WEINZIERL

KNX BAOS Binary Protocol

BAOS Binary Services Version 2.0 / 2.1 for

KNX BAOS Module 830
KNX BAOS Module 832
KNX BAOS Module 838 *kBerry*
KNX BAOS Module RF 840

KNX USB Interface 312
KNX USB Module 322
KNX USB Interface 332 *Stick*

KNX IP BAOS 771
KNX IP BAOS 772
KNX IP BAOS 773
KNX IP BAOS 774
KNX IP BAOS 777

WEINZIERL ENGINEERING GmbH
DE-84508 Burgkirchen
E-Mail: info@weinzierl.de
Web: www.weinzierl.de

Document history

Document status	Date	Editor
Release	2011-01-14	Hl
Added: - Server Item 18 - Error message: Busy - TCP encapsulation: Extension of description and example	2011-06-22	Hl
Changed: Format	2011-10-17	St
Changed: Format	2012-10-05	Wz
Added KNX IP BAOS 777	2015-07-30	Wz
Added additional BAOS Services version 2.1	2015-09-02	Ms
Added Timers section	2015-09-03	Ms
Added version 2.1 Item ID’s	2015-10-14	Ms
Updated Access and Indication columns of Server Items	2015-12-04	Ms
Added Timers description	2015-12-07	Ms
Added BAOS Modules	2015-12-15	Gi
Spelling	2016-01-16	Gi
Protocol corrections	2016-02-29	Ky
Minor improvements	2016-03-03	Gi
Correct offset of SetServerItem.Req	2016-08-31	Bu
Extended list of supported server items of BAOS Modules 830/840	2016-08-31	Bu
Correct length of FT1.2 frame	2016-09-08	Bu
Fixed FT1.2 communication example	2017-02-27	Gi
Added 773, 774 at Title	2017-06-14	Gi
ServerItems updated	2017-06-29	Gi
Added USB devices	2017-07-13	Wz

Contents

1. What is an ObjectServer? 5

1. Getting started 6

2. Communication protocol 7

2.1. GetServerItem.Req 8

2.2. GetServerItem.Res 9

2.3. SetServerItem.Req 10

2.4. SetServerItem.Res 11

2.5. ServerItem.Ind 12

2.6. GetDatapointDescription.Req 13

2.7. GetDatapointDescription.Res 14

2.8. GetDescriptionString.Req 17

2.9. GetDescriptionString.Res 18

2.10. GetDatapointValue.Req 20

2.11. GetDatapointValue.Res 21

2.12. DatapointValue.Ind 24

2.13. SetDatapointValue.Req 25

2.14. SetDatapointValue.Res 27

2.15. GetParameterByte.Req 28

2.16. GetParameterByte.Res 29

2.17 SetDatapointHistoryCommand.Req 30

2.18 SetDatapointHistoryCommand.Res 31

2.19 GetDatapointHistoryState.Req 32

2.20 GetDatapointHistoryState.Res 33

2.21 GetDatapointHistory.Req 35

2.22 GetDatapointHistory.Res 36

3. Timers _______ 38

3.1 Overview _______ 38

3.2 Code _______ 38

3.3 Trigger Type Date _______ 39

3.4 Trigger Type Interval _______ 39

3.5 Job Type SetDatapointValue _______ 40

3.6 Get Timer _______ 40

3.7 Set Timer _______ 43

4. Encapsulating of the ObjectServer protocol _______ 47

4.1. Serial FT1.2 _______ 48

4.1. USB HID _______ 49

4.2. KNXnet/IP _______ 50

4.3. TCP/IP _______ 51

5. Discovery procedure _______ 53

5.1. KNXnet/IP discovery algorithm _______ 54

Appendix A. Item IDs _______ 57

Appendix B. Error codes _______ 61

Appendix C. Datapoint value types _______ 62

Appendix D. Datapoint types (DPT) _______ 63

Appendix E. FT1.2 protocol _______ 64

D.1. Communication procedure _______ 64

D.2. Frame format _______ 65

D.3. Communication example _______ 66

1. What is an *ObjectServer*?

The ObjectServer is a hardware component, which is connected to the KNX bus and represents it for the client as 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 normally encapsulated into some other communication protocol (e.g. FT1.2, IP, etc.).

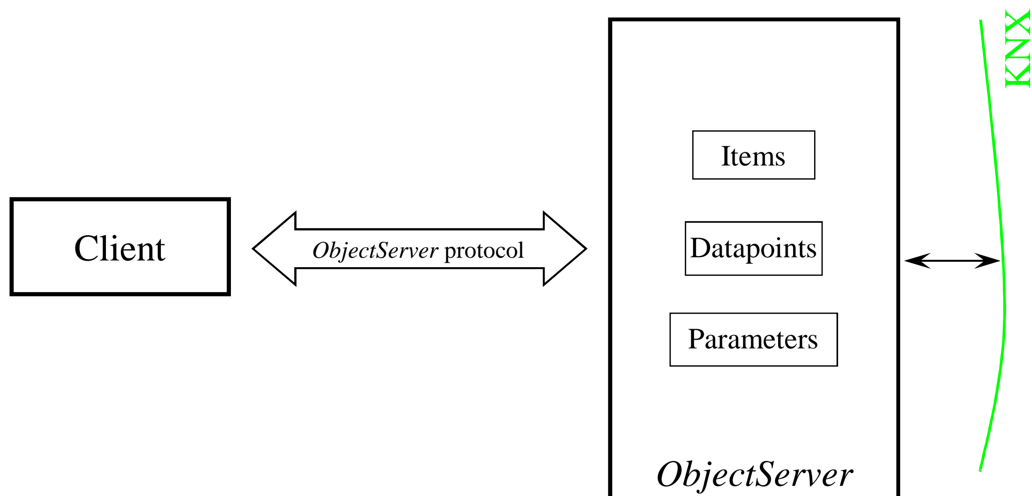

Figure 1: Communication between *ObjectServer* and Client

1. Getting started

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

Net'n Node telegram view with BAOS data points

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

2. Communication protocol

How is mentioned above, the communication between the server and the client is based on an ObjectServer protocol and consists of the requests sent by client and the server responses. To inform the client about the changes of datapoint's value an indication is defined, which will be sent asynchronously from the server to the client. In this version of the protocol are defined following services:

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

2.1. 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		Maximal number of items to return

As response the server sends to the client the values of supported items from the range [StartItem … StartItem+NumberOfItems-1].

The defined item IDs are specified in appendix A.

2.2. GetServerItem.Res

This response is sent by the server as reaction to the GetServerItem request. If an error is detected during the request processing server send a negative response that has following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x81	Subservice code
+2	StartItem	2		Index of bad item
+4	NumberOfItems	2	0x00	
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

If request can be successfully processed by the server it sends a positive response to the client that has 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

2.3. SetServerItem.Req

This request is sent by the client to set the new value of the server item.

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 defined item IDs are specified in appendix A.

2.4. SetServerItem.Res

This response is sent by the server as reaction to the SetServerItem request. If an error is detected during the request processing server send a negative response that has following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x82	Subservice code
+2	StartItem	2		Index of bad item
+4	NumberOfItems	2	0x00	
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

If request can be successfully processed by the server it sends a positive response to the client that has 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	0x00	
+6	ErrorCode	1	0x00	

2.5. ServerItem.Ind

This indication is sent asynchronously by the server if the datapoint(s) value is changed and has 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 data length	1		Data length of last item
+9	First item data	1-255		Data of last 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

For the coding of the item data see the description of the GetServerItem response.

2.6. GetDatapointDescription.Req

This request is sent by the client to get the description(s) of the datapoint(s). 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		Maximal number of descriptions to return

As response the server sends to the client the descriptions of datapoints from the range [StartDatapoint … StartDatapoint + NumberOfDatapoints - 1].

2.7. GetDatapointDescription.Res

This response is sent by the server as reaction to the GetDatapointDescription request. If an error is detected during the request processing, the server sends a negative response with 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	0x00	
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

If request can be successfully processed by the server it sends a positive response to the client with 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

The defined types of the datapoint value are specified in appendix C.

The coding of the datapoint configuration flags is following:

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

The defined datapoint types can be found in appendix D.

2.8. GetDescriptionString.Req

This request is sent by the client to get the human-readable description string(s) of the datapoint(s). 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 string
+4	NumberOfStrings	2		Maximal number of strings to return

As response server sends to the client the description strings of datapoints from the range [StartString … StartString+NumberOfStrings-1].

Note: This service is optional and could be not implemented in some servers.

2.9. GetDescriptionString.Res

This response is sent by the server as reaction to the GetDescriptionString request. If an error is detected during the processing of the request, the server sends a negative response with 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	0x00	
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

If request can be successfully processed by the server it sends a positive response to the client with 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 do not include a termination null. The length of each datapoint description string is given with the corresponding StrLen.

2.10. GetDatapointValue.Req

This request is sent by the client to get the value(s) of the datapoint(s). 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		Maximal 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

As response the server sends to the client the values of datapoints from the range [StartDatapoint … StartDatapoint+NumberOfDatapoints-1].

2.11. GetDatapointValue.Res

This response is sent by the server as reaction to the GetDatapointValue request. If an error is detected during the processing of the request, the server sends a negative response with 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	0x00	
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

If request can be successfully processed by the server, it sends a positive response to the client with 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

The state byte is coded as follows:

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

The KNX datapoints with the length less than one byte are coded into the one byte value as follows:

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

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

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

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

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

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

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

2.12. DatapointValue.Ind

This indication is sent asynchronously by the server if the datapoint(s) value(s) is/has changed and 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

For the coding of the state byte see the description of the GetDatapointValue request.

For the coding of the datapoint value see the description of the GetDatapointValue response.

2.13. SetDatapointValue.Req

This request is sent by the client to set the new value(s) of the datapoint(s) or to request/transmit the new value on the bus. It also can be used to clear the transmission state of the datapoint.

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

The command byte is coded as follows:

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 with the value length, which is selected in 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 or similar.

2.14. SetDatapointValue.Res

This response is sent by the server as reaction to the SetDatapointValue request. If an error is detected during the processing of the request, the server sends a negative response with 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	0x00	
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

If request can be successfully processed by the server, it sends a positive response to the client with 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	0x00	
+6	ErrorCode	1	0x00	

2.15. GetParameterByte.Req

This request is sent by the client to get the parameter byte(s). A parameter is free-defined variable of the 8-bits length, which can be set and programmed by the Engineering Tool Software (ETS).

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		Maximal number of bytes to return

As response the server sends to the client the values of parameters from the range [StartByte … StartByte+NumberOfBytes-1].

2.16. GetParameterByte.Res

This response is sent by the server as reaction to the GetParameterByte request. If an error is detected during the request processing server send a negative response that has 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	0x00	
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

If request can be successfully processed by the server it sends a positive response to the client that has 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

2.17 SetDatapointHistoryCommand.Req

This request is sent by the client to either Start, Stop or Clear the history for one or more datapoints. See the History Command table below for the list of commands.

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x08	Subservice code
+2	StartDatapoint	2		ID of first datapoint
+4	NumberOfDatapoints	2		number of datapoints
+6	Command	1		The history command applied to each of the datapoints

The Command is coded as follows:

Value	Description
0x01	Clear. Clears the previous capture.
0x02	Start. Starts a new capture. Does not clear the previous history.
0x03	StartClear. Clears the previous capture and starts a new capture
0x04	Stop. Stops a capture. Does not clear the previous history.
0x05	StopClear. Stops and clears the capture.

Note: Not available for BAOS Modules 83x/840.

2.18 SetDatapointHistoryCommand.Res

This response is sent by the server as reaction to the SetDatapointHistoryCommand request. If an error is detected during the processing of the request, the server sends a negative response with the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x88	Subservice code
+2	StartDatapoint	2		ID of first datapoint
+4	NumberOfDatapoints	2		number of datapoints
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

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

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x88	Subservice code
+2	StartDatapoint	2		As in request
+4	NumberOfDatapoints	2	0x00	
+6	ErrorCode	1	0x00	

Note: Not available for BAOS Modules 83x/840.

2.19 GetDatapointHistoryState.Req

This request is sent by the client to get the capture state of one or more datapoints. For each datapoint the server will return its state (whether it is currently logging history or not) and the count of available items.

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x09	Subservice code
+2	StartDatapoint	2		ID of first datapoint
+4	NumberOfDatapoints	2		number of datapoints

As response server sends to the client the values of datapoints from range [StartDatapoint … StartDatapoint+NumberOfDatapoints-1].

Note: Not available for BAOS Modules 83x/840.

2.20 GetDatapointHistoryState.Res

This response is sent by the server as reaction to the GetDatapointHistoryState request. If an error is detected during the processing of the request, the server sends a negative response with the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x89	Subservice code
+2	StartDatapoint	2		ID of first datapoint
+4	NumberOfDatapoints	2	0x00	number of datapoints
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

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

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x89	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		The History state
+9	Count	4		The number of history items
…	…		…	…
+x	Last DP ID	2		ID of last datapoint
+x	Last DP State	1		The History state
+x	Count	4		The number of history items

The State is coded as follows:

Value	Description
0x00	Inactive
0x01	Available
0x02	Active
0x03	ActiveAvailable

Note: Not available for BAOS Modules 83x/840.

2.21 GetDatapointHistory.Req

This request is sent by the client to get the history for one or more datapoints. Note: There will be a limit to the size of the response, similar to the Version 1.0 and 2.0 protocols, which may dictate how this service is used. For example, you propably won't be able to get the history for 1000 objects for the past [time] in one request. History for a datapoint will include: the datapoint id, the timestamp and the value. Data will be sorted by timestamp descending, which means history items for multiple datapoints will be interleaved. It will be possible to limit the results by specifying a start and end timestamp. We will accept a zero value for both start and end timestamp. If the start or end timestamp has a value of zero we will ignore it in the query. For example, if you only specify the start timestamp, the end timestamp will be assumed to be now.

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x0A	Subservice code
+2	StartDatapoint	2		ID of first datapoint
+4	NumberOfDatapoints	2		number of datapoints
+6	StartTimestamp	2		Start Timestamp
+8	EndTimestamp	2		End Timestamp

As response server sends to the client the values of datapoints from range [StartDatapoint … StartDatapoint+NumberOfDatapoints-1].

Note: Not available for BAOS Modules 83x/840.

2.22 GetDatapointHistory.Res

This response is sent by the server as reaction to the GetDatapointHistory request. If an error is detected during the processing of the request, the server sends a negative response with the following format:

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x8A	Subservice code
+2	StartDatapoint	2		ID of first datapoint
+4	NumberOfDatapoints	2	0x00	number of datapoints
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

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

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x8A	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 Timestamp	4		Timestamp
+12	First DP Length	1		Datapoint Length
+13	First DP Value	variable		Datapoint Value
…	…		…	…
+x	Last DP ID	2		ID of last datapoint
+x	Last DP Timestamp	4		Timestamp
+x	Last DP Length	1		Datapoint Length
+x	Last DP Value	variable		Datapoint Value

Note: Not available for BAOS Modules 83x/840.

3. Timers

3.1 Overview

The binary services to get and set triggers are somewhat more complex to encode and decode than the standard baos services as they contain variable length information blocks depending on both the trigger and job types. Currently we support two trigger types, Date and Interval and one job type, SetDatapointValue.

A date trigger is essentially a one-shot trigger, which will fire once at the specified date. An interval trigger on the other hand will continue to fire according to the interval parameters, for example, once every hour.

To delete a timer we use the SetTrigger service with the trigger type to 0 (i.e. DeleteTimer). This effectively ends the timer block and no further timer information should be set for that timer.
Note: The SetTimer service can contain both set and delete blocks within the single service.

Note: Timers are not available for BAOS Modules 83x/840.

3.2 Code

Trigger Types

Type	**Value**
DeleteTimer	0
TriggerDate	1
TriggerInterval	2

Job Types

Type	**Value**
JobSetDatapointValue	1

3.3 Trigger Type Date

A date trigger is a one-shot event that will schedule the job to be run when the date and time is reached. The date trigger takes a single parameter, *date/time*, which indicates the date/time to run the job at.

Date - Trigger Parameter

Offset	**Field**	**Size**	**Value**	**Description**
+x	date/time	4		Seconds since epoch

3.4 Trigger Type Interval

This trigger schedules jobs to be run periodically, on selected intervals.

You can also specify the starting and ending dates for the schedule through the *Start date/time* and *End date/time* parameters, respectively.

If the start date is in the past, the trigger will not fire many times retroactively but instead calculates the next run time from the current time, based on the past start time.

Interval - Trigger Parameter

Offset	**Field**	**Size**	**Value**	**Description**
+x	Start date/time	4		Seconds since epoch
+x	End date/time	4		Seconds since epoch
+x	Weeks	2		Number of weeks
+x	Days	1		Number of days
+x	Hours	1		Number of hours

Interval - Trigger Parameter (continued)

Offset	Field	Size	Value	Description
+x	Minutes	1		Number of Minutes
+x	Seconds	1		Number of Seconds

3.5 Job Type SetDatapointValue

Job Parameter SetDatapointValue

Offset	Field	Size	Value	Description
+x	Datapoint Id	2		The datapoint id to set
+x	DP command	1		Command byte of DP
+x	DP length	1		Length byte of DP
+x	DP value	1-14		DP value

3.6 Get Timer

Get Timer Request

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x0B	Subservice code
+2	StartTimer	2		As in request
+4	NumberOfTimers	2		Number of Timers in this response

Get Timer negative Response

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x8B	Subservice code
+2	StartTimer	2		As in request
+4	NumberOfTimers	2	0x00	Number of Timers in this response
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

Get Timer positive Response

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main Service code
+1	SubService	1	0x8B	Subservice code
+2	StartTimer	2		As in request
+4	NumberOfTimers	2		Number of Timers in this response
+6	Timer Id	2		The unique timer id
+x	Trigger	1		The trigger (one of date, interval, could later be datapoint ind)
+x	Trigger Params Length	1		The length of the trigger parameters
+x	Trigger Params			The specific trigger parameters

Get Timer positive Response (continued)

Offset	**Field**	**Size**	**Value**	**Description**
+x	Job	1		The job (currently only SetDatapointValue)
+x	Job Params Length	1		The length of the job parameters
+x	Job Params			The specific job parameters
+x	Timer Description String Length	2		The timer description string length
+x	Timer Description String			The timer description string
…	…			…
+x	Timer Id	2		The unique timer id
+x	Trigger	1		The trigger (one of date, interval, could later be datapoint ind)
+x	Trigger Params Length	1		The length of the trigger parameters
+x	Trigger Params			The specific trigger parameters
+x	Job	1		The job (currently only SetDatapointValue)
+x	Job Params Length	1		The length of the job parameters
+x	Job Params			The specific job parameters
+x	Timer Description String Length	2		The timer description string length
+x	Timer Description String			The timer description string

3.7 Set Timer

Set Timer Request

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main Service code
+1	SubService	1	0x0C	Subservice code
+2	StartTimer	2		As in request
+4	NumberOfTimers	2		Number of Timers in this response
+6	Timer Id	2		The unique timer id
+x	Trigger	1		The trigger (one of date, interval, could later be datapoint ind)
+x	Trigger Params Length	1		The length of the trigger parameters
+x	Trigger Params			The specific trigger parameters
+x	Job	1		The job (currently only SetDatapointValue)
+x	Job Params Length	1		The length of the job parameters
+x	Job Params			The specific job parameters
+x	Timer Description String Length	2		The timer description string length
+x	Timer Description String			The timer description string
…	…			…
+x	Timer Id	2		The unique timer id

Set Timer Request (continued)

Offset	Field	Size	Value	Description
+x	Trigger	1		The trigger (one of date, interval, could later be datapoint ind)
+x	Trigger Params Length	1		The length of the trigger parameters
+x	Trigger Params			The specific trigger parameters
+x	Job	1		The job (currently only SetDatapointValue)
+x	Job Params Length	1		The length of the job parameters
+x	Job Params			The specific job parameters
+x	Timer Description String Length	2		The timer description string length
+x	Timer Description String			The timer description string

Set Timer negative Response

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x8C	Subservice code
+2	StartTimer	2		As in request
+4	NumberOfTimers	2	0x00	Number of Timers in this response
+6	ErrorCode	1		Error code

The defined error codes are specified in appendix B.

Set Timer positive Response

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main Service code
+1	SubService	1	0x8C	Subservice code
+2	StartTimer	2		As in request
+4	NumberOfTimers	2		Number of Timers in this response
+6	Timer Id	2		The unique timer id
+x	Trigger	1		The trigger (one of date, interval, could later be datapoint ind)
+x	Trigger Params Length	1		The length of the trigger parameters
+x	Trigger Params			The specific trigger parameters
+x	Job	1		The job (currently only SetDatapointValue)
+x	Job Params Length	1		The length of the job parameters
+x	Job Params			The specific job parameters
+x	Timer Description String Length	2		The timer description string length
+x	Timer Description String			The timer description string
…	…			…
+x	Timer Id	2		The unique timer id

Set Timer positive Response (continued)

Offset	**Field**	**Size**	**Value**	**Description**
+x	Trigger	1		The trigger (one of date, interval, could later be datapoint ind)
+x	Trigger Params Length	1		The length of the trigger parameters
+x	Trigger Params			The specific trigger parameters
+x	Job	1		The job (currently only SetDatapointValue)
+x	Job Params Length	1		The length of the job parameters
+x	Job Params			The specific job parameters
+x	Timer Description String Length	2		The timer description string length
+x	Timer Description String			The timer description string

4. Encapsulating of the *ObjectServer* protocol

The ObjectServer protocol has been defined to achieve the whole functionality also on the smallest embedded platforms and on the data channels with the limited bandwidth. As a result of this fact the protocol is kept very slim and has no connection management, like the connection establishment, user authorization, etc. Therefore it is advisable und mostly advantageous to encapsulate the ObjectServer protocol into some existing transport protocol to get a powerful solution for the easy access to the KNX datapoints and directly to the KNX bus.

Depending on the interface type the BAOS protocol is encapsulated:

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

4.1. Serial FT1.2

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

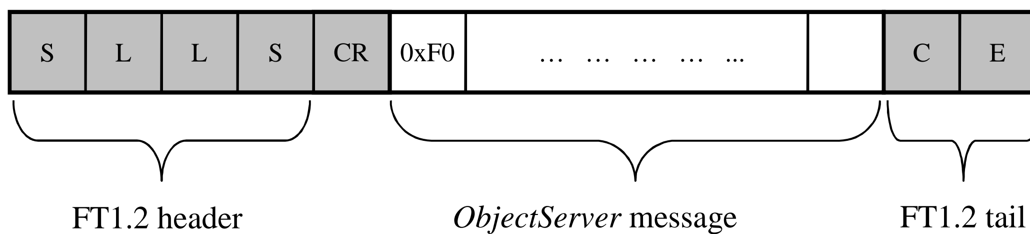

Figure 2: Integration of the *ObjectServer* message into the FT1.2 frame

The short description of the FT1.2 protocol can be found in appendix D.

4.1. USB HID

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 in 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.

4.2. KNXnet/IP

The clients that communicate over the KNXnet/IP protocol with the ObjectServer should use the “Core” services of the KNXnet/IP protocol to discovery the servers, to get the list of the 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 of the connect request.

The ObjectServer communication procedure is like for the tunneling connection of the KNXnet/IP protocols (see the chapter 3.8.4 of the KNX specification for the 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 3).

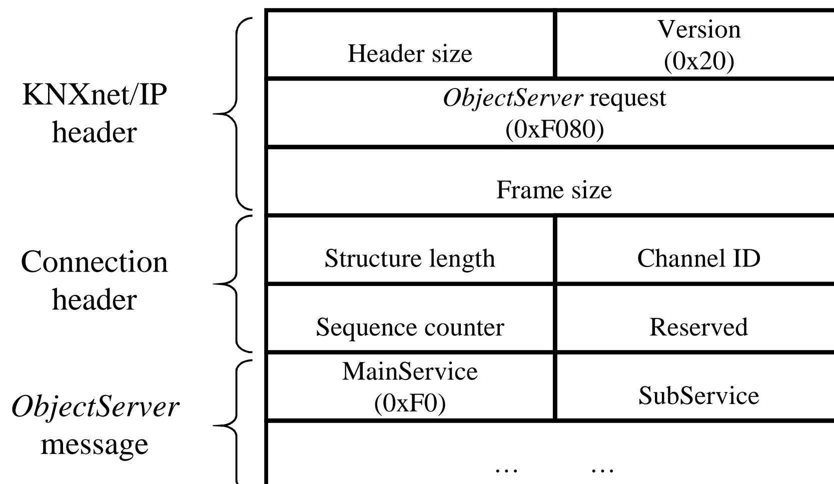

Figure 3: Integration of the ObjectServer message into the KNXnet/IP frame

4.3. TCP/IP

The TCP/IP provides the whole required functionality from connection management and maintenance to the data integrity. The encapsulating of the ObjectServer protocol into the TCP/IP is simple. Only a header shall be added (see Figure 4) 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 ObjectServer.

The default value for the ObjectServer port 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.

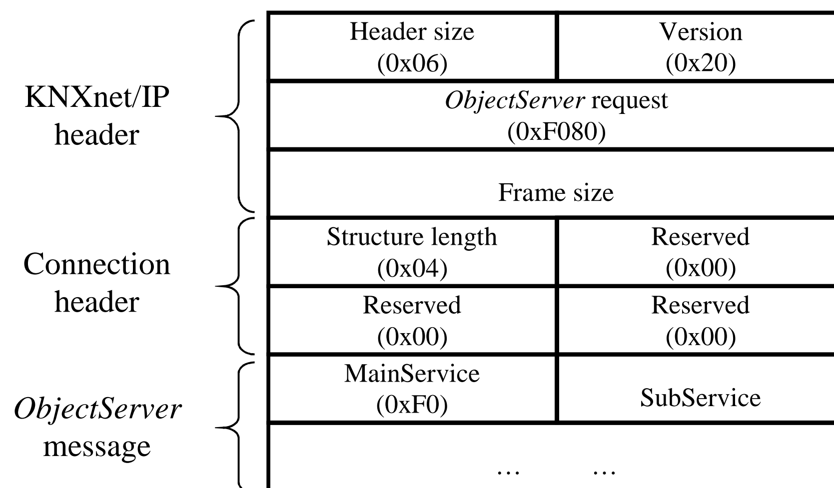

Figure 4: Integration of the ObjectServer message into TCP/IP

Example (GetServerItem):

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

Request:

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

Figure 5: Example (GetServerItem.Req)

Response:

Header										Object Server Message															
KNXnet/IP Header						Connection Header																			
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 6: Example (GetServerItem.Resp)

5. Discovery procedure

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 definite ObjectServer for the communication, alternatively to the manual input from the user. Currently only one discovery procedure is supported, which is based on the KNXnet/IP discovery algorithm. The next chapter describes it briefly. For the full description of the KNXnet/IP discovery algorithm please refer to the KNX handbook Volume 3.8.

5.1. KNXnet/IP discovery algorithm

The KNXnet/IP discovery procedure works in the way showed on the Figure 7. The client, which is looking for the installed ObjectServers, sends a search request via the multicast on the predefined multicast address 224.0.23.12 and port 3671 (decimal). The ObjectServers sends back a search response with the device information block (DIB), which contains among other things the information about the support of the ObjectServer protocol.

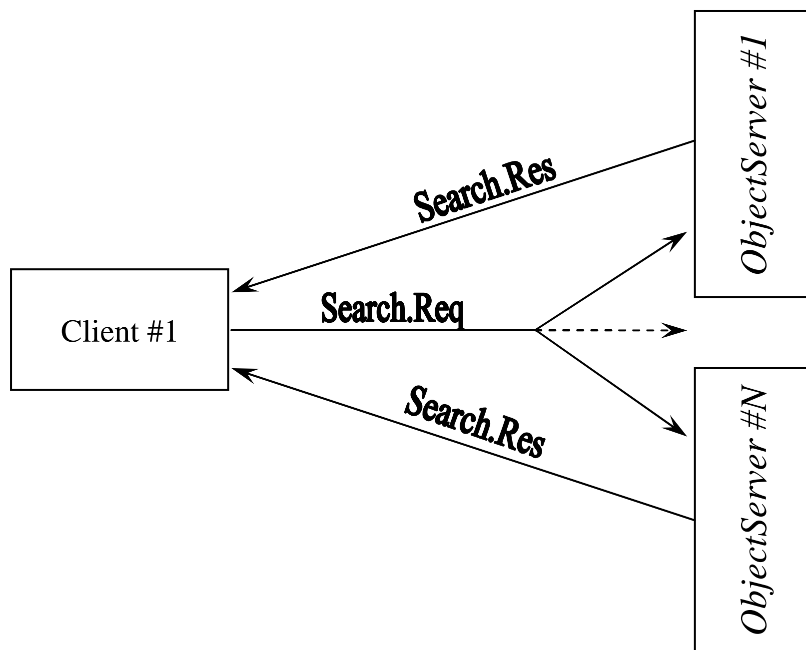

Figure 7: KNXnet/IP discovery

The search request has the length of 14 bytes and its format is presented on Figure 8. 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. For fields, which are longer than one byte, the big-endian format is applied.

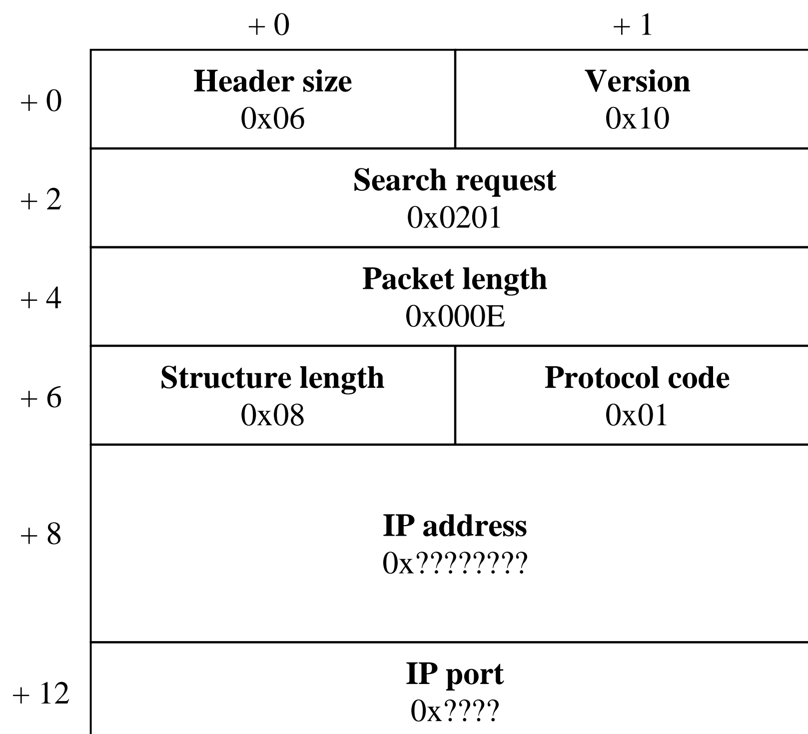

Figure 8: Structure of the Search.Req packet

The search response from the ObjectServer has in the version 1.0 of the protocol the length of 84 bytes and its format is presented on Figure 9. The support of the ObjectServer protocol by the device is indicated through the existence of the manufacturer DIB at the 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 9: Structure of the Search.Res packet

Appendix A. Item IDs

Following items are present in all device types and protocol versions:

ID	Item	Size in bytes	Ac-ces s	Ind.
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 Ex.: 0x10 = Version 1.0	1	R	N
3	**Firmware version** Version of the ObjectServer firmware Coding Ex.: 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	**Maximal 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” – 19200 “2” – 115200	1	RW	N
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 Ex.: 0x20 = Version 2.0	1	R	N
17	**Indication Sending** Values (bit 0): “0” – not active “1” – active	1	RW	N

Following items are optional and can be fully or partly implemented in some device types:

ID	**Item**	**Size in bytes**	**Access**	**Ind.**
18	**Protocol Version (WebService)** Version of the ObjectServer protocol via web services Coding Ex.: 0x20 = Version 2.0	1	R	N
19	**Protocol Version (RestService)** Version of the ObjectServer protocol via rest services Coding Ex.: 0x21 = Version 2.1	1	R	N
20	**Individual Address** The individual KNX address of the device	2	RW	Y
21	**Mac Address**	6	R	N
22	**Tunnelling Enabled** KNXnet/IP tunneling active or not Values: “0” – disabled “1” – enabled	1	RW	Y
23	**Baos Binary Enabled** Access via BAOS Binary connection available or not Values: “0” – disabled “1” – enabled	1	RW	Y
24	**Baos Web Enabled** Web Services active or not Values: “0” – disabled “1” – enabled	1	RW	Y
25	**Baos Rest Enabled** REST services active or not	1	RW	Y

	Values: "0" – disabled "1" – enabled			
26	**Http File Enabled** Webserver active or not Values: "0" – disabled "1" – enabled	1	RW	Y
27	**Search Request Enabled** Device responds to search requests(yes / no) Values: "0" – disabled "1" – enabled	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	Y
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

Attention: For values, which are longer than one byte, the big-endian format is applied.

Appendix B. Error codes

Error code	Description
0	No error
1	Internal error
2	No element found
3	Buffer is too small
4	Item is not writeable
5	Service is not supported
6	Bad service parameter
7	Bad ID
8	Bad command / value
9	Bad length
10	Message inconsistent
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..254	Reserved
255	Unknown DPT

Appendix E. 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). As the hardware interface for the transmission is the Universal Asynchronous Receiver Transmitter (UART) used. The frame format for the FT1.2 protocol is fixed to the 8 data bits, 1 stop bit and even parity bit. The default communication speed is 19200 Baud.

D.1. Communication procedure

The typical communication procedure between the host and the *ObjectServer* is shown on Figure 10.

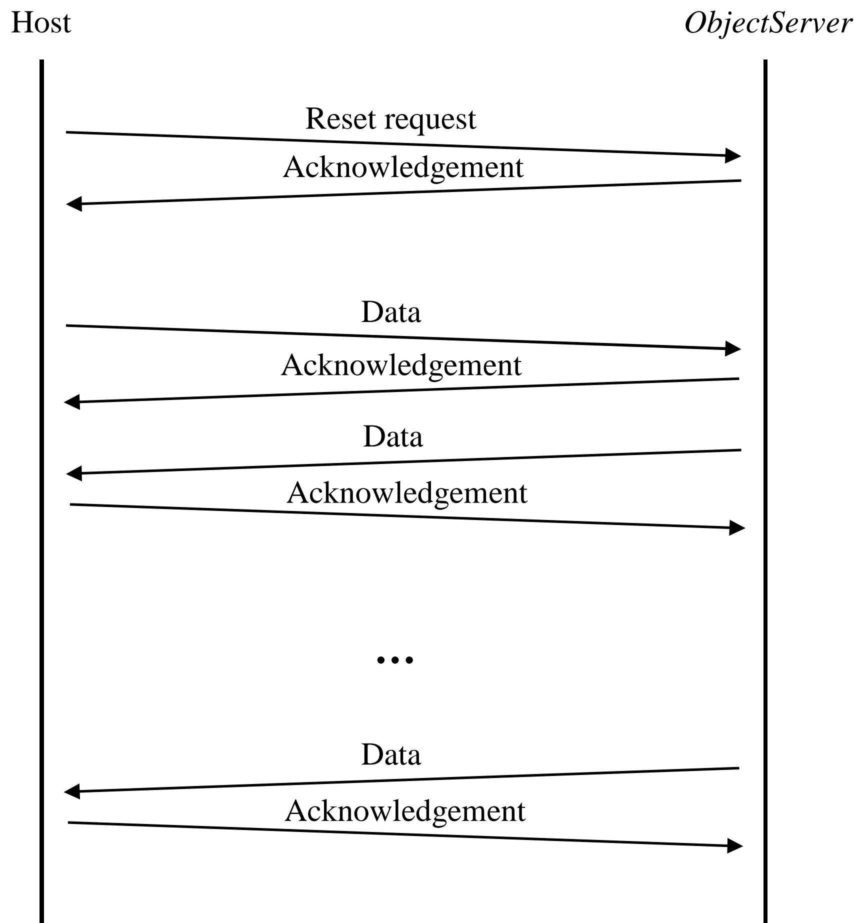

Figure 10: Typical communication procedure

In chapter D.3 is presented an example of the communication between the host and the *ObjectServer*.

D.2. Frame format

Three frame types are defined by the FT1.2 protocol .

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

The second frame type is 4 bytes length and is used for the reset request and reset indication messages (Figure 11).

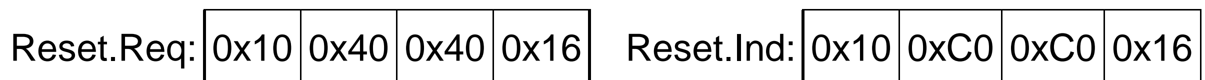

Figure 11: Structure of the Reset.Req and Reset.Ind frames

The third frame type is variable length and used for the data messages. The frame structure is presented on Figure 12.

Figure 12: Structure of the data message

The both fields L contain the length of the data in this frame +1 for the control byte.

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

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

D.3. 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 0x06 0x06 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}