command will receive a reply from every SPBD on the bus (Figure 5, Broadcast_scan command with 4 replies shows a broadcast request on the bus with 4 replies, each with an interval of roughly 25ms).





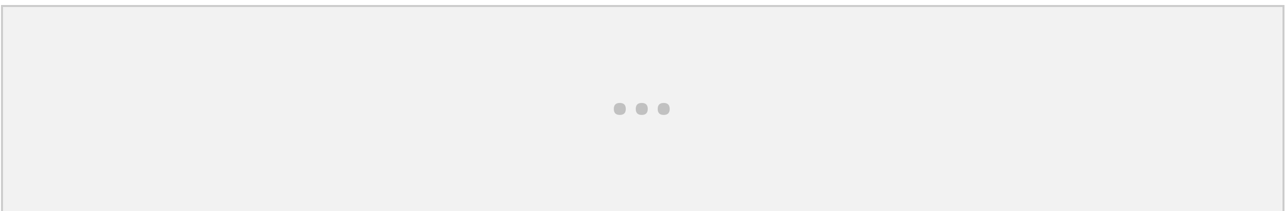


Figure SEQ "Figure" * ARABIC 11, Broadcast_scan reply from next PDU, propagated back to master

6.5. Message frame construction examples

This chapter shows a few example constructions of typical message frames. Note that "With little-endian" describes the packet as expected over the SPBUS.

6.5.1. Read firmware version of PDU with unit address 42, transaction identifier 1, and FW version 233 (2.33)

- Request:
  - [Packet_start.STX][Command.Read_register_layer_1][Address][Identifier][Register_start][Register_length][CRC][Packet_end.ETX]
  - [0x02][0x01][0x002A][0x0001][0x0066][0x0002][0x630D][0x03]
  - Without little-endian: 0x0201002A000100660002630D03
  - With little-endian: 0x02012A000100660002000D6303
- Reply (in this case successful):
  - [Packet_start.ACK][Command.Read_register_layer_1][Address][Identifier][Register_start][Register_length][Data][CRC][Packet_end.ETX]
  - [0x06][0x01][0x002A][0x0001][0x0066][0x0002][0x00E9][0x282A][0x03]
  - Without little-endian: 0x0601002A00010066000200E9282A03
  - With little-endian: 0x06012A00010066000200E9002A2803

6.5.2. Scan SPBUS network which consists of 2 SPBDs, one with unit address 8, and the other with unit address 87

- Request:
  - [Packet_start.STX][Command.Broadcast_scan][CRC][Packet_end.ETX]
  - [0x02][0x90][0xF8D4][0x03]
  - Without little-endian: 0x0290F8D403
  - With little-endian: 0x0290D4F803
- Reply:
  - [Packet_start.ACK][Command.Broadcast_scan][Address][Hardware_id][CRC][Packet_end.ETX]
    1. SPBD with unit address 8
       - [0x06][0x90][0x0008][0x7AB016400000][0x5FCB][0x03]
       - Without little-endian: 0x069000087AB0164000005FCB03
       - With little-endian: 0x06900800B07A40160000CB5F03
    2. SPBD with unit address 87
       - [0x06][0x90][0x0057][0xABED14E10000][0x1CAB][0x03]
       - Without little-endian: 0x06900057ABED14E100001CAB03
       - With little-endian: 0x06905700EDABE1140000AB1C03

7. FIRMWARE UPGRADE

Upgrading firmware can be done over SPBUS either per SPBD or in a whole batch where upgrading occurs for every device within the SPBUS network, assuming the firmware upgrade is supported on the SPBD. This chapter describes how an upgrade can be initiated. Binary firmware files can be found on the Schleifenbauer download page.

7.1. Firmware upload preparations

Uploading a new version of the firmware is done by first writing firmware upgrade information to the upload info register, '*upvers*' (register 10000). The upload info packet that has to be written to the '*upvers*' register looks as follows:

[version][checksum][crc][numberOfBlocks][size]

Note that every block between square brackets must be in little-endian regardless of size (i.e. 32-bit values are written in 32-bit style little-endian, 16-bit values are written in 16-bit style little-endian). Please refer to the SPDM for more information on registers.

The majority of the firmware upgrade information necessary for uploading is given in the firmware filename and delimited by dashes.

The firmware filename format is as follows:

- oem-version-checksum-crc-extra.bin.

From the firmware's filename the 'version', 'checksum', and 'crc' attributes directly correspond to the upload info packet's [version], [checksum], and [crc] fields (little-endian). The remaining [size] and [numberOfBlocks] fields can be retrieved as follows:

size = sizeof("<oem>-<version>-<checksum>-<crc>-<extra>.bin")

numberOfBlocks = size / 256

7.2. Firmware data upload

Uploading firmware data has to be done consecutively, in packets of 258-bytes. Each upload packet must contain an updated block number (2-bytes) and updated block data (256-bytes). This upload packet has to be written to the '*upblnr*' register (register number 10100). The format looks as follows:

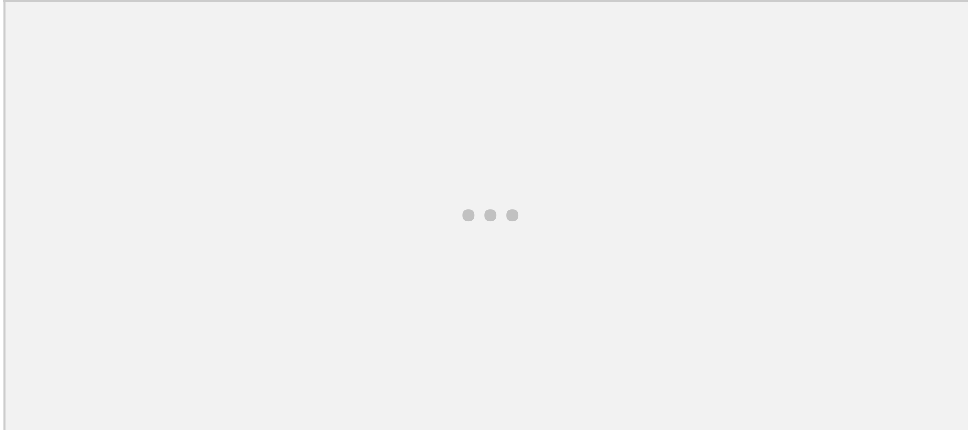
[dataBlockNumber][dataBlock]

Note that the [dataBlockNumber] block must be in little-endian.

Finalizing firmware data upload is done by rebooting the SPBD. This must happen after all blocks are uploaded and is done by writing to the '*rsboot*' register (see SPDM).

8. SPBUS PACKET OVER ETHERNET (Client <-> Master)

With new revisions of some SPBDs, like the hPDU and DPM3, it is possible to communicate over Ethernet. The protocol used is an extension of the SPBUS protocol as it uses the SPBUS protocol message frame over TCP/IP (Transfer Communication Protocol/Internet Protocol) with some additional data and encryption. This extension is referred to as IPAPI (Internet Protocol Application Programming Interface).



Even if a SPBD supports IPAPI, it has to be enabled. Support for IPAPI can be verified on either the web interface or the SPBD display. On the web interface, it is listed in the 'API' section, under the 'Interfaces' tab (see Figure 13, Web interface IPAPI configuration screen).

On the SPBD display it can be verified by going to the 'IP INTERFACES' tab. If the 'API' variable shows value 'off', then it is disabled. Otherwise the communication protocol is shown together with the port used (**the default port used is 7783, and the default encryption key used is '0000000000000000'**).

8.1. Connection

To use the IPAPI, a TCP connection must be made with the SPBD. If a SPBD supports IPAPI and the IPAPI interface is enabled, then the SPBD hosts a server from its IP address on the port used (as mentioned, the default port used is port 7783). The port used can be verified by going to the 'IP INTERFACES' tab on the SPBD's display and looking at the 'API' value, or on the web interface as described in the introduction (see Figure 13, Web interface IPAPI configuration screen).

The IP address of a SPBD can be retrieved using either the web interface, or the SPBD display. For the web interface, this can be found in the 'Interfaces' tab. For the SPBD display, this can be found in the 'IP LINK' tab of the display.

8.2. Arc4

Arc4 encryption is used to create the new payload for transmission over TCP/IP. The private key of arc4 can be set in the web interface below the 'API' section in the 'interfaces' tab (as mentioned, the default key used is '0000000000000000'). The arc4 encryption algorithm is also its decryption algorithm, meaning that decryption of IPAPI message is done using the arc4 encryption function.

8.3. Packet construction

The packets for SPBUS over TCP/IP consists of a normal SPBUS command message frame as described in previous chapters, but with additional construction steps, which includes a checksum and arc4 encryption. The steps to construct a IPAPI message is as follows:

1. Make a regular SPBUS frame (with the little-endian encoding of the fields) as described in previous chapters with the required command(s).

2. The first 4 digits of the encryption key are converted to their ASCII character value equivalent and prefixed to the SPBUS message frame, creating a new payload.
3. Sum all the byte values of the new payload to create the checksum and use the checksum as suffix of the new payload as a 4byte, big-endian style value.
4. Encrypt the current payload with arc4 using the encryption key as found on the web interface.
5. Take the size of the payload in byte and prefix it to the payload as a 2byte value (big-endian style).
6. Prefix the current payload with the string 'SAPI'.

To decrypt a response, the steps are to be done in reverse order and some verification should be used (verify checksum). Note that arc4 encryption and decryption are involute functions; therefore, the encryption function can be used to decrypt the payload.

8.3.1. Example packet construction

For the IPAPI packet construction a similar SPBUS message frame will be used as described in the 'Message frame construction examples' chapter. The SPBUS message will be a read firmware version register of a PDU with unit address 76, firmware version 236, key '0000000000000000', and transaction identifier 1.

SPBUS message frame: [0x02][0x01][0x4C00][0x0100][0x6600][0x0200][0x7E6E][0x03]

1. SPBUS message frame (non-delimited): 0x02014C000100660002007E6E03
2. '0' = 0x30

new payload = 0x3030303002014C000100660002007E6E03

3. Checksum = 0x30 + 0x30 + 0x30 + 0x30 + 0x02 + 0x01 + 0x4C + 0x00 + 0x01 + 0x00 + 0x66 + 0x00 + 0x02 + 0x00 + 0x7E + 0x6E + 0x03 = 615 = 0x0267.

New payload = 0x3030303002014C000100660002007E6E0300000267

4. key = '0000000000000000'

new payload = 0xB84F96B0C17A982735FF8474F36620D484062CEC32

5. size (in bytes) = 21 = 0x15

new payload = 0x0015B896B0C17A982735FF8474F36620D484062CEC32

6. Prefix = 'SAPI' = [0x53][0x41][0x50][0x49] = 0x53415049

final payload = 0x534150490015B84F96B0C17A982735FF8474F36620D484062CEC32