Specification

KNX BAOS Binary Protocol

BAOS Binary Services Version 2.x for

KNX BAOS Module 830 / 832
KNX BAOS Module 830.1 *secure*
KNX BAOS Module 838 kBerry
KNX BAOS Module RF 840

KNX USB Interface 312 / 332
KNX USB Module 322
KNX USB Module 323 *secure*
KNX USB Interface Stick 333 *secure*

KNX IP BAOS 771 / 772
KNX IP BAOS 773 / 774
KNX IP BAOS 774.1 *secure*
KNX IP BAOS 777

kTux: KNX Stack for Linux

WEINZIERL ENGINEERING GmbH
Achatz 3-4
84508 Burgkirchen an der Alz
GERMANY

Tel.: +49 8677 / 916 36 – 0
E-Mail: info@weinzierl.de
Web: www.weinzierl.de

Protocol version supported by devices

Device	Protocol version
KNX BAOS Module 830 / 832 KNX BAOS Module 838 kBerry KNX BAOS Module RF 840 KNX USB Interface 312 / 332 KNX USB Module 322 KNX IP BAOS 771 / 772 KNX IP BAOS 773 / 774	2.0
KNX IP BAOS 777	2.1 (777 specific)
KNX BAOS Module 830.1 secure KNX USB Module 323 secure KNX USB Interface Stick 333 secure KNX IP BAOS 774.1 secure kTux: KNX Stack for Linux	2.2 (Secure extensions)

Content

1 What is an ObjectServer?

The KNX Standard defines a protocol stack according to the ISO/OSI reference model. While standard KNX interfaces grant an access to the KNX network on data link layer, an object server offers an access on application layer. So a client can talk to KNX data points without knowing telegram format or addresses used in the installation.

The ObjectServer is a hardware or software component, which is connected to the KNX bus and offers to the client a set of the defined “objects”. These objects are the server properties (called “items”), KNX datapoints (known as “communication objects” or as “group objects”) and KNX configuration parameters (Figure 1). The communication between server and clients is based on the ObjectServer protocol that is encapsulated into some other communication protocol (e.g. FT1.2, IP) depending on the interface type used.

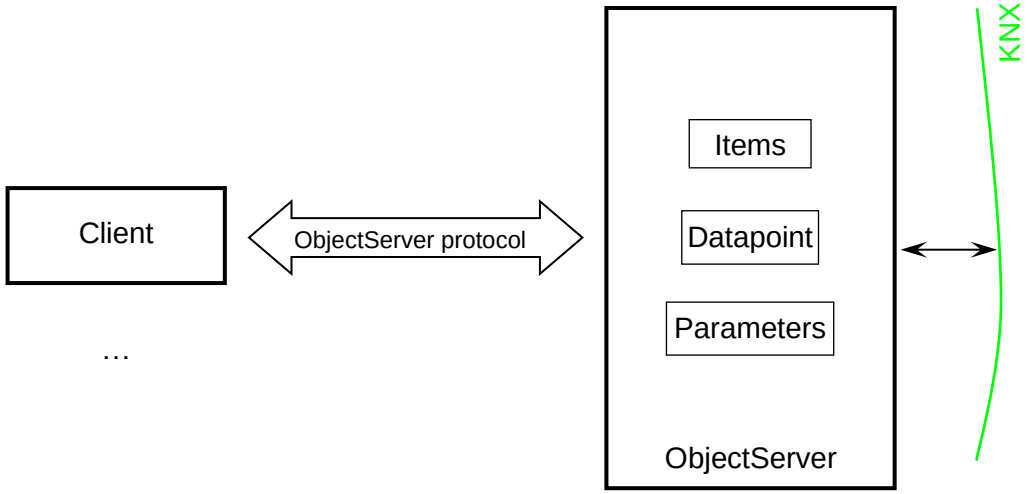

Figure 1: Communication between ObjectServer and Client

2 Encapsulation of the ObjectServer protocol

The ObjectServer protocol has been defined to achieve the whole functionality on small embedded platforms and on data channels with limited bandwidth. As a consequence of this, the protocol is kept very slim and has no connection management, like connection establishment, user authorization, etc. Therefore, it is advisable and highly recommended to encapsulate the ObjectServer protocol into some existing transport protocol to get a useful solution for an easy access to the KNX datapoints and to the KNX bus.

Depending on the interface type the BAOS protocol is encapsulated in:

- Serial: FT1.2 frames
- USB: HID reports
- IP: UDP or TCP frames

2.1 Getting started

To get familiar with the BAOS architecture we recommend to try the BAOS protocol using the free version of our bus monitor and analyzer tool Net'n Node (Figure 2). The integrated BAOS view supports serial, USB and IP connections. Net'n Node can also send and receive KNX telegrams in parallel. So, it shows the relation between BAOS services and KNX communication.

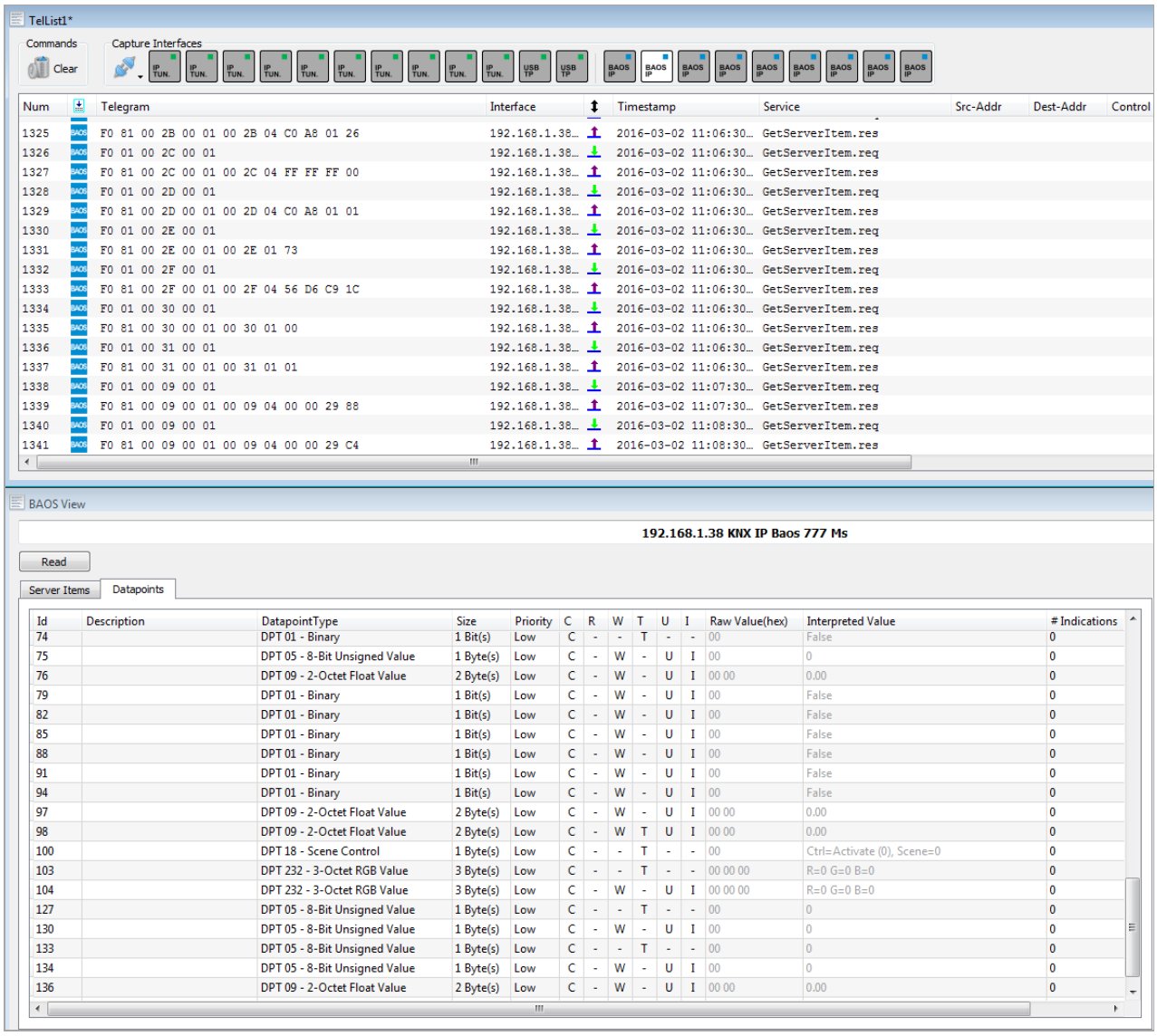

Figure 2: Net'n Node telegram view with BAOS data points

For the serial modules, a starter kit is available which allows to connect the BAOS Modules to a PC using a virtual comport. To implement a client application, a demo project with source for the starter kit is available at our web page.

3 BAOS core protocol

The communication between the server and the client is based on an ObjectServer protocol which consists of requests sent by a client and server responses. Indications inform the client about changes of a datapoint's value, which is sent asynchronously from the server to the client.

The BAOS core protocol is used in all BAOS modules or devices. Depending on the interface type (serial, USB, IP) corresponding frames are used as transport protocols.

The following services are defined:

- GetServerItem.Req/Res
- SetServerItem.Req/Res
- ServerItem.Ind
- GetDatapointDescription.Req/Res
- GetDescriptionString.Req/Res
- GetDatapointValue.Req/Res
- DatapointValue.Ind
- SetDatapointValue.Req/Res
- GetParameterByte.Req/Res
- SetParameterByte.Req/Res

3.1 GetServerItem.Req

This request is sent by the client to get one or more server items (properties). The data packet consists of six bytes:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x01	Subservice code
+2	StartItem	2		ID of first item
+4	NumberOfItems	2		Maximum number of items to return

The server sends a response to the client containing the values of supported items in the range of [StartItem … StartItem+NumberOfItems-1].

See Appendix A for the item IDs.

3.2 GetServerItem.Res

This response is sent by the server as answer to the GetServerItem request. In case of an error, the server sends a negative response using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x81	Subservice code
+2	StartItem	2		ID of bad item
+4	NumberOfItems	2	0	
+6	ErrorCode	1		Error code

See Appendix B for the error codes.

If the request has been successfully processed by the server it sends a positive response to the client using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x81	Subservice code
+2	StartItem	2		As in request
+4	NumberOfItems	2		Number of items in this response
+6	First item ID	2		ID of first item
+8	First item data length	1		Data length of first item
+9	First item data	1-255		Data of first item
…	…		…	…
+N-3	Last item ID	2		ID of last item
+N-1	Last item data length	1		Data length of last item
+N	Last item data	1-255		Data of last item

Unsupported items are silently ignored. They do not appear in the response.

3.3 SetServerItem.Req

This request is sent by the client to set a new value of one or more server items.

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x02	Subservice code
+2	StartItem	2		ID of first item to set
+4	NumberOfItems	2		Number of items in this request
+6	First item ID	2		ID of first item
+8	First item data length	1		Data length of first item
+9	First item data	1-255		Data of first item
…	…		…	…
+N-3	Last item ID	2		ID of last item
+N-1	Last item data length	1		Data length of last item
+N	Last item data	1-255		Data of last item

The server sends a response to the client whether the setting was successful or not.

See Appendix A for the item IDs.

3.4 SetServerItem.Res

This response is sent by the server as answer to the SetServerItem request. In case of an error, the server sends a negative response using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x82	Subservice code
+2	StartItem	2		ID of bad item
+4	NumberOfItems	2	0	
+6	ErrorCode	1		Error code

Note: No item is set in case of an error. Even if the previous items would do.

See Appendix B for the error codes.

If a request has been successfully processed by the server it sends a positive response to the client using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x82	Subservice code
+2	StartItem	2		As in request
+4	NumberOfItems	2	0	
+6	ErrorCode	1	0	

3.5 ServerItem.Ind

This indication is sent asynchronously by the server if one or more server items have been changed. Not all server items support indications. See Appendix A. ServerItem.Ind has the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0xC2	Subservice code
+2	StartItem	2		ID of first item
+4	NumberOfItems	2		Number of items in this indication
+6	First item ID	2		ID of first item
+8	First item length	1		Data length of first item
+9	First item data	1-255		Data of first item
…	…		…	…
+N-3	Last item ID	2		ID of last item
+N-1	Last item data length	1		Data length of last item
+N	Last item data	1-255		Data of last item

3.6 GetDatapointDescription.Req

This request is sent by the client to get the description of one or more datapoints. The data packet consists of six bytes:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x03	Subservice code
+2	StartDatapoint	2		ID of first datapoint
+4	NumberOfDatapoints	2		Maximum number of descriptions to return

The server sends a response to the client containing the descriptions of datapoints in the range of [StartDatapoint … StartDatapoint + NumberOfDatapoints - 1].

3.7 GetDatapointDescription.Res

This response is sent by the server as answer to the GetDatapointDescription request. In case of an error, the server sends a negative response using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x83	Subservice code
+2	StartDatapoint	2		As in request
+4	NumberOfDatapoints	2	0	
+6	ErrorCode	1		Error code

See Appendix B for the error codes.

If a request has been successfully processed by the server it sends a positive response to the client using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x83	Subservice code
+2	StartDatapoint	2		As in request
+4	NumberOfDatapoints	2		Number of descriptions in this response
+6	First DP ID	2		ID of first datapoint
+8	First DP value type	1		Value type of first datapoint
+9	First DP config flags	1		Configuration flags of first datapoint
+10	First DP DPT	1		Datapoint type (DPT) of first datapoint
…	…		…	…
+N-4	Last DP ID	2		ID of last datapoint
+N-2	Last DP value type	1		Value type of last datapoint
+N-1	Last DP config flags	1		Configuration flags of last datapoint
+N	Last DP DPT	1		Datapoint type (DPT) of last datapoint

See Appendix C for the value types and 0 for the datapoint types.

Format of configuration flags:

Bit	Meaning	Value	Description
1 - 0	Transmit priority	00	System priority
		01	High priority
		10	Alarm priority
		11	Low priority
2	Datapoint communication	0	Disabled
		1	Enabled
3	Read from bus	0	Disabled
		1	Enabled
4	Write from bus	0	Disabled
		1	Enabled
5	Read on init	0	Disabled
		1	Enabled
6	Transmit to bus	0	Disabled
		1	Enabled
7	Update on response	0	Disabled
		1	Enabled

3.8 GetDescriptionString.Req

This request is sent by the client to get the human-readable description string of one or more datapoints. The data packet consists of six bytes:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x04	Subservice code
+2	StartString	2		ID of first datapoint string
+4	NumberOfStrings	2		Maximum number of strings to return

The server sends a response to the client containing the strings of datapoints in the range of [StartString … StartString+NumberOfStrings-1].

Note: This service is not available on some servers.

3.9 GetDescriptionString.Res

This response is sent by the server as answer to the GetDescriptionString request. In case of an error, the server sends a negative response using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x84	Subservice code
+2	StartString	2		As in request
+4	NumberOfStrings	2	0	
+6	ErrorCode	1		Error code

See Appendix B for the error codes.

If a request has been successfully processed by the server it sends a positive response to the client using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x84	Subservice code
+2	StartString	2		As in request
+4	NumberOfStrings	2		Number of strings in this response
+6	StrLen of first DP	2		Length of first DP description string
+8	First DP description string	StrLen		Description string of first datapoint
…	…		…	…
+N-2	StrLen of last DP	2		Length of last DP description string
+N	Last DP description string	StrLen		Description string of last datapoint

The datapoint description strings are not null terminated. The length of each datapoint description string is given by the corresponding StrLen.

3.10 GetDatapointValue.Req

This request is sent by the client to get the value of one or more datapoints. The data packet consists of seven bytes:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x05	Subservice code
+2	StartDatapoint	2		ID of first datapoint
+4	NumberOfDatapoints	2		Maximum number of data-points to return
+6	Filter	1		Criteria which data points shall be retrieved

The filter criteria are coded as follows:

Value	Description
0x00	Get all datapoint values
0x01	Get only valid datapoint values
0x02	Get only updated datapoint values
0x03 … 0xFF	Reserved

The server sends a response to the client containing the values of datapoints in the range of [StartDatapoint … StartDatapoint+NumberOfDatapoints-1].

3.11 GetDatapointValue.Res

This response is sent by the server as answer to the GetDatapointValue request. In case of an error, the server sends a negative response using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x85	Subservice code
+2	StartDatapoint	2		Index of the bad datapoint
+4	NumberOfDatapoints	2	0	
+6	ErrorCode	1		Error code

See Appendix B for the error codes.

If a request has been successfully processed by the server it sends a positive response to the client using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x85	Subservice code
+2	StartDatapoint	2		As in request
+4	NumberOfDatapoints	2		Number of datapoints in this response
+6	First DP ID	2		ID of first datapoint
+8	First DP state	1		State byte of first datapoint
+9	First DP length	1		Length of first datapoint
+10	First DP value	1-14		Value of first datapoint
…	…		…	…
+N-4	Last DP ID	2		ID of last datapoint
+N-2	Last DP state	1		State byte of last datapoint
+N-1	Last DP length	1		Length byte of last datapoint
+N	Last DP value	1-14		Value of last datapoint

Format of state byte:

Bit	Meaning	Value	Description
7	Reserved	0	Reserved
6	Reserved	0	Reserved
5	Reserved	0	Reserved
4	Valid flag	0	Object value is unknown
		1	Object has already been received
3	Update flag	0	Value is not updated
		1	Value is updated from bus
2	Read request flag	0	Write request should be sent
		1	Read request should be sent
1 – 0	Transmission status	00	Idle/OK
		01	Idle/error
		10	Transmission in progress
		11	Transmission request

A length of less than one byte (e.g. DPT 3: 4-bits, Dimming, Blinds) for KNX datapoints are coded as follows:

	7	6	5	4	3	2	1	0
1-bit:	0	0	0	0	0	0	0	x
2-bits:	0	0	0	0	0	0	x	x
3-bits:	0	0	0	0	0	x	x	x
4-bits:	0	0	0	0	x	x	x	x
5-bits:	0	0	0	x	x	x	x	x
6-bits:	0	0	x	x	x	x	x	x
7-bits:	0	x	x	x	x	x	x	x

3.12 DatapointValue.Ind

This indication is sent asynchronously by the server if a value of one or more datapoints have been changed. It has the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0xC1	Subservice code
+2	StartDatapoint	2		ID of first datapoint
+4	NumberOfDatapoints	2		Number of datapoints in this indication
+6	First DP ID	2		ID of first datapoint
+8	First DP state	1		State byte of first datapoint
+9	First DP length	1		Length of first datapoint
+10	First DP value	1-14		Value of first datapoint
…	…		…	…
+N-4	Last DP ID	2		ID of last datapoint
+N-2	Last DP state	1		State byte of last datapoint
+N-1	Last DP length	1		Length byte of last datapoint
+N	Last DP value	1-14		Value of last datapoint

Format of the state byte see the description of the GetDatapointValue response.

See 0 for the datapoint types.

3.13 SetDatapointValue.Req

This request is sent by the client to set the new value of one or more datapoints or to request/transmit the new value on the bus. It can be used to clear the transmission state of the datapoint, also.

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x06	Subservice code
+2	StartDatapoint	2		ID of first datapoint to set
+4	NumberOfDatapoints	2		Number of datapoints to set
+6	First DP ID	2		ID of first datapoint
+8	First DP command	1		Command byte of first datapoint
+9	First DP length	1		Length byte of first datapoint
+10	First DP value	1-14		Value of first datapoint
…	…		…	…
+N-4	Last DP ID	2		ID of last datapoint
+N-2	Last DP command	1		Command byte of last datapoint
+N-1	Last DP length	1		Length byte of last datapoint
+N	Last DP value	1-14		Value of last datapoint

Format of command byte:

Bit	Meaning	Value	Description
7-4	Reserved	0000	Reserved
3-0	Datapoint command	0000	No command
		0001	Set new value
		0010	Send value on bus
		0011	Set new value and send on bus
		0100	Read new value via bus
		0101	Clear datapoint transmission state
		0110	Reserved
		…	
		1111	Reserved

The datapoint value length must match the value length, which is selected by the ETS project database.

The value length “zero” is acceptable and means: “no value in frame”. It can be used for instance to clear the transmission state of the datapoint or to send the current datapoint value on the bus.

3.14 SetDatapointValue.Res

This response is sent by the server as answer to the SetDatapointValue request. In case of an error, the server sends a negative response using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x86	Subservice code
+2	StartDatapoint	2		Index of bad datapoint
+4	NumberOfDatapoints	2	0	
+6	ErrorCode	1		Error code

Note: No datapoint is set in case of an error. Even if the previous datapoints would do.

See Appendix B for the error codes.

If a request can be successfully processed by the server, it sends a positive response to the client using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x86	Subservice code
+2	StartDatapoint	2		As in request
+4	NumberOfDatapoints	2	0	
+6	ErrorCode	1	0	

3.15 GetParameterByte.Req

This request is sent by the client to get/read one or more parameter bytes. In the KNX system parameters are used for device settings that typically are not changed during runtime operation. Parameters have to be defined by the device manufacturer in the ETS product entry using a tool called KNX Manufacturer Tool (MT), available from KNX Association.

The installation specific settings of the parameters can be done in the ETS parameter dialog by the installer. A parameter can have a size of 1 bit up to several bytes. The BAOS service GetParameterByte.Req allows an access to parameter values on byte level.

The data packet of the GetParameterByte request consists of six bytes:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x07	Subservice code
+2	StartByte	2		Index of first byte
+4	NumberOfBytes	2		Maximum number of bytes to return

The server sends a response to the client containing the values of the parameters in the range of [StartByte … StartByte+NumberOfBytes-1].

3.16 GetParameterByte.Res

This response is sent by the server as answer to the GetParameterByte request. In case of an error, the server sends a negative response using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x87	Subservice code
+2	StartByte	2		Index of the bad parameter
+4	NumberOfBytes	2	0	
+6	ErrorCode	1		Error code

See Appendix B for the error codes.

If the request has been successfully processed by the server it sends a positive response to the client using following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x87	Subservice code
+2	StartByte	2		As in request
+4	NumberOfBytes	2		Number of bytes in this response
+6	First byte	1		First parameter byte
…	…		…	…
+N	Last byte	1		Last parameter byte

3.17 SetParameterByte.Req

This request is sent by the client to set (write) one or more the parameter bytes. Starting with version 5 ETS is able to read parameter values modified by the device.

The data packet of the SetParameterByte request has the following structure:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x08	Subservice code
+2	StartByte	2		Index of first byte
+4	NumberOfBytes	2		Number of bytes
+6	First byte	1		First parameter byte
…	…		…	…
+N	Last byte	1		Last parameter byte

When all the parameter bytes have been written by the client, it can send a request without data to the server to indicate that the written data can be flushed to the non-volatile memory:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x08	Subservice code
+2	StartByte	2	0x0000	Index of first byte
+4	NumberOfBytes	2	0x0000	Number of bytes

The server sends a response to the client with the result of the operation.

Note: Implemented in devices supporting protocol version 2.2 and above.

3.18 SetParameterByte.Res

This response is sent by the server as answer to the SetParameterByte request. In case of an error, the server sends a negative response using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x88	Subservice code
+2	StartByte	2		Index of the bad parameter
+4	NumberOfBytes	2	0	
+6	ErrorCode	1		Error code

See Appendix B for the error codes.

If a request has been successfully processed by the server it sends a positive response to the client using the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x88	Subservice code
+2	StartByte	2		As in request
+4	NumberOfBytes	2	0	
+6	ErrorCode	1	0	

Note: Implemented in devices supporting protocol version 2.2 and above.

4 BAOS Protocol via Serial FT1.2

The encapsulation of the ObjectServer protocol into FT1.2 (also known as PEI type 10) protocol is simple and is shown in Figure .

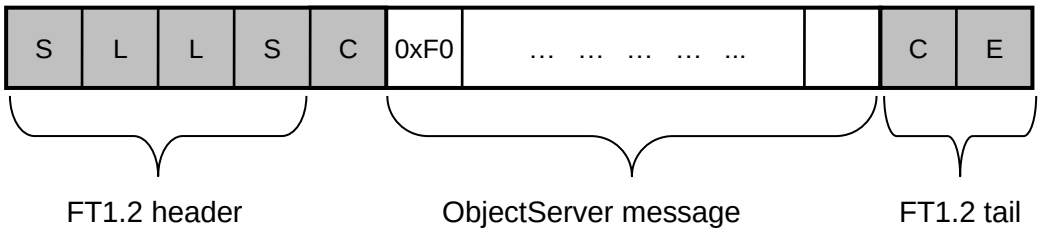

Figure 3: Integration of the ObjectServer message into a FT1.2 frame

4.1 FT1.2 protocol

The FT1.2 transmission protocol is based on the international standard IEC 870-5-1 and IEC 870-5-2 (DIN 19244). The used hardware interface for the transmission is the Universal Asynchronous Receiver Transmitter (UART). The frame format of the FT1.2 protocol is fixed to 8 data bits, 1 stop bit and even parity bit. The default communication speed is 19200 Baud.

Communication procedure

The typical communication procedure between the host and the ObjectServer is shown in Figure 4.

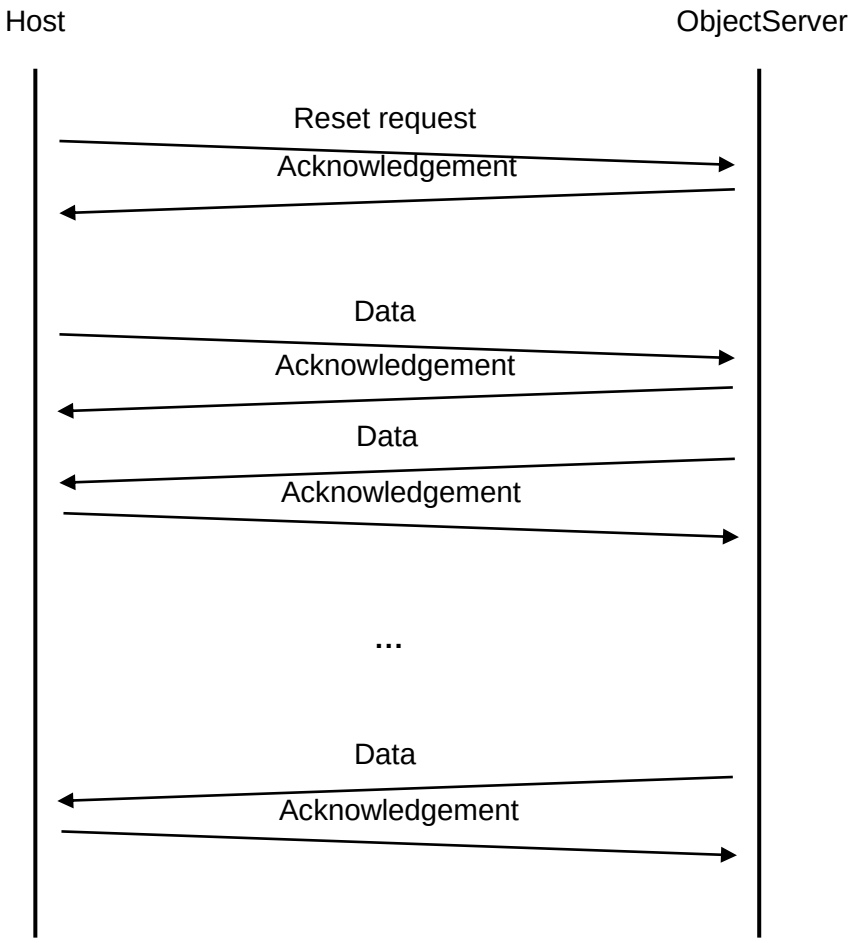

Figure 4: Typical communication procedure

Below is an example of the communication between the host and the ObjectServer.

Frame format

The FT1.2 protocol defines three frame types.

The first one is the positive acknowledgement frame and consists of only one byte of the value 0xE5.

The second frame type is 4 bytes long and is used for the reset request and reset indication messages (Figure 5).

Reset.Req:	0x10	0x40	0x40	0x16		Reset.Ind:	0x10	0xC0	0xC0	0x16

Figure 5: Structure of the Reset.Req and Reset.Ind frames

The third frame type has a variable length and is used for the data message. The frame structure is shown in Figure 6.

0x68	**L**	**L**	0x68	**CR**	data	**C**	0x16

Figure 6: Structure of the data message

The both fields **L** contain the length twice of the data in this frame + 1 (for the control byte **CR**).

The field **CR** is the control byte of the frame. Its value either 0x73 for odd frames (after reset request sent by the host) and 0x53 for even frames. In the opposite direction (from ObjectServer to host) the control byte is 0xF3 for odd frames and 0xD3 for even frames.

The field **C** contains the checksum of the frame and is the arithmetic sum disregarding overflows (modulo 256) over all data and the control byte.

Communication example

Host -> ObjectServer: Reset Request

{0x10 0x40 0x40 0x16}

ObjectServer -> Client: Acknowledgement

{0xE5}

Host -> ObjectServer: GetServerItem.Req (Firmware version)

{0x68 0x07 0x07 0x68 0x73 0xF0 0x01 0x00 0x03 0x00 0x01 0x68 0x16}

ObjectServer -> Client: Acknowledgement

{0xE5}

ObjectServer -> Client: GetServerItem.Res (Firmare version)

{0x68 0x0B 0x0B 0x68 0xF3 0xF0 0x81 0x00 0x03 0x00 0x01 0x00 0x03 0x01 0x10 0x7C 0x16}

Host -> ObjectServer: Acknowledgement

{0xE5}

Host -> ObjectServer: GetServerItem.Req (Serial number)

{0x68 0x07 0x07 0x68 0x53 0xF0 0x01 0x00 0x08 0x00 0x01 0x4D 0x16}

ObjectServer -> Client: Acknowledgement

{0xE5}

ObjectServer -> Client: GetServerItem.Res (Serial number)

{0x68 0x0F 0x0F 0x68 0xD3 0xF0 0x81 0x00 0x08 0x00 0x01 0x00 0x08 0x06 0x00 0xC5 0x08 0x02 0x00 0x00 0x2A 0x16}

Host -> ObjectServer: Acknowledgement

{0xE5}

4.2 Host protocol security for serial or USB interface

Note: Implemented in devices supporting protocol version 2.2 and above.

4.2.1 Secure frames

The security extension of the ObjectServer protocol defines two new message frames:

- ObjectServer secure wrapper frame
- ObjectServer secure failure frame
- ObjectServer secure Sync request frame
- ObjectServer secure Sync response frame

4.2.1.1 ObjectServer secure wrapper frame

The ObjectServer secure wrapper frame is used to transport the encrypted ObjectServer service. This could be any standard ObjectServer request, response or indication.

The ObjectServer secure wrapper frame has variable length and following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xC0	Main service code
+1	Sequence counter	6		Sequence counter
+7	Encrypted data	1-240		Encrypted standard ObjectServer service
…	…		…	…
+N-4	Encrypted MAC	4		Encrypted message authentication code (MAC)

4.2.1.2 ObjectServer secure failure frame

This response is sent by the server to the client, if any security failure is detected:

Offset	Field	Size	Value	Description
+0	MainService	1	0xC1	Main service code
+1	Failure code	1		Failure code

Defined failure codes:

Code	Description
0xCE	*ObjectServer security violation*: Server rejects the received frame for security reasons.

4.2.1.3 ObjectServer secure Sync request frame

The ObjectServer secure Sync request frame is used to allow the lost client to obtain information about the ObjectServer's activated/deactivated security sequence counters or their values.

The ObjectServer secure Sync request frame has fixed length and following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xC2	Main service code
+1	Sequence counter	6		Client sequence counter (set to 0, if unknown)
+7	Encrypted request challenge	6		Encrypted random value
+13	Encrypted MAC	4		Encrypted message authentication code (MAC)

Note: the client should generate a new challenge for every request.

4.2.1.4 ObjectServer secure Sync response frame

This response is sent by the server to answer the ObjectServer secure Sync request when a request has been successfully processed by the server:

Offset	Field	Size	Value	Description
+0	MainService	1	0xC3	Main service code
+1	Response challenge	6		= request challenge XOR random value
+7	Encrypted receive counter	6		BAOS receive counter value of the server
+13	Encrypted send counter	6		BAOS send counter value of the server
+19	Encrypted MAC	4		Encrypted message authentication code (MAC)

4.2.2 ObjectServer data encryption

The ObjectServer data encryption is likely the KNX data security based on the standard AES counter mode with CBC-MAC (AES-CCM). For the details and background can be referenced to the (1) KNX specification, Chapter 3/3/7 “Application Layer” or (2) RFC3610 “Counter with CBC-MAC (CCM)”.

The block diagram of the AES-CCM algorithm is presented in the figure 1.

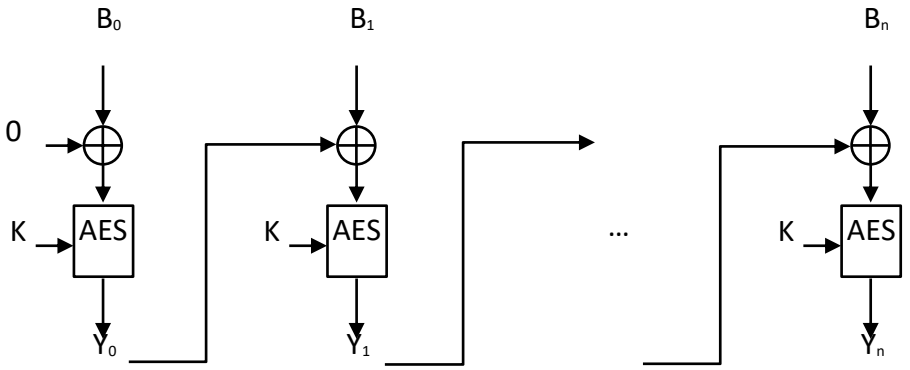

Figure 7: Block diagram AES-CCM

The following chapters do not describe the AES-CCM algorithm itself, but only the AES-CCM parameters specific for the ObjectServer security extension. For the definition of the AES-CCM, please refer to (2) RFC3610 “Counter with CBC-MAC (CCM)”.

4.2.2.1 Encryption of the ObjectServer secure wrapper frame

Following steps are required to encrypt the plain ObjectServer service:

1. Generate first block B_0 as follow:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
SeqNum						**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**8**	**L**

SeqNum is the next BAOS Send Counter

L is the length of the plain ObjectServer service

2. Generate blocks $B_1..B_n$ filled with plain ObjectServer service, padded with zeros to the next block end
3. Encrypt all B_n blocks with AES and the BAOS Client Key
4. Save MAC = four first bytes of the Y_n block
5. Generate Ctr_j block as follow:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
SeqNum						**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**9**	**J**

j=[0..N] is the block number

6. Encrypt all Ctr_j blocks with AES and the BAOS Client Key

7. XOR first four bytes of the Ctr_0 with stored MAC to get the encrypted MAC

8. XOR the rest of the Ctr_n with the plain ObjectServer service to get the encrypted ObjectServer service

9. Compose the BAOS secure wrapper frame as defined in chapter “4.2.1.1 ObjectServer secure wrapper frame”

Stepwise example of the message encryption is shown in Appendix E.

4.2.2.2 Decryption of the ObjectServer secure wrapper frame

Following steps are required to decrypt the encrypted ObjectServer service:

1. Generate Ctr_j block as follow:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
SeqNum						**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**9**	**J**

   SeqNum is the Sequence Counter of the ObjectServer secure wrapper frame

   j=[0..N] is the block number

2. Encrypt all Ctr_j blocks with AES and the BAOS Client Key

3. XOR first four bytes of the Ctr_0 with the MAC to decrypt it

4. XOR the rest of the Ctr_n with the encrypted data to get the plain ObjectServer service

5. Generate first block B_0 as follow:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
SeqNum						**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**8**	**L**

   L is the length of the encrypted data of the ObjectServer secure wrapper frame

6. Generate blocks $B_1..B_n$ filled with the decrypted plain ObjectServer service, padded with zeros to the next block end

7. Encrypt all B_n blocks with AES and the BAOS Client Key

8. Compare in step 3 decrypted MAC with calculated MAC (four first bytes of the Y_n block)

Stepwise example of the message decryption is shown in Appendix F.

4.2.2.3 Encryption of the ObjectServer secure Sync request frame

Following steps are required to encrypt the ObjectServer secure Sync request frame:

1. Generate the Challenge value as random value of six bytes
2. Generate first block B_0 as follow:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
SeqNum						**Challenge**						**0**	**0**	**12**	**6**

3. Encrypt B_0 block with AES and the BAOS Client Key
4. Save MAC = four first bytes of the Y_0 block
5. Generate Ctr_0 block as follow:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
SeqNum						**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**13**	**0**

6. Encrypt Ctr_0 block with AES and the BAOS Client Key
7. XOR first four bytes of the Ctr_0 with stored MAC to get the encrypted MAC
8. XOR the next six byte of the Ctr_0 with the Challenge to get the encrypted Challenge
9. Compose the ObjectServer secure Sync request frame as defined in chapter “4.2.1.3 ObjectServer secure Sync request frame”

4.2.2.4 Decryption of the ObjectServer secure Sync response frame

Following steps are required to decrypt the encrypted ObjectServer secure Sync response frame::

1. Calculate the server's random value = request challenge XOR response challenge
2. Generate Ctr_0 block as follow:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
Server's random value						**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**13**	**0**

3. Encrypt Ctr_0 block with AES and the BAOS Client Key
4. XOR first four bytes of the Ctr_0 with the MAC to decrypt it
5. XOR the rest of the Ctr_0 with the encrypted data to get the plain data
6. Generate first block B_0 as follow:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15	Byte 16
Server's random value						**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**12**	**12**

7. Generate block B_1 filled with the decrypted plain data, padded with zeros to the next block end
8. Encrypt all B_n blocks with AES and the BAOS Client Key
9. Compare in step 4 decrypted MAC with calculated MAC (four first bytes of the Y_n block)

4.2.3 Defined resources

ObjectServer binary protocol with security extension defines three new resources:

- BAOS Client Key
- BAOS Receive Counter
- BAOS Send Counter

4.2.3.1 BAOS Client Key

This resource stores the AES-128 key, used for en-/decryption of the ObjectServer secure wrapper messages. If the BAOS Client Key is set to the special value (=FF FF FF … FF FF), then the en-/decryption of the ObjectServer services isn't possible anymore and as a result only the plain services from client will be accepted.

The BAOS Client Key is implemented as the Server Item (Appendix A).

- Item ID: ID_CLIENT_KEY (=54)
- Value length: 16 bytes
- Access: Write only
- Data:

Byte 1															Byte 16

4.2.3.2 BAOS Receive Counter

This resource stores the BAOS receive counter, the last sequence counter which is received from client. Only the ObjectServer secure wrapper frames with the higher sequence counter will be accepted by server from client to prevent the replay attack. If the BAOS Receive Counter is set to the special value (=FF FF FF FF FF FF), then the check of the sequence counter of the received services will be deactivated and as a result all ObjectServer secure wrapper frames from client will be accepted.

The BAOS Receive Counter is implemented as the Server Item (Appendix A).

- Item ID: ID_COUNTER_RCV (=55)
- Value length: 6 bytes
- Access: Read/Write
- Data:

MSB					LSB

4.2.3.3 BAOS Send Counter

This resource stores the BAOS send counter, the last sequence counter which is sent to client. It will be consequently incremented in every ObjectServer secure wrapper frame which is sent by server to client.

The BAOS Send Counter is implemented as the Server Item (Appendix A).

- Item ID: ID_COUNTER_SND (=56)
- Value length: 6 bytes
- Access: Read/Write
- Data:

MSB					LSB

4.2.4 Factory Reset

To initiate a reset of the ObjectServer to the factory state the following request should be sent from the client:

Byte 1	Byte 2	Byte 3	Byte 4
0xF1	**0x01**	**0x02**	**0x00**

This will also reset the standard KNX resources to the factory state.

Using this service is the only way to revive the ObjectServer if the BAOS Client Key is lost by the client.

5 BAOS Protocol via USB

The USB implementation of the BAOS protocol is in-line with the USB specification of KNX. Therefore, HID reports are used as transfer channel. Each report has a size of 64 bytes and starts with the report ID = 1. Longer BAOS messages are split into several reports.

For details of the USB usage in KNX please refer to the KNX specification. To integrate the USB BAOS solution in your application please contact Weinzierl concerning the BAOS SDK.

The following is an example of how the GetServerItem.req service is encapsulated in the KNX HID Report:

KNX HID Report Body													
KNX USB Transfer Protocol Header								KNX USB Transfer Protocol Body					
Protocol version	Header length	Body length		Protocol ID	EMI ID	Manufacturer code							
00	08	00	06	01	03	00	00	F0	01	00	01	00	01

Figure 8: KNX HID Report with GetServerItem.Req service

5.1 Host protocol security USB interface

The security protocol for the BAOS interface via USB is identical to the security algorithms via serial interface. Pleas refer to 4.2 Host protocol security for serial or USB interface.

6 BAOS Protocol via IP

Clients communicating over the KNXnet/IP protocol to the ObjectServer should use the “Core” services of the KNXnet/IP protocol to discover the servers, to get a list of supported services and to manage the connection. If the ObjectServer protocol is supported by the KNXnet/IP server, a service family with the ID = 0xF0 is present in the device information block (DIB) “supported service families”. The same ID = 0xF0 should be used by the client to set the “connection type” field in the connection request.

The ObjectServer communication procedure is like the tunneling connection of KNXnet/IP protocols (see chapter 3.8.4 of the KNX specification for details). The communication partners send the requests (ServiceType = 0xF080) to each other, which will be acknowledged (ServiceType = 0xF081) by the opposite side. Each request includes the ObjectServer message (Figure 9).

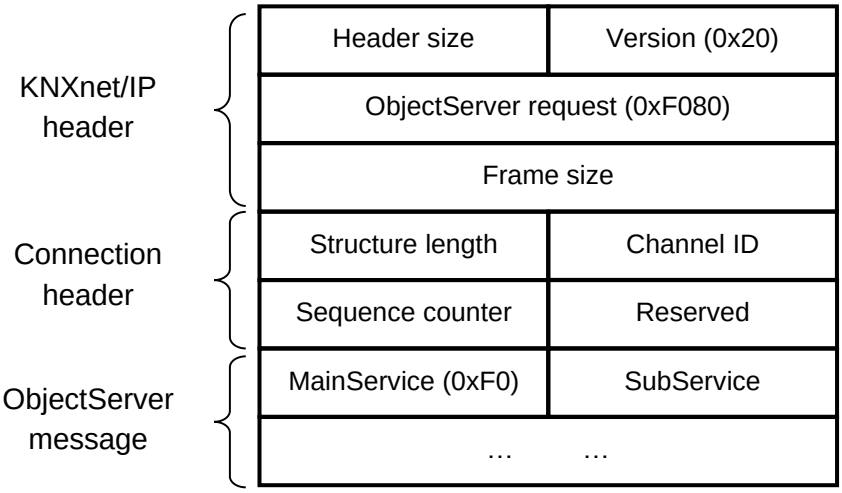

Figure 9: Integration of the ObjectServer message into a KNXnet/IP frame

6.1 TCP/IP

TCP/IP provides the whole required functionality from connection management and maintenance to data integrity. The encapsulation of the ObjectServer protocol into TCP/IP is simple. Only a header shall be added (see Figure 10) to the ObjectServer protocol. This header consists of a KNXnet/IP header including the frame length and a connection header.

The frame length is calculated like this:

Header size (6 bytes) + structure length (4 bytes) + length of object server message

Before the client is able to send the requests to the ObjectServer it must establish a TCP/IP connection to the IP address and the TCP port of an ObjectServer.

The default port for the ObjectServer is 12004 (decimal).

To prevent a timeout of the TCP/IP connection, at least every 60 seconds a communication shall be performed (e.g. requesting a server item).

Only a single object server request shall be transmitted via TCP.

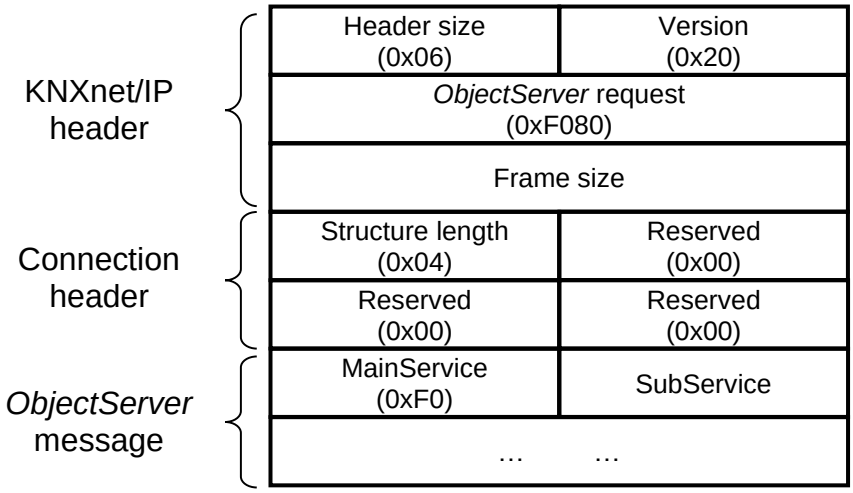

Figure 10: Integration of the ObjectServer message into TCP/IP

6.1.1 Example (GetServerItem)

This example shows how to get the first server item (hardware type) of the device using the TCP/IP encapsulation:

6.1.1.1 Request

Header										Object Server Message					
KNXnet/IP Header						Connection Hdr									
06	20	F0	80	00	10	04	00	00	00	F0	01	00	01	00	01

Figure 11: Example (GetServerItem.Req)

6.1.1.2 Response

Header										Object Server Message														
KNXnet/IP Header						Connection Hdr																		
06	20	F0	80	00	19	04	00	00	00	F0	81	00	01	00	01	00	01	06	00	00	C5	07	00	02

Figure 12: Example (GetServerItem.Res)

6.2 IP Discovery

This chapter describes the possibilities to find the installed ObjectServers in the local network. This allows the clients to find and to select automatically a certain ObjectServer for the communication, alternatively manually selected by the user. Currently only one discovery procedure is supported, which is based on the KNXnet/IP discovery algorithm. This chapter describes it briefly. For the full description of the KNXnet/IP discovery algorithm please refer to the KNX handbook Volume 3.8.

For the IP discovery procedure is shown in Figure 13. The client, which is looking for the installed ObjectServers, sends a search request via multicast to the predefined address 224.0.23.12, port 3671 (decimal). The ObjectServers send back a search response with the device information block (DIB), which contains the information about the support of the ObjectServer protocol and other things.

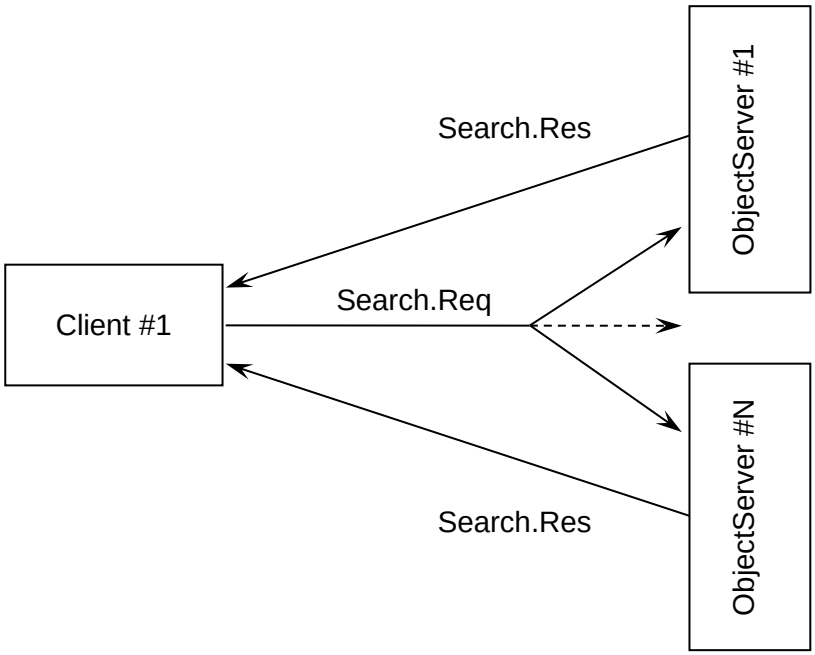

Figure 13: IP discovery

6.2.1 Search.Req

The search request has a length of 14 bytes and its format is shown in Figure 14. Most fields are fixed, the client should fill only the fields “IP address” and “IP port”. These fields are used by the ObjectServer as destination IP address and port for the search response. Fields longer than one byte are big-endian formatted.

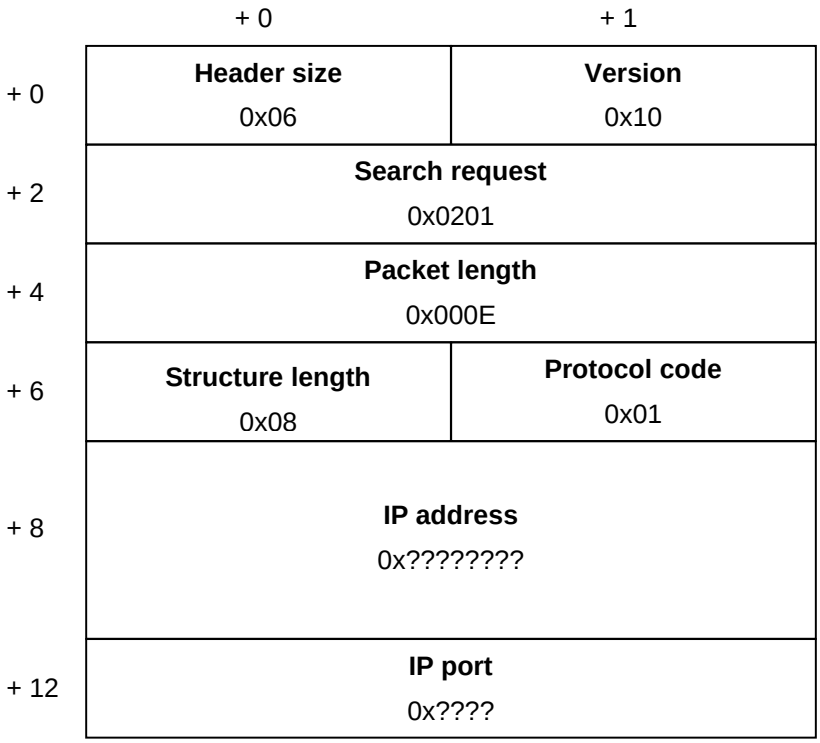

Figure 14: Structure of the Search.Req packet

6.2.2 Search.Res

The search response from the ObjectServer has, in version 1.0 of the protocol, the length of 84 bytes and its format is shown in Figure 15. The support of the ObjectServer protocol by the device is indicated through the existence of the manufacturer DIB at offset +76 bytes in the packet. This manufacturer DIB has the length of 8 bytes.

	+ 0	+ 1
+ 0	**Header size** 0x06	**Version** 0x10
+ 2	**Search response** 0x0202	
+ 4	**Packet length** 0x0054	
+ 6	**HPAI length** 0x08	
	Host Protocol Address Information (HPAI)	
+ 14	**DEV DIB length** 0x36	
	Device information block (DEV DIB)	
+ 68	**SVC DIB length** 0x08	
	Supported services DIB (SVC DIB)	
+ 76	**Manufacturer DIB len** 0x08	**Manufacturer DIB type** 0xFE
+ 78	**Manufacturer ID** 0x00C5	
+ 80	**Record type** 0x01	**Record length** 0x04
+ 82	**ObjectServer protocol** 0xF0	**ObjectServer version** 0x20

Figure 15: Structure of the Search.Res packet

6.3 IP Security Extension

The ObjectServer protocol with IP Security Extension is based on KNXnet/IP Security. A major benefit of this solution is that it does not require certificates.

The ObjectServer services are encapsulated into KNXnet/IP Secure Wrapper frames:

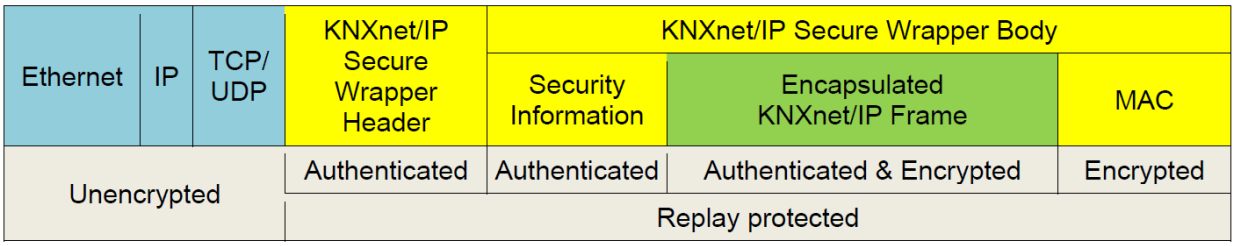

Figure 16: KNXnet/IP Secure Wrapper frame

For the details and background can be referenced to (3) KNX specification, Chapter 3/8/9 “KNX IP Secure”.

The main difference to KNXnet/IP Security is that the ObjectServer protocol is using another port than 3671. This port can be configured using the ETS, the default port number is 12004.

6.3.1 Secure session

Firstly, a secure session has to be set up. This is based on an Elliptic-Curve Diffie-Hellman (ECDH) key agreement algorithm using the Curve25519.

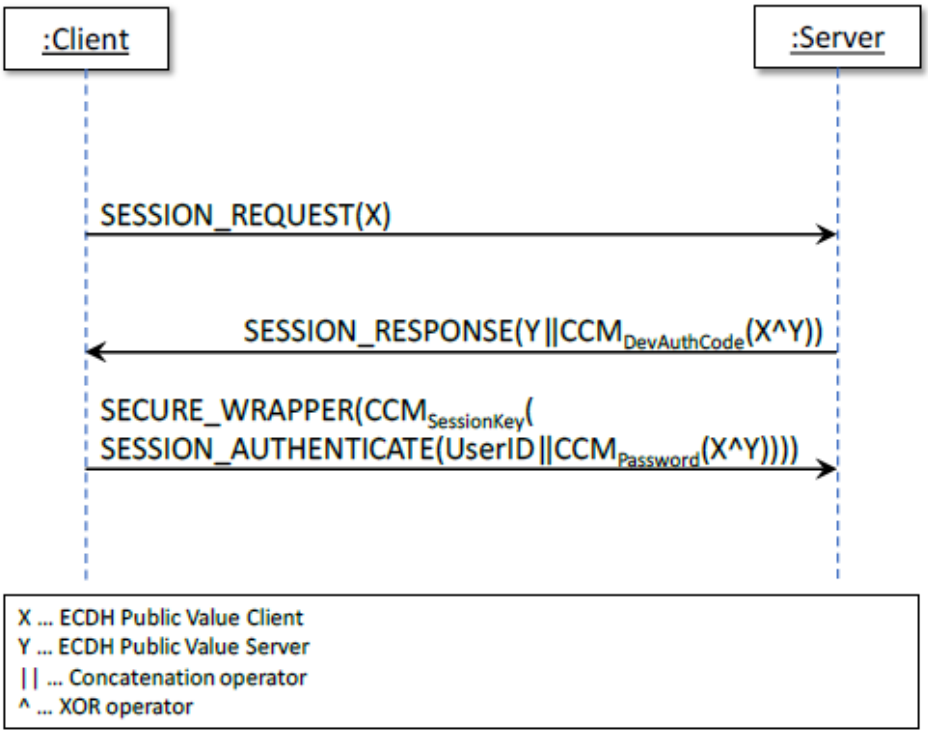

Figure 30 - Secure session setup handshake

Figure 17: Secure session setup handshake

A detailed explanation can be found in the chapter 2.2.3 Unicast connections of (3) KNX specification, Chapter 3/8/9 “KNX IP Secure”.

The secure ObjectServer services require the User ID = 1.

The password hash is derived from the user chosen password text in the ETS parameter dialogue (General Settings). It has to be calculated by the client when authenticating a secure ObjectServer session:

PasswordHash =

PBKDF2(HMAC-SHA256,<PASSWORD>,"user-password.1.secure.ip.knx.org", 500, 128)

6.3.2 Connection

The ObjectServer connection has be established using the service Connect.req / Connect.Resp services. This can be done inside a secure session.

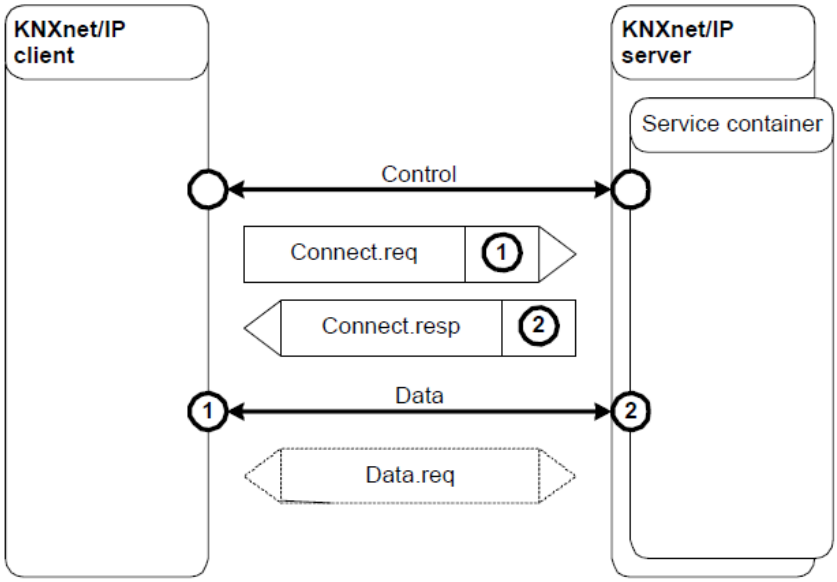

Figure 5 – Establishing a data connection

Figure 18: Establishing a ObjectServer connection

This is the required connect request information (CRI):

Index	Length	Name	
0	1 byte	Structure Length	=
1	1 byte	ConnectionType Code	= 0xFE = Manufacture Specific Protocol
2+3	2 bytes	Manufacturer Id	= 0x00 0xC5 = Weinzierl
4	1 byte	Manuf. spec. Protocol	= 0xF0 = BAOS
5	1 byte	reserved	

Figure 19: ObjectServer CRI

6.3.3 Communication

The ObjectServer services are similar to KNXnet/IP Tunneling.

This is the required ObjectServer request:

Index	Length	Name	
0	1 byte	HeaderLength	KNXnet/IP Header = 6
1	1 byte	ProtocolVersion	= 0x20
2	2 bytes	ServiceTypeId	= BAOS_REQUEST = 0xF080
4	2 bytes	TotalLength	
6	1 byte	Structure Length	KNXnet/IP connection header: = 4
7	1 byte	Channel ID	
8	1 bytes	Sequence Counter	= reserved = 0
9	1 bytes	reserved	= 0
10	var.	ObjectServer service	Data = BAOS frame

Figure 20: ObjectServer request

6.4 Device discovery

Available KNX IP BAOS devices can be discovered using the KNXnet/IP search services:

SearchReq

Sent from the client via multicast (KNX System multicast IP address: 223.0.23.12, port 3671)

SearchResp

Direct response to the client.

SearchReqExt

SearchRespExt

The search responses include a manufacturer specific device Information Block (DIB). Record type 1 is always present, record type 2 only if security is enabled.

This is the DIB including record type 1 & 2:

	1 byte	Manufacturer DIB len	= 12
	1 byte	Manufacturer DIB type	= 0xFE
	2 bytes	Manufacturer Id	= 0x00C5 = Weinzierl
	1 byte	Record type	= 1 (Weinzierl specific protocols)
	1 byte	Record length	= 4
	1 byte	Weinzierl protocol	= 0xF0 (ObjectServer protocol)
	1 byte	Protocol version	= 0x22 (ObjectServer version)
	1 byte	Record type	= 2 (Weinzierl security protocols)
	1 byte	Record length	= 4
	1 byte	Weinzierl protocol	= 0xF0 (ObjectServer protocol)
	1 byte	Required Security Version	= 0x01 (KNXnet/IP security version)

Encoded similar to the Supported service families DIB as in "3/8/2 Core".
Encoded similar to the Secured service families DIB as in "3/8/9 Security 2.6.2.2".

Figure 21: ObjectServer DIB

6.5 Examples

6.5.1 Unsecure ObjectServer communication

Open a ObjectServer connection

Connect.req

06 20 02 05 00 1C 08 02 00 00 00 00 00 00 08 02 00 00 00 00 00 00 06 FE 00 C5 F0 00

Connect.res

06 20 02 06 00 12 01 00 08 02 00 00 00 00 00 00 02 F0

Get ObjectServer server item

GetServerItem.req

F0 01 00 01 00 01

Encapsulated in BAOS.req

06 20 F0 80 00 10 04 01 00 00 F0 01 00 01 00 01

GetServerItem.res

F0 81 00 01 00 01 00 01 06 00 00 C5 07 00 14

Encapsulated in BAOS.res

06 20 F0 80 00 19 04 01 00 00 F0 81 00 01 00 01 00 01 06 00 00 C5 07 00 14

Close a ObjectServer connection

Disconnect.req

06 20 02 09 00 10 01 00 08 02 00 00 00 00 00 00

Disconnect.res

06 20 02 0A 00 08 01 00

6.5.2 Secure ObjectServer communication

In order to be able to fully interpret a secure telegram log, all keys (including the private ones) would have to be known.

This example shows the sequence of steps required. Detailed examples can be found in annex A of (3) KNX specification, Chapter 3/8/9 “KNX IP Secure”.

Create a secure session using Curve25519

Session.req

Session.res

Session authentification

SessionAuth

SecureWrapper

SessionStatus

SecureWrapper

Open a ObjectServer connection

Connect.req

SecureWrapper

Connect.res

SecureWrapper

Get ObjectServer server item

GetServerItem.req

Encapsulated in BAOS.req

SecureWrapper

BAOS GetServerItem.res

Encapsulated in BAOS.res

SecureWrapper

Close a BAOS connection

Disconnect.req

SecureWrapper

Disconnect.res

SecureWrapper

Close session

SessionStatus

SecureWrapper

SessionStatus

SecureWrapper

Appendix A. Server Item IDs

The following items are present in all device types and protocol versions 2.1:

ID	Item	Size in bytes	Ac-cess	Indi-cation
1	**Hardware type** Can be used to identify the hardware type. Coding is manufacturer specific. It is mapped to property PID_HARDWARE_TYPE in device object.	6	R	N
2	**Hardware version** Version of the ObjectServer hardware Coding e.g.: 0x10 = Version 1.0	1	R	N
3	**Firmware version** Version of the ObjectServer firmware Coding e.g.: 0x10 = Version 1.0	1	R	N
4	**KNX manufacturer code DEV** KNX manufacturer code of the device, not modified by ETS. It is mapped to property PID_MANUFACTURER_ID in device object.	2	R	N
5	**KNX manufacturer code APP** KNX manufacturer code loaded by ETS. It is mapped to bytes 0 and 1 of property PID_APPLICATION_VER in application object.	2	R	N
6	**Application ID (ETS)** ID of application loaded by ETS. It is mapped to bytes 2 and 3 of property PID_APPLICATION_VER in application object.	2	R	N
7	**Application version (ETS)** Version of application loaded by ETS. It is mapped to byte 4 of property PID_APPLICATION_VER in application object.	1	R	N
8	**Serial number** Serial number of device. It is mapped to property PID_SERIAL_NUMBER in device object.	6	R	N
9	**Time since reset [ms]**	4	R	N
10	**Bus connection state** Values: “0” – disconnected “1” – connected	1	R	Y
11	**Maximum buffer size**	2	R	N
12	**Length of description string**	2	R	N
13	**Baudrate** (only if serial port is present) Values: “0” – unknown	1	RW	N

	“1” – 19200 “2” – 115200			
14	**Current buffer size**	2	RW	N
15	**Programming mode** Values (bit 0): “0” – not active “1” – active	1	RW	Y
16	**Protocol Version (Binary)** Version of the ObjectServer binary protocol Coding e.g.: 0x20 = Version 2.0	1	R	N
17	**Indication Sending** Values (bit 0): “0” – not active “1” – active	1	RW	N

The following items are optional and are fully or partly implemented in some device types:

ID	Item	Size in bytes	Ac-cess	Indi-cation
18	**Protocol Version (WebService)** Version of the ObjectServer protocol via web services Coding e.g.: 0x20 = Version 2.0	1	R	N
19	**Protocol Version (RestService)** Version of the ObjectServer protocol via rest services Coding e.g.: 0x21 = Version 2.1	1	R	N
20	**Individual Address** The individual KNX address of the device	2	RW	N
21	**Mac Address**	6	R	N
22	**Tunnelling Enabled** KNXnet/IP tunneling Values: “0” – disabled “1” – enabled	1	RW	Y
23	**Baos Binary Enabled** Access via BAOS Binary connection Values: “0” – disabled “1” – enabled	1	RW	Y
24	**Baos Web Enabled** Web Services Values: “0” – disabled “1” – enabled	1	RW	Y
25	**Baos Rest Enabled** REST services Values: “0” – disabled “1” – enabled	1	RW	Y
26	**Http File Enabled** Webserver disabled or disabled	1	RW	Y

	Values: “0” – disabled “1” – enabled			
27	**Search Request Enabled** Device responds to search requests (yes/no) Values: “0” – no “1” – yes	1	RW	Y
28	**Is Structured** Indicates if the current loaded database is structured Values: “0” – False “1” – True	1	R	N
29	**Max Management Clients** Max amount of available management connections	1	R	N
30	**Connected Management Clients**	1	R	N
31	**Max Tunneling Clients**	1	R	N
32	**Connected Tunneling Clients**	1	R	N
33	**Max Baos UDP Clients**	1	R	N
34	**Connected Baos UDP Clients**	1	R	N
35	**Max Baos TCP Clients**	1	R	N
36	**Connected Baos TCP Clients**	1	R	N
37	**Device Friendly Name** String of an optionally given name for this device.	30	RW	N
38	**Max Datapoints** Number of available data points	2	R	N
39	**Configured Datapoints** Current number of configured data points	2	R	N
40	**Max Parameter Bytes** Number of available parameter bytes	2	R	N
41	**Download Counter** ETS download counter	2	R	N
42	**IP Assignment** DHCP or manual	1	RW	Y
43	**IP Address**	4	RW	Y
44	**Subnet Mask**	4	RW	Y
45	**Default Gateway**	4	RW	Y
46	**Time Since Reset Unit** x=ms, s=seconds, m=minutes, h= hours	1	RW	Y
47	**System Time**	variable	RW	Y
48	**System Timezone Offset**	1	RW	Y
49	**Menu Enabled** Values can be edited on the device menu Values: “0” – disabled “1” – enabled	1	RW	Y

50	**Enable Suspend** Device can enter the suspend state if enabled. This feature is used for USB only. Values: “0” – disabled “1” – enabled (default after reset)	1	RW	N
51	**RF Domain Address** It is mapped to property PID_RF_DOMAIN_ADDRESS in device object.	6	RW	N
52	**Supported Status Flags** (related to Item #53) Values: Bit 0-15: “0” – flag not supported “1” – flag supported	2	R	N
53	**Status Flags** Values: Bit 0: application running Bit 1: application and tables loaded Bit 2: TL connection open Bit 3: secure mode activated Bit 4-15: reserved	2	R	N
54	**Client Key** BAOS security: AES-128 key for en-/decryption. (s. 4.2.3.1 BAOS Client Key for details)	16	W	N
55	**Receive Counter** BAOS security: receive counter. (s. 4.2.3.2 BAOS Receive Counter for details)	6	RW	N
56	**Send Counter** BAOS security: send counter. (s. 4.2.3.3 BAOS Send Counter for details)	6	RW	N

Note: Fields longer than one byte are big-endian formatted.

Appendix B. Error codes

In case of an error, the response informs about the failed item ID (Start) and Count is always 0.

Error code	Description
0	**No error**
1	**Internal error**
2	**No element found**: - Server item is not available, - Datapoint is not configured by ETS - Datapoint not updated from KNX-Bus
3	**Buffer is too small**: - Buffer for description string too small - Buffer for parameters too small
4	**Item is not writeable**: - Server item is not writable
5	**Service is not supported** E.g. BAOS Modules do not support Description Strings.
6	**Bad service parameter**: Server item, Datapoint or Parameter ID out of range.
7	**Bad ID**: Writing of item failed - Server item ID out of range - Datapoint out of range - Datapoint not configured by ETS
8	**Bad command/value**: - Writing Server item failed (internal error) - Writing Datapoint failed (illegal command)
9	**Bad length**: Wrong length writting Server item or Datapoint or Parameter bytes.
10	**Message inconsistent**: Count in writting request is wrong.
11	**Object server is busy**

Appendix C. Datapoint value types

Type code	Value size
0	1 bit
1	2 bits
2	3 bits
3	4 bits
4	5 bits
5	6 bits
6	7 bits
7	1 byte
8	2 bytes
9	3 bytes
10	4 bytes
11	6 bytes
12	8 bytes
13	10 bytes
14	14 bytes

Appendix D. Datapoint types (DPT)

Type code	Value size
0	Datapoint disabled
1	DPT 1 (1 Bit, Boolean)
2	DPT 2 (2 Bit, Control)
3	DPT 3 (4 Bit, Dimming, Blinds)
4	DPT 4 (8 Bit, Character Set)
5	DPT 5 (8 Bit, Unsigned Value)
6	DPT 6 (8 Bit, Signed Value)
7	DPT 7 (2 Byte, Unsigned Value)
8	DPT 8 (2 Byte, Signed Value)
9	DPT 9 (2 Byte, Float Value)
10	DPT 10 (3 Byte, Time)
11	DPT 11 (3 Byte, Date)
12	DPT 12 (4 Byte, Unsigned Value)
13	DPT 13 (4 Byte, Signed Value)
14	DPT 14 (4 Byte, Float Value)
15	DPT 15 (4 Byte, Access)
16	DPT 16 (14 Byte, String)
17	DPT 17 (1 Byte, Scene Number)
18	DPT 18 (1 Byte, Scene Control)
19	DPT 19 (8 Byte, Date Time)
20…31	Reserved
32	DPT 20 (1 Byte, HVAC Mode)
33	DPT 232 (3 Byte, Color RGB)
34	DPT 251 (6 Byte, Color RGBW)
35…254	Reserved
255	Unknown DPT

Appendix E. Encryption example

Plain ObjectServer service: *{0xF0, 0x01, 0x00, 0x01, 0x00, 0x01}*

BAOS Client Key: *[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]*

BAOS Send Counter: *0x010203040506*

Block B_0: *[0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x08, 0x06]*

Block B_1: *[0xF0, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]*

Block Y_0: *[0xC5, 0x6D, 0x1E, 0xF4, 0xBF, 0x06, 0xA8, 0xD4, 0x61, 0x9D, 0xB0, 0xBF, 0xDE, 0x30, 0x3B, 0x48]*

Block Y_1: *[0x51, 0x76, 0x04, 0xEA, 0x0B, 0x7A, 0x9F, 0x5B, 0x3F, 0x38, 0x6F, 0x14, 0xE5, 0xCA, 0xD9, 0xA5]*

Plain MAC: *[0x51, 0x76, 0x04, 0xEA]*

Block Ctr_0: *[0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09, 0x00]*

Encrypted Ctr_0: *[0xDA, 0x76, 0xC7, 0x9E, 0xFA, 0x39, 0x48, 0x6A, 0xBF, 0x7A, 0xBB, 0xF3, 0xF2, 0xC1, 0xDC, 0x4F]*

Encrypted MAC: *[0x8B, 0x00, 0xC3, 0x74]*

Encrypted Data: *[0x0A, 0x38, 0x48, 0x6B, 0xBF, 0x7B]*

Result ObjectServer secure wrapper frame:

{0xC0, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x0A, 0x38, 0x48, 0x6B, 0xBF, 0x7B, 0x8B, 0x00, 0xC3, 0x74}

Appendix F. Decryption example

ObjectServer secure wrapper frame:

{0xC0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0xFA, 0xF1, 0xD3, 0x3B, 0x60, 0x7A, 0xEE, 0xA4, 0x07, 0x29, 0x7B, 0xAF, 0x9A, 0x93, 0xF6, 0xB1, 0x0C, 0xB4, 0xB5}

BAOS Client Key: *[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]*

Block Ctr_0: *[0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09, 0x00]*

Block Ctr_1: *[0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09, 0x01]*

Encrypted Ctr_n:

[0x2A, 0x0E, 0xA5, 0x4E, 0x0A, 0x70, 0xD3, 0x3A, 0x60, 0x7B, 0xEE, 0xA5, 0x01, 0x29, 0x7B, 0x6A]

[0x99, 0x93, 0xFF, 0x56, 0xCD, 0x49, 0xD5, 0x8F, 0xFB, 0x35, 0xE2, 0x87, 0xBC, 0x5A, 0x98, 0x27]

Decrypted MAC: *[0x9B, 0x02, 0x11, 0xFB]*

Plain ObjectServer service:

{0xF0, 0x81, 0x00, 0x01, 0x00, 0x01, 0x00, 0x01, 0x06, 0x00, 0x00, 0xC5, 0x03, 0x00, 0x09}

Block B_0: *[0x00, 0x00, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x08, 0x0F]*

Block B_1: *[0xF0, 0x81, 0x00, 0x01, 0x00, 0x01, 0x00, 0x01, 0x06, 0x00, 0x00, 0xC5, 0x03, 0x00, 0x09, 0x00]*

Block Y_0: *[0x02, 0xD6, 0xD5, 0x22, 0xA6, 0x2B, 0xF9, 0xBD, 0xCE, 0xB5, 0x7A, 0xC7, 0xCA, 0x16, 0xF5, 0x83]*

Block Y_1*: [0x9B, 0x02, 0x11, 0xFB, 0x1F, 0x4D, 0xBD, 0x70, 0x80, 0xAD, 0xBD, 0x84, 0x6D, 0x4A, 0x67, 0x32]*

Calculated MAC: *[0x9B, 0x02, 0x11, 0xFB]*

WEINZIERL ENGINEERING GmbH
Achatz 3-4
84508 Burgkirchen an der Alz
GERMANY

Tel.: +49 8677 / 916 36 – 0
E-Mail: info@weinzierl.de
Web: www.weinzierl.de

2024-07-22