KNX BAOS Starter Kit

For

- KNX BAOS Modules 830 (TP)
- KNX BAOS Modules 832 (TP)
- KNX BAOS Modules 838 (TP, kBerry)
- KNX BAOS Modules 840 (RF)

User's Guide

WEINZIERL ENGINEERING GmbH
Achatz 3
84508 Burgkirchen / Alz
GERMANY
E-Mail: info@weinzierl.de
Web: www.weinzierl.de

KNX BAOS Starter Kit

User's Guide

The KNX BAOS Modules 830, 832, 838 and 840 allow a jump start into KNX device development with minimal invest. All versions include a KNX transceiver and a micro-controller with certified KNX Stack. The communication with the module is performed via a serial interface (UART/TTL) based on FT1.2 frame format. The module provides access to communication objects (application layer) as well as to KNX telegrams (link layer).

The KNX BAOS Module 830 provides electrical isolation and is suitable for devices with separate power supply. The KNX BAOS Module 832 offers direct coupling and provides power for the application from the bus.

The KNX BAOS Module 838 is for use with the Raspberry Pi and provides electrical isolation, also.

For the development, tools and a generic ETS entry with up to 1000 group objects (communication objects) are provided. For a quick start in the development, we recommend our starter kit with a demo project in source code for Atmel Cortex micro-controller and GNU compiler.

The KNX BAOS Modules are suitable for the development of KNX devices with small and medium quantities. By using the BAOS modules the development of KNX certified devices with an individual ETS database is possible of course. The protocol description and our demo application can be found at our web page.

For products with higher volume integration with a KNX stack could be an alternative. We will advice you on request.

For comments or questions please feel free to contact support@weinzierl.de.

Contents

Document history

State	Date	Author
Release	2016-12-22	Gi
Fixed some references	2017-02-07	Gi

1 Preface

1.1 About Us

1.1.1 The Company

Weinzierl Engineering GmbH develops software and hardware components for building control systems. The focus of our activities is Building Automation based on KNX Technology. Thanks to our specialization in this field we are able to offer a comprehensive range of products supporting the KNX Standard. We can advise you in the conceptual phase and develop all aspects of hardware, firmware and application software according to your requirements, including certification of your products with the KNX Association. In addition we develop and produce devices under our own name as well as OEM products. For any questions feel free to contact support@weinzierl.de.

1.1.2 Our Services and Products

- **KNX Devices**

  As a solutions provider we offer KNX system devices mainly with high complexity like interfaces and gateways. The devices are available under the Weinzierl brand as well as OEM versions with individual design.

- **Modules for KNX**

  Our range of KNX modules allows a fast integration of KNX protocol in devices. Especially for low volumes or for the extension of existing devices a module based approach often is an optimal solution.

- **Stacks for KNX**

  KNX describes a complex protocol, which means a considerable effort in the implementation and certification. With our KNX Stacks we offer complete solutions for professional device design.

- **Software for KNX**

  This is how we complete our offer for the development of KNX: A variety of tools and software development kits (SDKs) enables and facilitates the development of KNX client applications and tools.

- **Services for KNX**

  We advise you on the system design and provide on request the full development of hardware, firmware and application software. We develop a complete solution as full service or in co-operation with your development department.

- **Test laboratory for KNX**

  Our KNX Test Lab offers the complete service for KNX certification of your products.

1.1.3 Our focus: KNX

KNX has developed into one of the most important standards for home & building control and is the first worldwide standard to be compliant with EN and ISO/IEC. By building on our extensive experience we are able to offer the components and tools necessary for KNX development. Our product spectrum centers on our stack implementations for the various standardized device models and media of the KNX specification.

For more information about KNX-systems, see the KNX web site.

1.2 Feedback

In case of any errors, misspelled text or other bugs in this document, hardware or software, please contact support@weinzierl.de.

1.3 License Agreement

PLEASE READ THIS LICENSE AGREEMENT CAREFULLY BEFORE USING THE SOFTWARE OF WEINZIERL ENGINEERING GMBH. BY USING THE SOFTWARE YOU ARE AGREEING TO THE CONDITIONS OF THIS LICENSE AGREEMENT. DO NOT USE THE SOFTWARE IF YOU DO NOT AGREE THE TERMS OF THIS LICENSE AGREEMENT. IN THIS CASE YOU MAY RETURN THE COMPLETE PACKAGE WITHIN A PERIOD OF TWO WEEKS WHERE YOU PURCHASED IT.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANT-ABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.3.1 Definitions

"Firmware" means the software already stored into the micro controller of the KNX BAOS Module.

"SDK" is a Software Development Kit, which means all provided software to develop, write and compile software for the target which is using the KNX BAOS Module. This means the Demo Sources, the ETS®/MT® projects and databases and the documentation.

"Tools" are the provided software tools by Weinzierl Engineering GmbH. This means Net'n Node and TraceMon.

1.3.2 Permitted Uses

Subject to the terms and conditions of this agreement and restrictions and exceptions, Weinzierl Engineering GmbH grants you a non-exclusive, non-transferable, limited license without fees to

a) reproduce and use internally the SDK and Tools for the purposes of developing applications that communicate with KNX BAOS Modules from Weinzierl Engineering GmbH.
b) develop and distribute all software done with the SDK, but not the SDK itself.
c) reproduce and distribute the resulting software in binary form and resulting ETS databases for the sole purpose of running your application.

1.3.3 Restrictions

a) The Weinzierl Firmware (installed in flash of the KNX BAOS Module) is limited to be used on Weinzierl modules. It is not allowed to distribute it.
b) The Weinzierl SDK is limited to be used for Weinzierl modules or for use on appropriate computers/hardware in conjunction to these Weinzierl modules. It is not allowed to distribute the Weinzierl SDK.
c) You may not and you agree not to, or to enable others to, duplicate, de-compile, reverse engineer, disassemble, attempt to derive the source code of, de-crypt, modify, or create derivative works of the Weinzierl firmware, or any part thereof.

1.3.4 Overview of Restrictions/Permissions

Subject	Restrictions/Permissions
Firmware (in module)	**Not** allowed to copy.
SDK and Tools	Allowed to copy only **internally**.
Source code of demo application and ETS databases	Allowed to copy in **binary** form in conjunction with **Weinzierl hardware**.

2 Overview

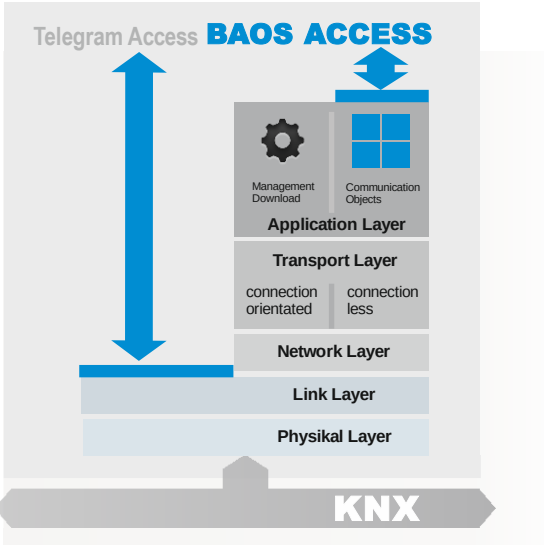
KNX is a well-established standard for modern electrical devices in house installations. It connects the devices by a bus system and thus all can communicate to each other. This communication is implemented by KNX messages which are sent via the bus. Furthermore the devices are powered by the bus (except for the RF 840).

Once installed, the devices must be configured and commissioned by *ETS*®. This connects the devices to each other in a logical way. E. g. which switch turns on what light? ETS is a standardized tool by the KNX organization.

All devices are categorized in *sensors*, *actuators* or both. Sensors tell other devices what to do (e. g. light switch, dimmer, heat control). Actuators are devices which receive messages from sensors and act accordingly (e. g. light, shutter).

The purpose of the KNX BAOS Modules is to help developing KNX hardware with little effort (the hardware must only support a UART serial interface at TTL level 3.3 V). The KNX BAOS Module is connected to the KNX bus. It handles the whole KNX communication, configuration and management. The other side, the application, must implement the communication to the KNX BAOS Modules. An example application is included in this starter kit.

The KNX BAOS Module serves as an interface access at the *telegram* and *BAOS* level. The telegram access is for more experienced usage and offers the possibility to manage KNX messages by the application itself.



The BAOS access uses data points (communication objects) for communication. A data point represents a numeric value which automatically generates (if configured so) KNX activity by change. This works also vice versa. If any KNX activity changes the value of the data point, the application will be notified.

This way, the application can interact with all devices on the KNX bus.

Structure of this manual:

- The first chapter "Quick Start" of this manual are for quick entry into the world of BAOS. It is not essential for the understanding of BAOS, but for a fast commissioning of the hardware.
- The next chapters, starting from "The Starter Kit" ending at "BAOS Protocol", are the main part of this manual. They tell everything about working with the hardware and developing BAOS applications.
- The remaining chapters, starting at "About KNX", are for advanced usage. They tell more about the background of KNX, ETS and creating own ETS databases.

3 Quick Start

This chapter is a quick start for commissioning the hardware.

3.1 Standard Kit (BAOS Modules 830 and 832)

The BAOS Starter Kit contains the following hardware parts:

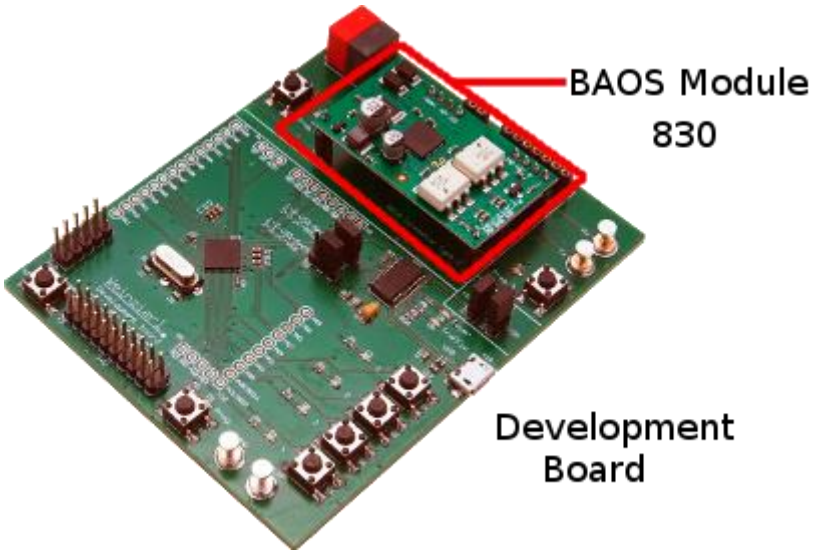
- Two KNX BAOS Modules 830 and 832.
- One Development Board.

3.1.1 Download the Software

The software and documentation are available for download at the Weinzierl web page at http://www.weinzierl.de. Download the following and you can start:

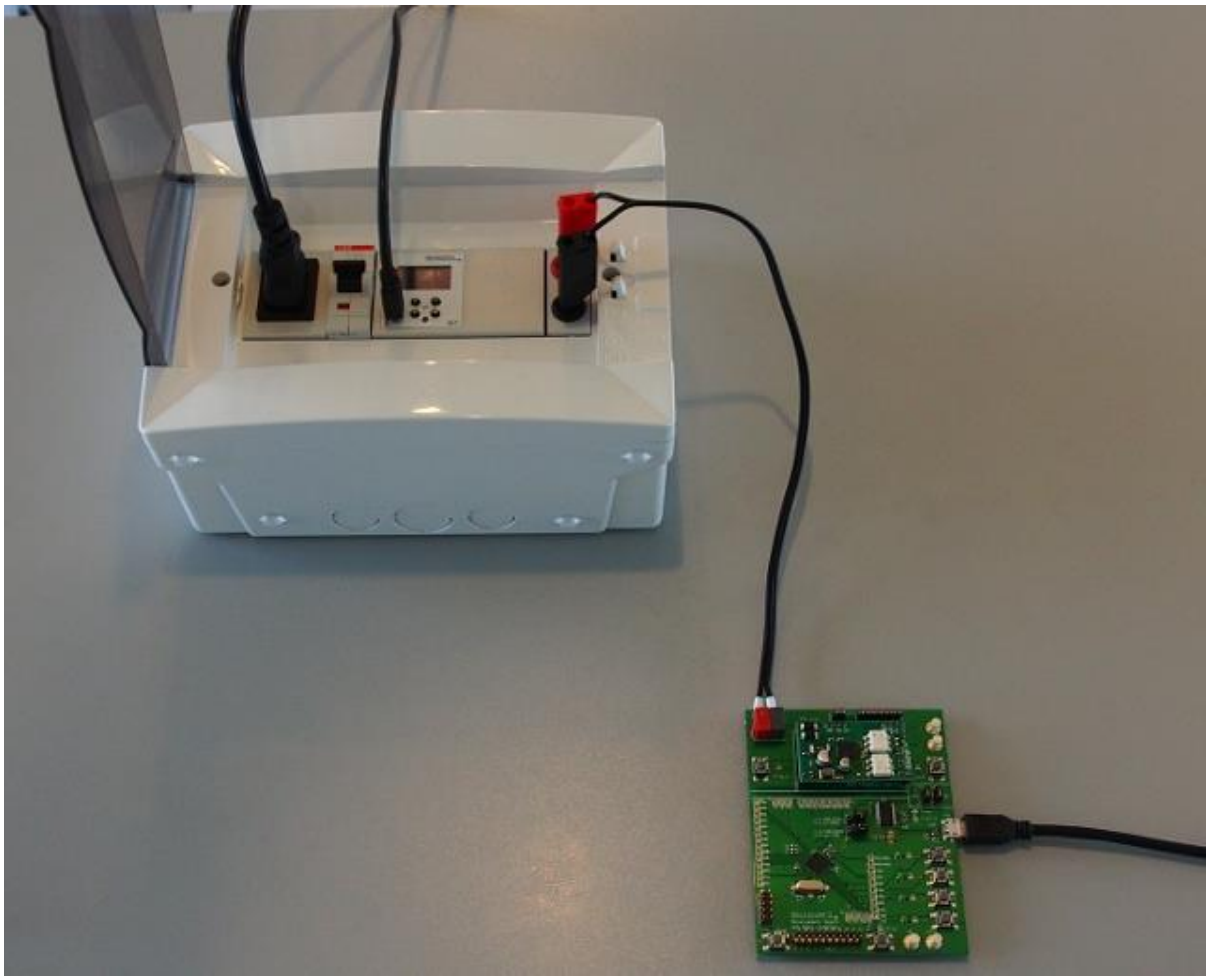
- **KnxBAOS 83x ETS Projects for Demo.**
  ETS is the standard configuration software for all KNX devices. For now only the generic solution is needed.

3.1.2 Hardware Setup



Assemble the KNX BAOS Module 830 to the Development Board as shown in the figure above. Make sure, the jumpers are set as follows:

- X13 and X14 (Vcc-Sel) must be closed.
- X8 (µC-BAOS) must be closed at position 2-3.
- X9 (µC-USB) must be closed at position 2-3.



To take the Development Board with the KNX BAOS Module 830 in operation state, the following items and steps are necessary:

- A KNX power supply with choke.
- A KNX USB interface to configure (commission) the board via ETS.
- Make sure the KNX BAOS Module is correctly connected to the Development Board.
- Connect the KNX bus (polarity is protected).
- Connect Micro-USB to a PC to power the Development Board (not in case of Module 832).

3.1.3 First Commissioning with ETS

ETS (Engineering Tool Software) is a manufacturer independent configuration tool to configure KNX systems. It can be downloaded from the KNX-Association page at http://www.knx.org. A more detailed introduction is in chapter "Commissioning with ETS".
Configure ETS to use the KNX USB Interface in the **Bus** folder. Look into the list of **Discovered Interfaces** and select the one, which is connected to the KNX bus. **Test** and **Select** the interface. Go back to the **Overview** folder.

For demonstration a simple project for 830 is available in the archive

Weinzierl_83x_KNX_BAOS_ETS_Projects_for_Demo.zip.

Unpack it and import the project

Weinzierl_83x_KNX_BAOS_ETS_Projects_for_Demo/
ETS_Project_using_generic_ETS_entry/
Project.knxproj

in ETS.

The project configures the board to simply handle its own LED. Open the project, select **Project Root** as view, select the device **KNX BAOS 830** with right mouse button and **Download/Full download**.

- Press the learn button S1 on the Development Board (red LED must light up).
  (In case of Module 832, use button S8.)
- After the download is finished, press the two push buttons S4 and S5 on the Development Board to switch the LED D3 on and off.

The Development Board acts now as both sensor and actuator. Pushing the buttons S4 and S5, the BAOS Module generates a KNX telegram on the bus. This can be verified this by selecting **Diagnostics** in ETS, **start**ing the **Group Monitor** and watching the telegrams. Furthermore ETS can generate a telegram to switch the LED on or off: Use **Group Address** 3/3/1, set the **Value** to 1 (on) or 0 (off) and select **Write**. The LED should act accordingly.

3.2 Standard Kit (BAOS Module RF 840)

The BAOS Starter Kit contains the following hardware parts:

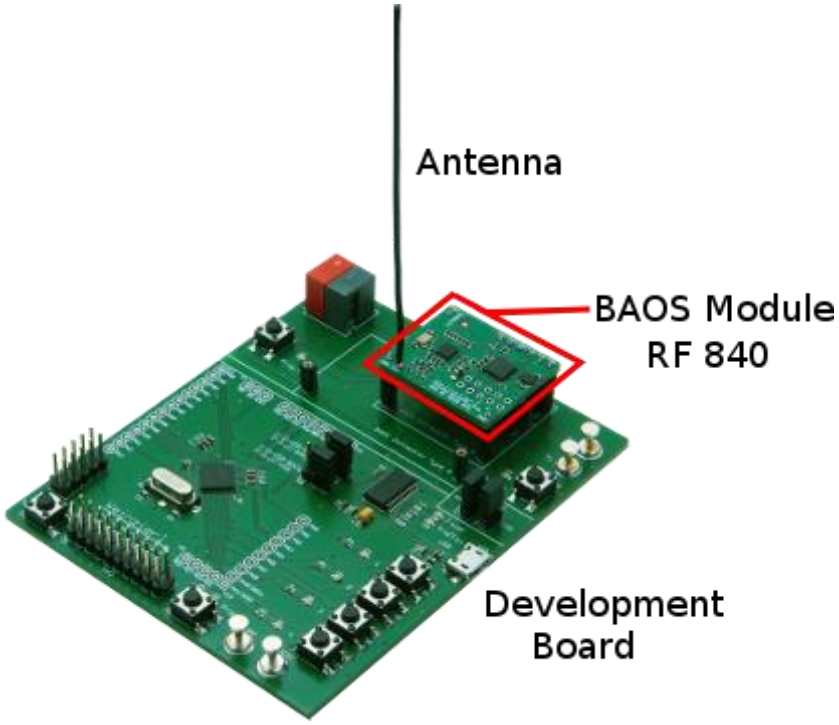
- One KNX BAOS Module RF 840.
- One Development Board.

3.2.1 Download the Software

The software and documentation are available for download at the Weinzierl web page at http://www.weinzierl.de. Download the following and you can start:

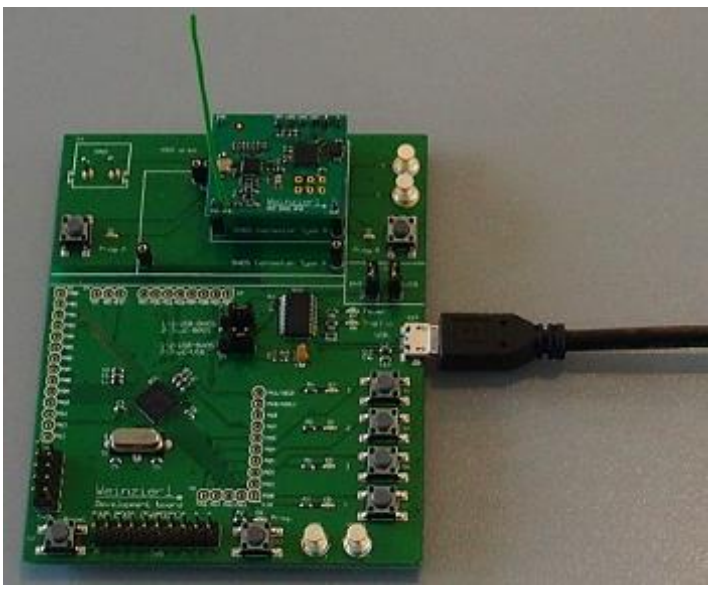
- **KnxBAOS 840 ETS Projects for Demo.**
  ETS is the standard configuration software for all KNX devices. For now only the generic solution is needed.

3.2.2 Hardware Setup



Assemble the KNX BAOS Module RF 840 to the Development Board as shown in the figure above. Make sure, the jumpers are set as follows:

- X13 and X14 (Vcc-Sel) must be closed.
- X8 (µC-BAOS) must be closed at position 2-3.
- X9 (µC-USB) must be closed at position 2-3.



To take the Development Board with the KNX BAOS Module RF 840 in operation state, the following items and steps are necessary:

- A KNX RF USB interface to configure (commission) the board via ETS.
- Make sure the KNX BAOS Module is correctly connected to the Development Board.
- Connect Micro-USB to a PC to power the Development Board.

3.2.3 First Commissioning with ETS

ETS (Engineering Tool Software) is a manufacturer independent configuration tool to configure KNX systems. It can be downloaded from the KNX-Association page at http://www.knx.org. A more detailed introduction is in chapter "Commissioning with ETS".
Configure ETS to use the KNX USB Interface (RF) in the **Bus** folder. Look into the list of **Discovered Interfaces** and select the one, which uses the RF KNX bus. **Test** and **Select** the interface.

Warning: Use the same domain address for the RF interface as configured in the project: FFFF:FFFFFFFF.

Go back to the **Overview** folder.

For demonstration a simple project for 840 is available in the archive

Weinzierl_840_KNX_BAOS_ETS_Projects_for_Demo.zip.

Unpack it and import the project

Weinzierl_840_KNX_BAOS_ETS_Projects_for_Demo/
ETS_Project_using_generic_ETS_entry/
Project.knxproj

in ETS.

The project configures the board to simply handle its own LED. Open the project, select **Project Root** as view, select the device **KNX BAOS 840** with right mouse button and **Download/Full download**.

- Press the learn button S8 on the Development Board (red LED must light up).
- After the download is finished, press the two push buttons S4 and S5 on the Development Board to switch the LED D3 on and off.

The Development Board acts now as both sensor and actuator. Pushing the buttons S4 and S5, the BAOS Module generates a KNX telegram on the bus. This can be verified this by selecting **Diagnostics** in ETS, **start**ing the **Group Monitor** and watching the telegrams. Furthermore ETS can generate a telegram to switch the LED on or off: Use **Group Address** 3/3/1, set the **Value** to 1 (on) or 0 (off) and select **Write**. The LED should act accordingly.

3.3 Kit for Raspberry Pi (BAOS Module 838)

If you purchased the kBerry (KNX BAOS Module 838), you have one hardware, only:

- KNX BAOS Module 838.

You also need the Raspberry Pi, which you have to purchase in an appropriate hardware shop.

The software and documentation are also available for download at the Weinzierl web page at http://www.weinzierl.de. Download the following and you can start:

- **KnxBAOS 83x ETS Projects for Demo.**
  ETS is the standard configuration software for all KNX devices. For now only the generic solution is needed.
- **Net'n Node Busmonitor Software.**
  A powerful bus monitor and analyser for the development of KNX devices for all KNX media. This tool helps you to understand the KNX communication. It can also communicate directly with the BAOS Module for testing purposes.

Continue reading the chapter "Programming the Raspberry Pi Board".

4 The Starter Kit

This chapter is about the contents of the Starter Kit: Hard- and software components.

4.1 Standard Kit (BAOS Modules 830 and 832)

The complete BAOS Starter Kit contains the following hardware parts:

- **Two KNX BAOS Modules: 830 and 832.**
  These modules are for developing own KNX hardware. The KNX system is implemented in these modules and separates your application from the KNX system, so you do not have to care about it. For experts it is also possible to bypass parts of this system and use the KNX communication more directly.
- **One Development Board.**
  This board is for implementing your own application and to take your first steps into the world of KNX. A demo application is also available. This board can be replaced by your own hardware, later.

The software and documentation are available for download at the Weinzierl web page at http://www.weinzierl.de. Download the following files and you can start:

- **BAOS User's Guide.**
  This document.
- **BAOS V2 Protocol Description.**
  The BAOS protocol is used for communication between the BAOS Module and your application. This document describes the protocol.
- **Data sheet KNX Module 830.**
  Technical specification about the module with galvanic isolations.
- **Data sheet KNX Module 832.**
  Technical specification about the module.
- **KnxBAOS 83x ETS Projects for Demo.**
  ETS is the standard configuration software for all KNX devices. This project contains two aspects:
  - **using generic ETS entry:**
    It is an example with pre-configured data points and parameters using a generic ETS entry for KNX BAOS Modules. The generic ETS entry is intended for development, if an individual ETS entry is not yet available.
  - **using individual ETS entry:**
    It is an example with an individual ETS entry. It has been created with KNX manufacturer tool MT. The corresponding MT project files are included, too. MT is the standard software by the KNX organization to create own ETS databases for your own devices.

- **KnxBAOS Demo Source.**
  A demo application to communicate to the BAOS Module. You can use this demo application as your base for your own application. It is an Atmel Studio Project in C. A software project with source code. This software contains a serial driver for frame format FT1.2, a client library for the BAOS protocol and a simple demo application. The demo application implements an actuator and sensor channel, both can be configured to switching, dimming and shutter by parameters.
- **Net'n Node Busmonitor Software.**
  A powerful bus monitor and analyser for the development of KNX devices for all KNX media. This tool helps you to understand the KNX communication. It can also communicate directly with the BAOS Module for testing purposes.
- **Product database for ETS 4.2/5.**
  Last, but not least, the ETS product databases for each module are available at their respective pages.

4.2 Standard Kit (BAOS Module RF 840)

The complete BAOS Starter Kit contains the following hardware parts:

- **One KNX BAOS Module RF 840.**
  This module is for developing own KNX hardware. The KNX system is implemented in this module and separates your application from the KNX system, so you do not have to care about it. For experts it is also possible to bypass parts of this system and use the KNX communication more directly.
- **One Development Board.**
  This board is for implementing your own application and to take your first steps into the world of KNX. A demo application is also available. This board can be replaced by your own hardware, later.

The software and documentation are available for download at the Weinzierl web page at http://www.weinzierl.de. Download the following files and you can start:

- **BAOS User's Guide.**
  This document.
- **BAOS V2 Protocol Description.**
  The BAOS protocol is used for communication between the BAOS Module and your application. This document describes the protocol.
- **Data sheet KNX Module 840.**
  Technical specification about the module with galvanic isolations.
- **KnxBAOS 840 ETS Projects for Demo.**
  ETS is the standard configuration software for all KNX devices. This project contains two aspects:
  - **using generic ETS entry:**
    It is an example with pre-configured data points and parameters using a generic ETS entry for KNX BAOS Modules. The generic ETS entry is intended for development, if an individual ETS entry is not yet available.

    - **using individual ETS entry:**
      It is an example with an individual ETS entry. It has been created with KNX manufacturer tool MT. The corresponding MT project files are included, too. MT is the standard software by the KNX organization to create own ETS databases for your own devices.
- **KnxBAOS Demo Source.**
  A demo application to communicate to the BAOS Module. You can use this demo application as your base for your own application. It is an Atmel Studio Project in C. A software project with source code. This software contains a serial driver for frame format FT1.2, a client library for the BAOS protocol and a simple demo application. The demo application implements an actuator and sensor channel, both can be configured to switching, dimming and shutter by parameters.
- **Net'n Node Busmonitor Software.**
  A powerful bus monitor and analyser for the development of KNX devices for all KNX media. This tool helps you to understand the KNX communication. It can also communicate directly with the BAOS Module for testing purposes.
- **Product database for ETS 5.**
  Last, but not least, the ETS product databases for each module are available at their respective pages.

4.3 Controller only Solution

The controller contains the same KNX BAOS software as the Modules, so all chapters about ETS, Net'n Node, BAOS protocol, KNX, FT1.2 protocol, common EMI and Individual ETS entries apply to you.

4.4 Kit for Raspberry Pi (BAOS Module 838)

If you purchased the kBerry (KNX BAOS Module 838), you have one hardware, only:

- **KNX BAOS Module 838.**
  This module connects the Raspberry Pi to the KNX bus. The KNX system is implemented in this module and separates your application from the KNX system, so you do not have to care about it. For experts it is also possible to bypass this system and use the KNX communication directly.

You also need the Raspberry Pi, which you have to purchase in an appropriate hardware shop.

The software and documentation are also available for download at the Weinzierl web page at http://www.weinzierl.de. Download the following files and you can start:

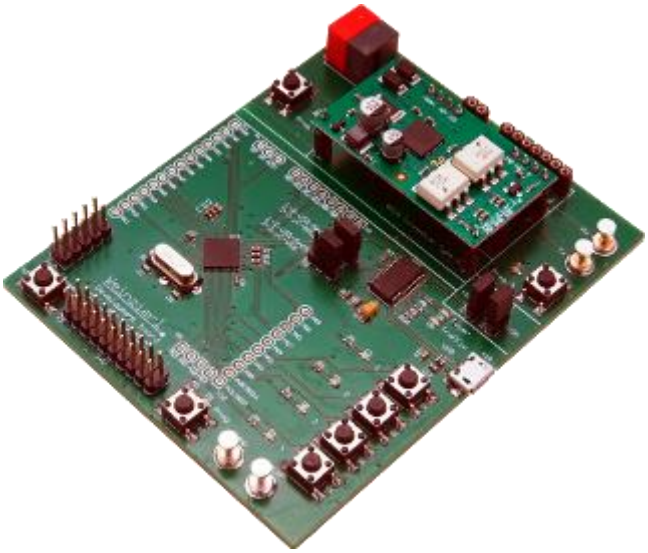
- **BAOS User's Guide.**
  This document.
- **BAOS V2 Protocol Description.**
  The BAOS protocol is used for communication between the BAOS Module and your application. This document describes the protocol.

- **Data sheet KNX Module 838.**
  Technical specification about the module 838.
- **KnxBAOS 83x ETS Projects for Demo.**
  ETS is the standard configuration software for all KNX devices. This project contains two aspects:
  - **using generic ETS entry:**
    It is an example with pre-configured data points and parameters using a generic ETS entry for KNX BAOS Modules. The generic ETS entry is intended for development, if an individual ETS entry is not yet available.
  - **using individual ETS entry:**
    It is an example with an individual ETS entry. It has been created with KNX manufacturer tool MT. The corresponding MT project files are included, too. MT is the standard software by the KNX organization to create own ETS databases for your own devices.
- **KnxBAOS Sources for Raspberry Pi.**
  The software development kit (SDK) is not available at the Weinzierl web site. It is hosted at GitHub. To read more about the Raspberry Pi and how to download this software, see chapter "Programming the Raspberry Pi Board".
- **Net'n Node Busmonitor Software.**
  A powerful bus monitor and analyser for the development of KNX devices for all KNX media. This tool helps you to understand the KNX communication. It can also communicate directly with the BAOS protocol for testing purposes.
- **Product database for ETS 4.2/5.**
  Last, but not least, the ETS product databases for each module are available at their respective pages.

In case of using the Raspberry Pi, you can skip the chapters "The Development Board", "Programming the Development Board" and "The Demo Application ".

5 The Development Board

This chapter introduces the Development Board, its application software and how to use its connections.



The Development Board is for development and testing own software applications for its capability using KNX. It offers various input/output elements connected to a freely programmable micro-controller.

5.1 Introduction

The Development Board makes the entry into the KNX development as easy as possible. It uses the 830, 832 or 840 module which contains the communication stack. The Development Board contains a 32 bit micro-controller and additional elements like LEDs and buttons.

Every Development Kit consists of two parts:

1. The Development Board.
2. The KNX BAOS Modules, which are located on the connectors of the Development Board.

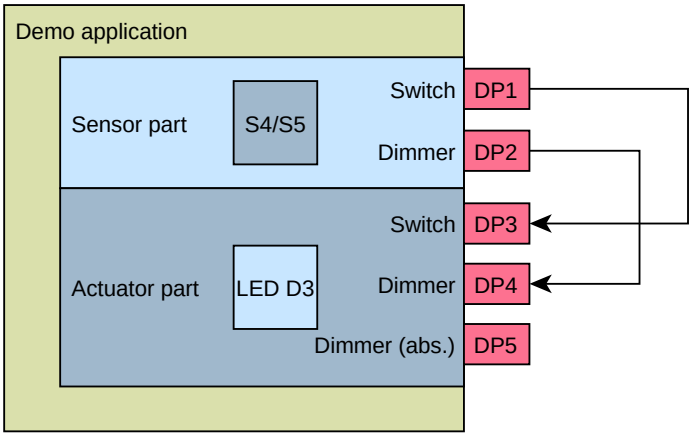
5.2 KNX BAOS Module

The BAOS Module is located on the Development Board. It is responsible for the whole KNX communication and comes with a certified KNX stack. It is configured by ETS and handles the KNX communication. It is connected via a serial port (UART) to the Development Board.

KNX BAOS Modules: left 830, middle 832, right 840.

For more information see chapter "KNX BAOS Modules".

5.3 The Demo Application



The demo application of the Development Board is a two channel dimming actuator and sensor. The buttons S4 and S5 are used as output channel. They use the data points 1 and 2 for sending switching and dimming telegrams respectively. The state of the actuator channel is shown by the LED D3. Data points 3 and 4 accept switching and dimming telegrams respectively. Data point 5 accepts an absolute dimming value.

5.3.1 Data Points/Group Objects

The data points (DP) are accessible as follows:

DP#	DPT	Size	Sensor/Actuator	Development Board
1	1.001	1 bit	Sensor	Switch S4, S5 on/off. This simply switches a light on and off.
	1.007	1 bit	Sensor	Shutter step S4, S5 open/close. For moving a shutter just one step.
2	3.007	4 bit	Sensor	Dimming S4, S5 brighter/darker. This dims a light relatively.
	1.008	1 bit	Sensor	Shutter move S4, S5 up/down.

				For starting or stopping movement of a shutter.
3	1.001	1 bit	Actuator	Switch LED D3 on/off.
4	3.007	4 bit	Actuator	Dimming LED D3 relatively (up/down).
5	5.001	8 bit	Actuator	Dimming LED D3 absolutely.

The demo application uses data point #1 connected to data point #3, which simply switches the LED. The dimming feature is used by connecting data point #2 and #4. Dimming is triggered by pressing S4 or S5 long. Data point #5 is not used here, but can be used if a sensor sends absolute dimmer values.

5.3.2 Parameters

The KNX BAOS Module supports parameter bytes which can be written by the ETS. Each parameter can be read by the application via the BAOS protocol. The demo application reads parameter #1 and #2 at program start and after an ETS download. The following tables show the meaning of these parameters.

Parameter #1 controls data points #1 and #2 functionalities:

Value	Meaning	DP#1 Function	DP#2 Function
0	disabled	not used	not used
1	switch	used as 1 bit switch sensor (S4, S5)	not used
2	dimmer	used as 1 bit switch sensor (S4, S5)	used as 4 bit dimming sensor (S4, S5)
3	shutter	used as 1 bit shutter step sensor (S4, S5)	used as 1 bit shutter move sensor (S4, S5)

Parameter #2 controls data points #3, #4 and #5 functionalities:

Value	Meaning	DP#3 Function	DP#4 Function	DP#5 Function
0	disabled	not used	not used	not used
1	switch	used as 1 bit switch actuator (LED D3)	not used	not used
2	dimmer	used as 1 bit switch actuator (LED D3)	used as 4 bit relatively dimming actuator (LED D3)	used as 1 byte absolutely dimming actuator (LED D3)

If we set both parameters to 0 the sensor (button S4 and S5) does nothing and the actuator (LED D3) also does nothing. If set to 1, they can switch on and off the light. If set to 2, they can dim the light, additionally.

5.4 Connect and commission the Hardware

Connect and commission the Development Board and BAOS Module as described in the chapter "Quick Start".

Note: If you want to use the BAOS Module 832, don't use the USB connection to the PC for now.

The learn button (or programming button) sets a KNX device into programming mode. In this mode ETS can assign an individual address to the device. In our case it will be 1.1.32. This is a unique address for every KNX device on the installation. The default address of the BAOS Modules is 15.15.255.

5.5 Monitoring KNX using Net'n Node

The BAOS development kit includes the program *Net'n Node*. It is available for download at the BAOS web page. With this program you can communicate with the KNX BAOS Module without ETS. Install it and connect the PC to the KNX bus via bus interface, e. g. KNX USB Interface 0311. (More info available at the Weinzierl web site at http://www.weinzierl.de)

The download file of Net'n Node is an archive file. Unpack it and follow the READE file.

Important: To run Net'n Node a license file is required. It can be obtained from the Registration web site of Net'n Node 5.

Now open your bus interface by hitting one of buttons at the left side.

If the buttons on the Development Board are pressed, Net'n Node shows the telegrams. Near the right end of the table the values can be seen. The current address of our KNX BAOS Module is shown in the source address (**Src-Addr**) column of the table.

Net'n Node is also capable to send telegrams. Select the menu **Send KNX/Group Value Write/DPT 05 - 8-Bit Unsigned Value - 1 byte**. This opens a dialog window where we can select the contents of our telegram. Enter the correct **Group Address** for example **3/3/3**. Enter a value (example **16**) in the **Data** area and click the button **Send**. The LED of the previously configured BAOS Development board dims. Simultaneously the telegrams of the KNX bus are shown in the telegram view.

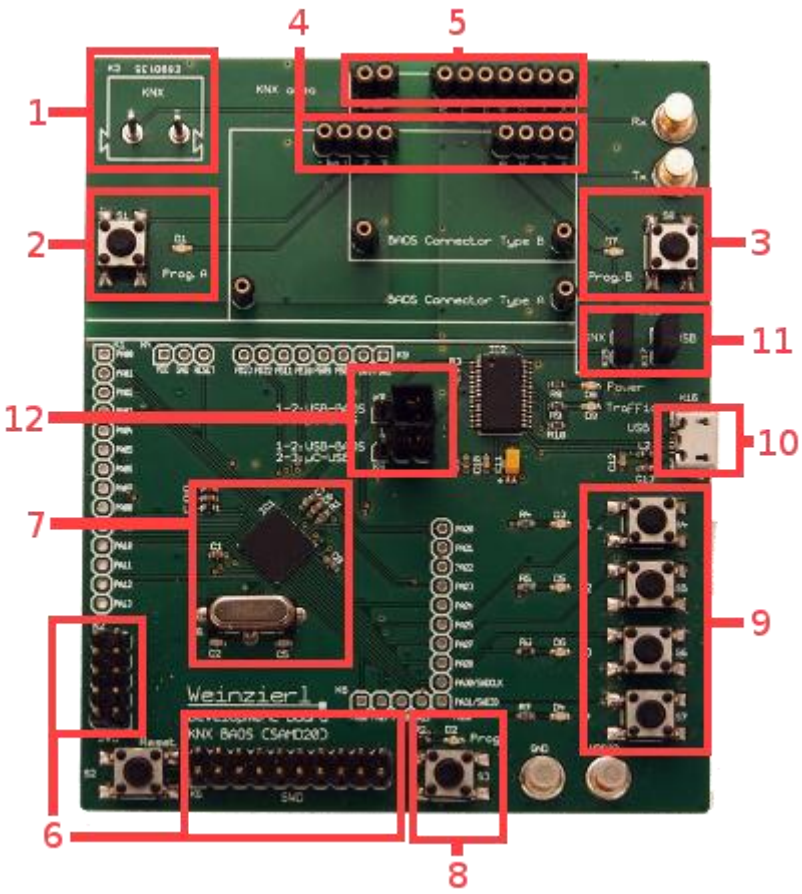
5.6 Development Board Hardware

This section describes the hardware of the Development Board and its usage.

The Development Board contains a micro-controller for the user application, 4 LEDs and 4 push buttons for the application software.

The KNX BAOS Module (mounted on the connectors of the Development Board) contains another micro-controller which handles the KNX stack.

5.6.1 Components



The Development Board contains the following:

1. The connector to the **KNX bus**.
2. **Learning Key** (or Programming Keys) and **LED** for BAOS Module 830.
3. **Learning Key** and **LED** for BAOS Module 832 or 840.
   These keys and LEDs are not for the application. They are used to program the individual address. (e. g. 1.1.32). The Development Board can host different KNX BAOS Modules (830, 832 or 840). Due to different power concepts individual LED/key pairs exist for each form factor.
4. **Connection to the BAOS Module 830**.
5. **Connection to the BAOS Module 832 or 840**.
   See section “Pinning of the KNX BAOS Modules”.
6. **SWD connectors** for programming and debugging the micro-controller of the Development Board.
7. Atmel SAMD20G18 Cortex M0+ **micro-controller** with 256 kB Flash and its 7.3728 MHz **crystal**.
8. Programming **LED and push button for the application software**. This can be used for an application triggered learning mode.
9. **Push buttons and LEDs** for the user application.
10. **Micro-USB connector**: Power supply for the board and UART (FTDI) connector for communication to the application or the KNX BAOS Module. Configurable with jumpers.

11. **Jumpers Vcc-Sel** for power configuration:

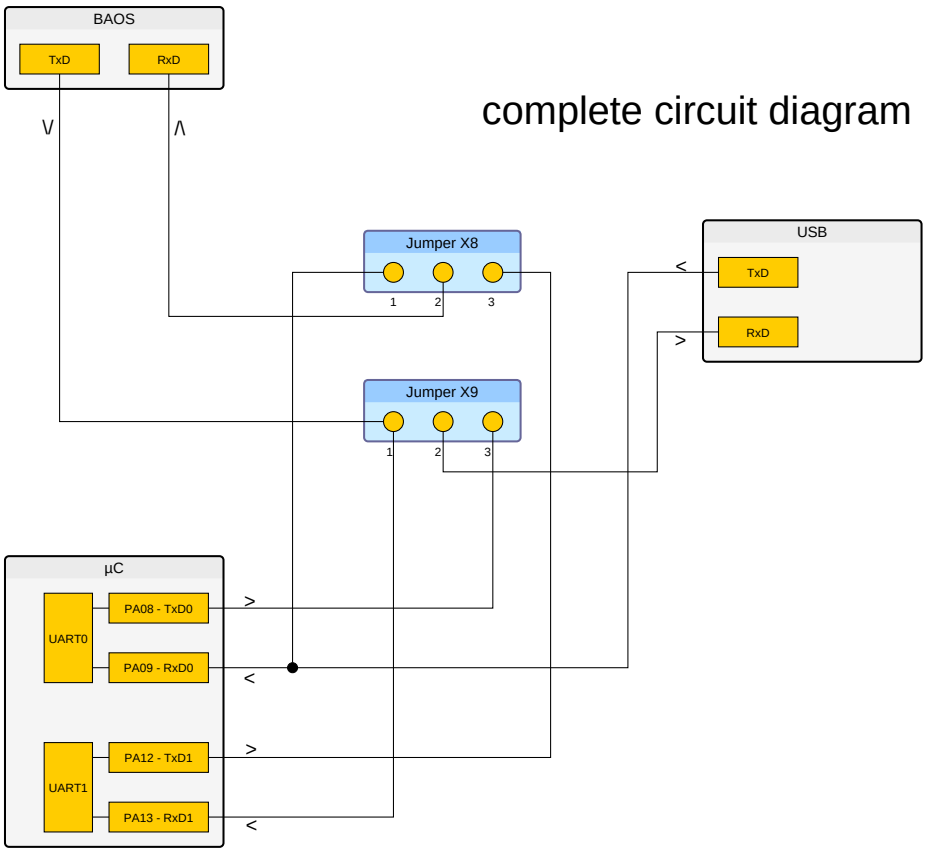
X13 (KNX)	X14 (USB)	Micro-controller	USB connector	Module
open	open	not powered	unusable	unusable
open	closed	powered by PC	must be connected to PC	832
closed	open	powered by KNX	can be connected to PC	832
closed	closed	powered by PC	must be connected to PC	830/840

Note: The first combination (X13 and X14 open) might work with the module 832, but this is not correct due to the power supply for the Development Board is done by the signals of the BAOS Module not the Vcc pin. Don't do this.

12. **Jumpers X8 and X9** for BAOS communication:
Connectors to route the communication between the KNX BAOS Module and the micro controller of the Development Board or the USB port. See more next section.

5.6.2 Jumpers for BAOS Communication

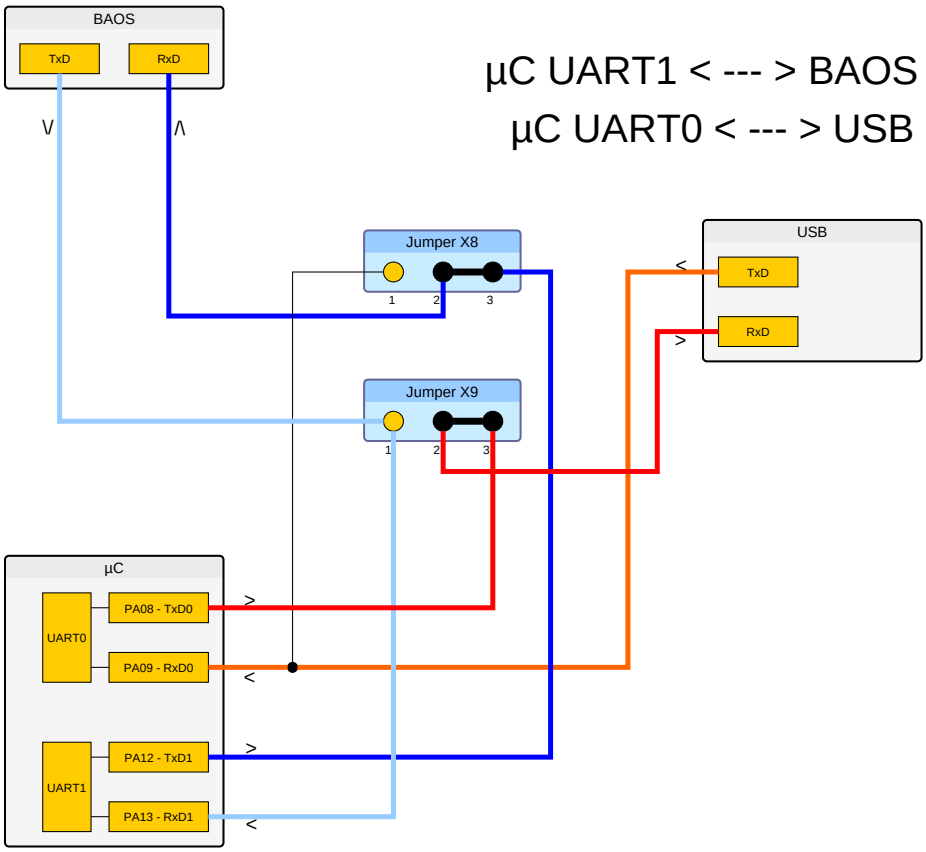
The Jumpers **X8** and **X9** can be used to route the communication between the micro controller of the Development Board, a PC and the BAOS Module. Its schematics are shown below.



The micro controller of the Development Board has two UART interfaces (UART0 and UART1). Both can be used, but each for a certain task and the USB interface can be connected to a PC.

5.6.2.1 Communication: BAOS and Development Board

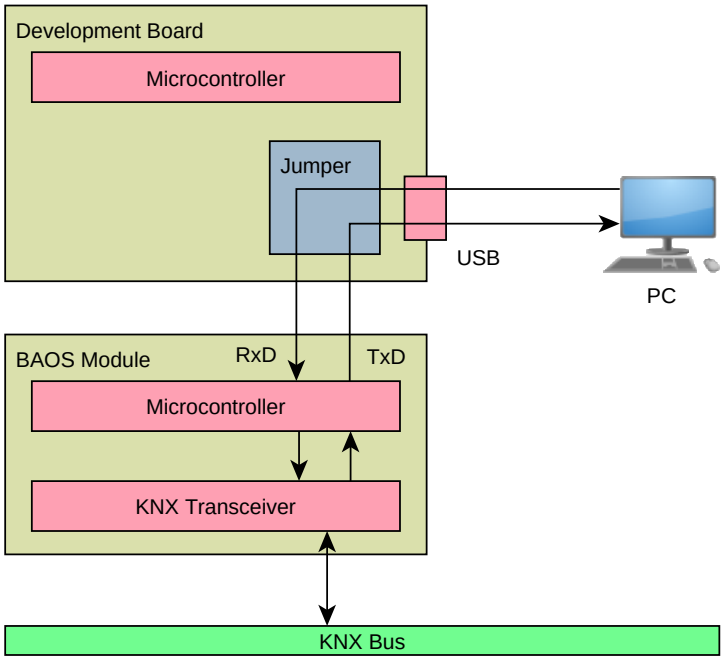
The standard case is to connect the micro controller of the Development Board to the BAOS Module. The controller uses UART1 for this task (blue and light blue lines).



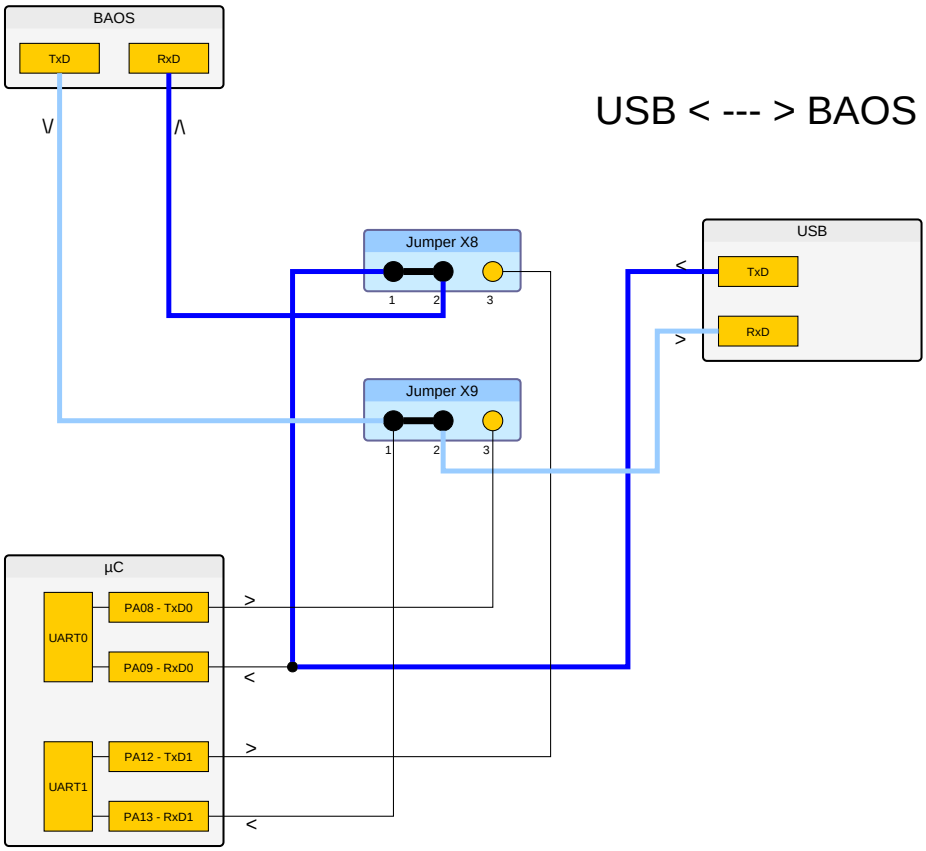
Furthermore it is possible to use UART0 of this micro controller to communicate to a PC via the USB port of the Development Board. This makes some data/command exchange possible to the PC (red and orange lines). In the demo program of the Development Board, this is not used.

5.6.2.2 Communication: BAOS and PC

Developing and debugging an embedded application can be quite inconvenient and difficult. Especially debugging and watching variables are time consuming tasks at an embedded system. To avoid this, the application can also be developed on a PC. It can be connected to the KNX BAOS Module the way the embedded micro controller is.



To do this, connect the USB interface of the PC to the Development Board and set the jumpers shown below.



This enables the BAOS communication directly to the PC (blue and light blue lines). Net'n Node can also use the direct communication to the KNX BAOS Module. The micro controller of the Development Board cannot communicate to the BAOS Module, but it can still listen to it.

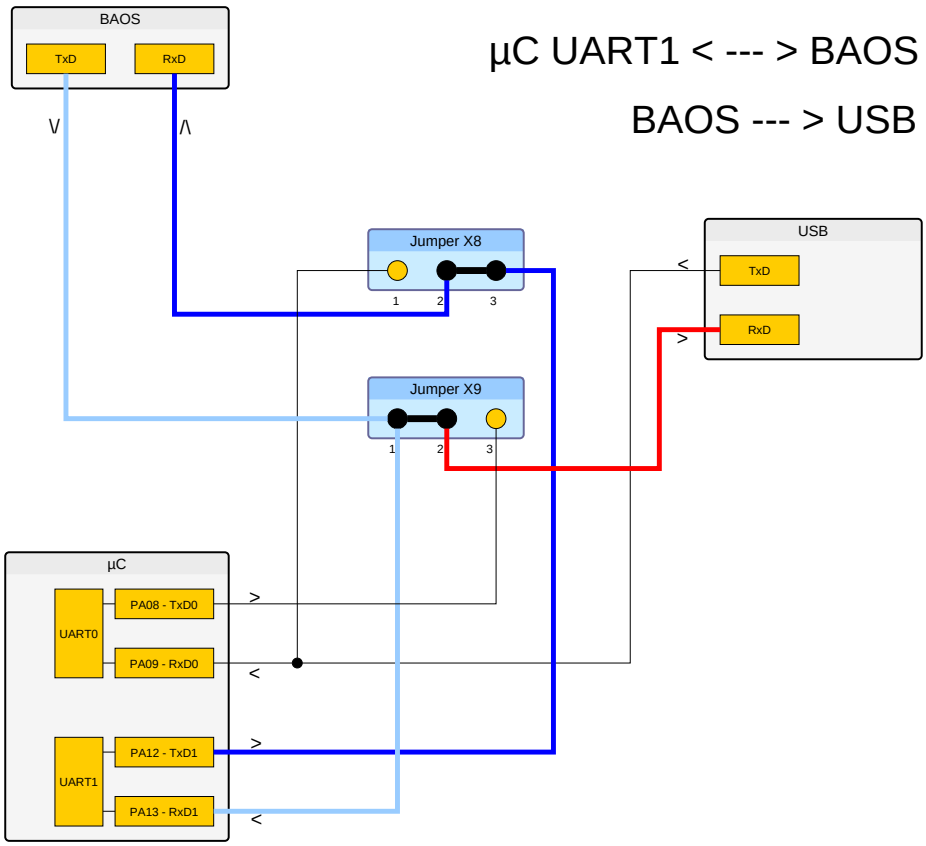
To use Net'n Node,

- select **View/Access Port Configuration**,
- select **Create new...**,
- select **BAOS FT1.2 Serial**,
- select the correct serial device name (**COMx**),
- and use **Test**.
- If the test is successful, a requester will ask for adding the port to the user list. Answer **Yes**.
- **Open** the port,
- choose a **BAOS** telegram (e. g. **GetServerItem**, start index: 9, number of: 1) and **Send** it.
- A response will be displayed in the Telegram View. Double click this telegram to get a more comprehensive interpretation. In case of ServerItem 9, the time since last reset [ms] is shown in a 4 byte value. Example: 0x006BA2C6. This means the module is up and running since 7054022 ms, or 1 hour, 57 minutes, 34 seconds and 22 milliseconds.
- To end this session, **close** the port

Don't forget to reset the jumpers after using Net'n Node.

5.6.2.3 Communication: Monitoring

The last usable combination is to monitor the BAOS communication.



In this case the micro controller of the Development Board (UART1) is connected normally to the BAOS Module (blue and light blue lines). But a PC connected to the USB port can monitor the communication **from** the BAOS Module (red line).

Using a FTDI USB converter cable it is also possible to monitor the communication **to** the BAOS Module. To do this, connect the FTDI cable to the Jumper X8 and use a second USB port of the PC.

To use Net'n Node,

- select **View/Access Port Configuration**,
- select **Create new...**,
- select **BAOS FT1.2 Serial**,
- select the correct serial device name (**COMx**),
- and use **Test**.
- The test will now be unsuccessful, since the module will not answer Net'n Node's requests. The requester will ask to add the port as Spy port. Answer **Yes**.
- Select **Baudrate** 19200,
- select **Decoder** FT1.2 Data,
- select **Packetizer** FT1.2
- and make sure the **Parameters** are 8E1. This will decode the BAOS communication.
- **Open** the port.

This shows the BAOS communication between the module and the application in one way. To see the other way, a second COM port must be opened in the same way.

Don't forget to reset the jumpers after using Net'n Node.

5.7 Schematics of the Development Board

6 KNX BAOS Modules

This chapter describes the BAOS Modules: the connections, dimensions and a firmware overview.

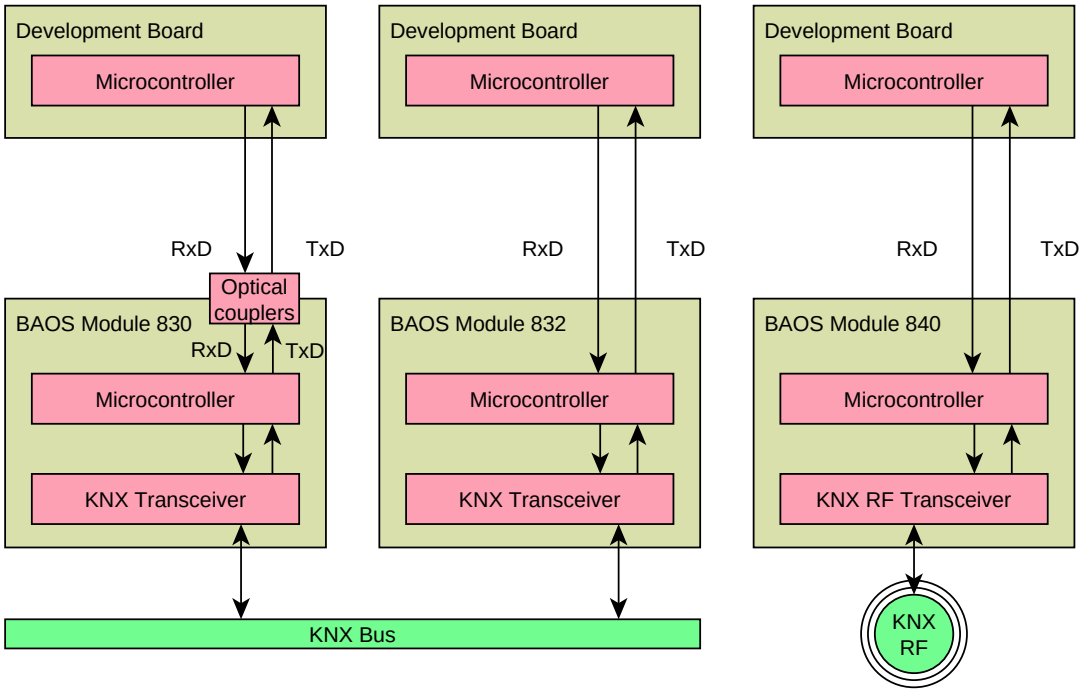
KNX BAOS Modules: from left to right: 830, 832, 838 and 840

6.1 Introduction

The KNX BAOS Module contains a micro-controller and a KNX transceiver to handle the KNX communication. Its interfaces are:

- **A KNX interface** to send and receive telegrams via the KNX bus (see chapter "About KNX" for more info). In case of 830, 832 and 838, this is also the power source for the module.
- **A serial port** (UART) for communication to the device (Development Board, other hardware or, in case of 838, the Raspberry Pi). This serial port uses an FT1.2 protocol for data integrity (see chapter "FT1.2 Protocol") which contains either the BAOS protocol or the cEMI protocol.
    - **The BAOS Protocol** (see chapter "BAOS Protocol") is used for reading and writing data point values, being notified of data point value changes, reading parameters and device settings.
    - **The cEMI Protocol** (see chapter "Common EMI Protocol") offers the possibility to generate own KNX telegrams for the Link Layer.
- Optionally a programming mode (learning key) and LED. The module 838 has them on board.

The modular overview shows two BAOS Modules and their Development Boards, connected by the KNX bus. The serial port (Rxd/TxD) connects the Module and the Development Board. The KNX bus connects both BAOS Modules.



The main difference between the 830 and 832 are the optical couplers.

6.2 Connection Requirements

To use the module the following is required:

- **Connection to the KNX bus**: The KNX BAOS Modules 830, 832 and 838 support a twisted pair (TP) KNX bus interface. This TP interface must be connected to the bus. Its nominal voltage is 29 V. Care must be taken about the polarity. A choke must be used for supplying the KNX bus with power. Without it, the bus will not work. Use an appropriate power supply. The typical KNX plug is black and red.

KNX TP plug

The Development Board supplies the KNX bus to the Modules 830 and 832. The 838 kBerry has its own TP plug.

The KNX BAOS Module 840 supports a radio frequency (RF) KNX bus interface and must be powered via the Development Board.

- **Connection to the application hardware**: The UART connects the application hardware with the KNX BAOS Module. It is a serial port using 3 – 5 V. Two baud rates are available. 19200 is the default. The baud rate can be changed to 115200 via protocol. Data bits are 8, even parity and 1 stop bit: 8e1.

  Warning: Don't connect an RS-232 serial port directly to the pins of the KNX BAOS Module. This will certainly damage the hardware. To connect a PC or anything else which is RS-232 compatible, a level converter is required.

  Note: The KNX BAOS Module's default baud rate is 19200 after every reset. An ETS download also resets the module.

- **Learning key**. A button should be connected to set the KNX BAOS Module into programming mode for downloading an individual address.

- **LED for the learning key**. The programming mode should be indicated by a red LED.

The KNX BAOS Modules 830 and 838 have optical couplers which galvanically isolate the application hardware from the KNX bus. If your application is powered externally, it is required to protect the KNX bus and the application hardware against interferences and different potentials. The application hardware must supply Vcc and GND to the KNX BAOS Module. It is required for the application side of the optical couplers.

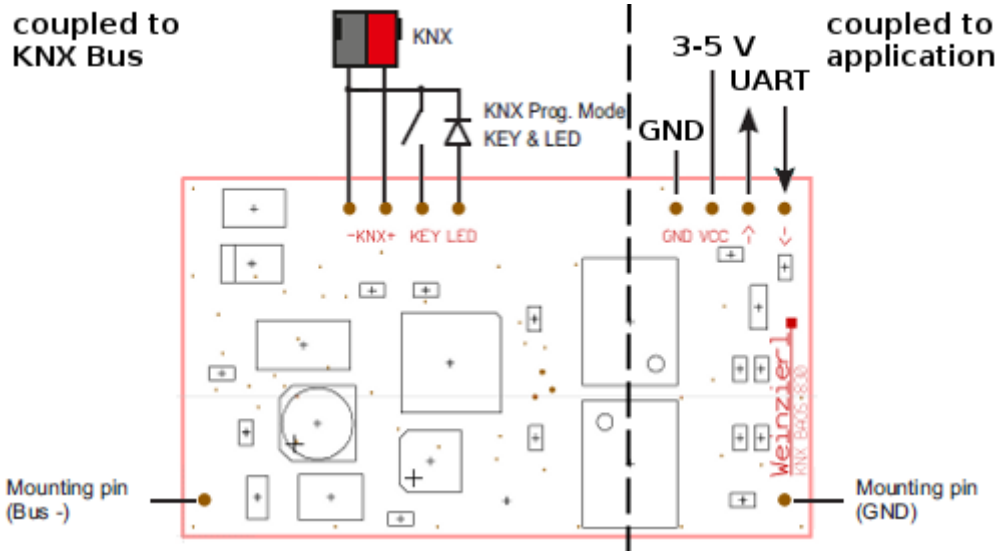
The KNX BAOS Module 832 supplies the application hardware with Vcc, V20 and GND. It will be powered by the KNX bus. This is only recommended for devices which have no other electrical connections (including ground).

Since RF cannot supply power, the KNX BAOS Module RF 840 must be powered by other means.

6.3 Pinning of the KNX BAOS Modules

The pinning figures of the KNX BAOS Modules show the interfaces (KNX bus and UART) plus the programming key & LED.

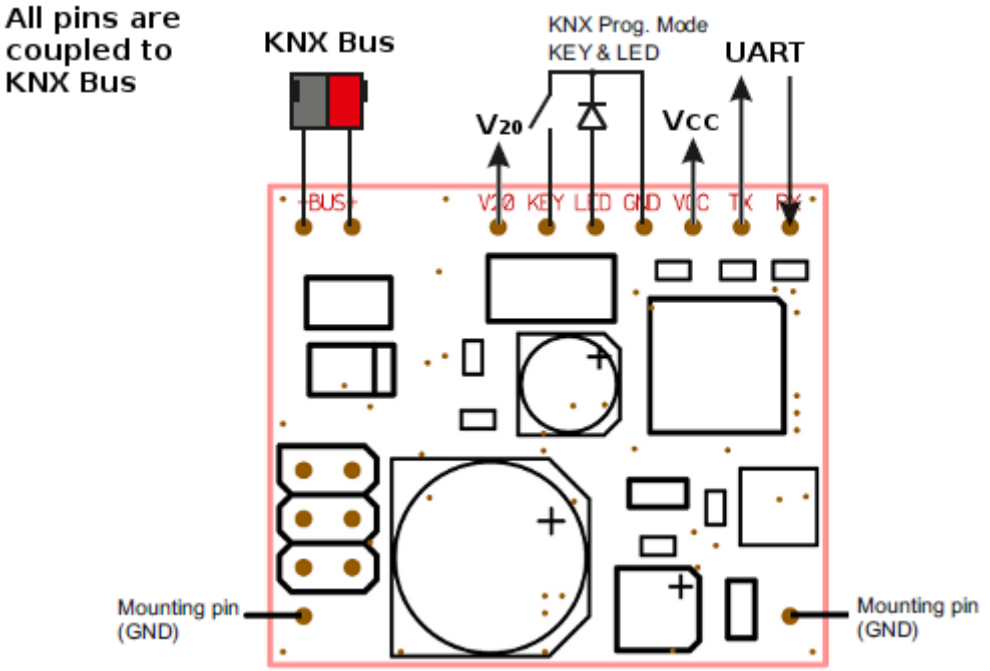
KNX BAOS Module 830:



The LED and programming key are coupled to the KNX bus voltage. The UART connects the Development Board. Vcc and GND must provide power for the isolated part of the module.

Note: Don't mix **KNX bus -** and **GND**.

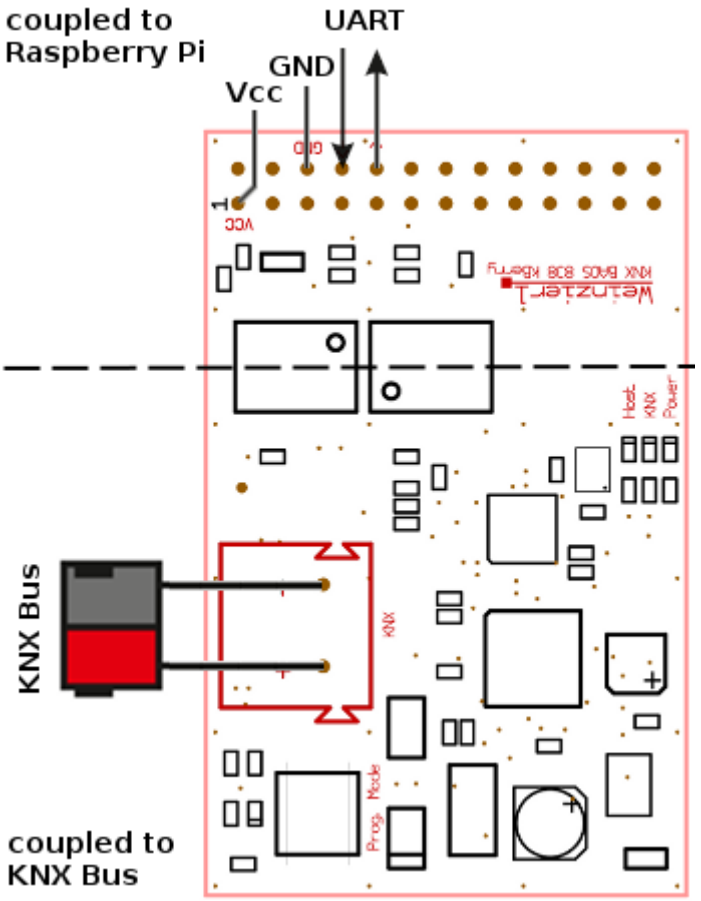
KNX BAOS Module 832:



The KNX BAOS Module 832 has no galvanic isolation, so it can power the application via Vcc and V20.

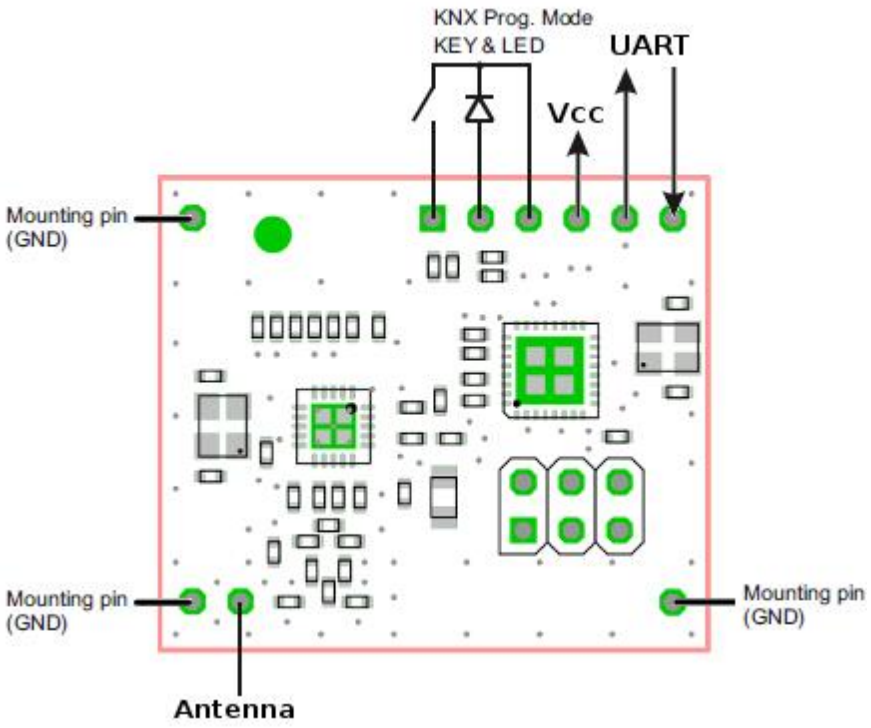
Note: **KNX bus -** and **GND** are at the same potential.

KNX BAOS Module 838:



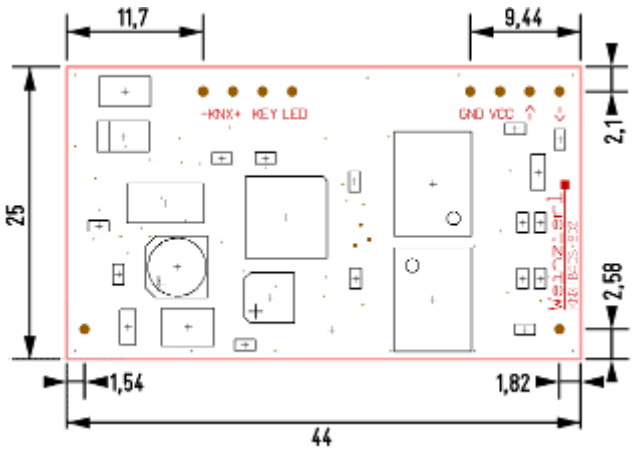
The KNX BAOS Module 838 is technically compatible to the 830, except it is for use with the Raspberry Pi. The connections to the Raspberry Pi are UART (RxD/TxD), Vcc and GND. The programming key & LED are on board.

KNX BAOS Module RF 840:

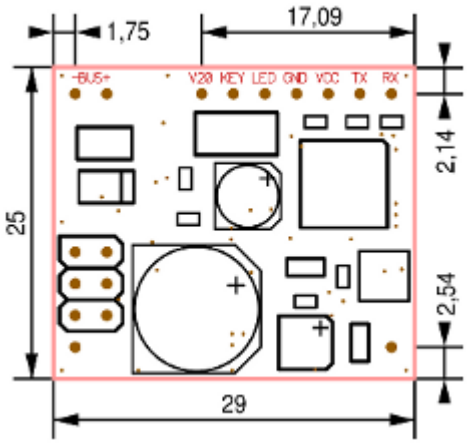


6.4 Dimensions of the KNX BAOS Modules

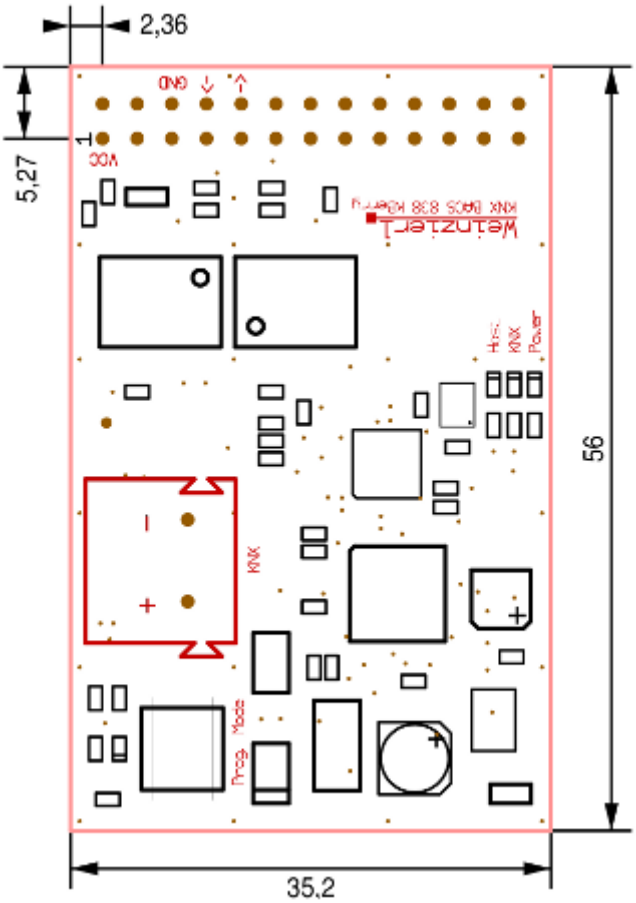
KNX BAOS Module 830:



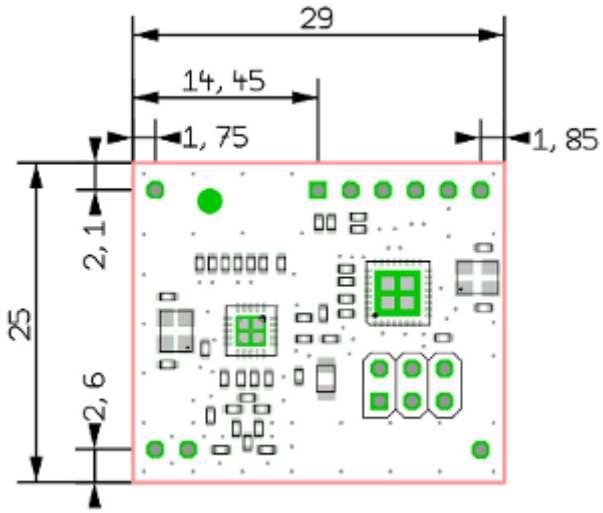
KNX BAOS Module 832:



KNX BAOS Module 838:



KNX BAOS Module RF 840:



6.5 Modular Overview of the Firmware

The firmware has a modular design. The most important modules are shown in the right figure.

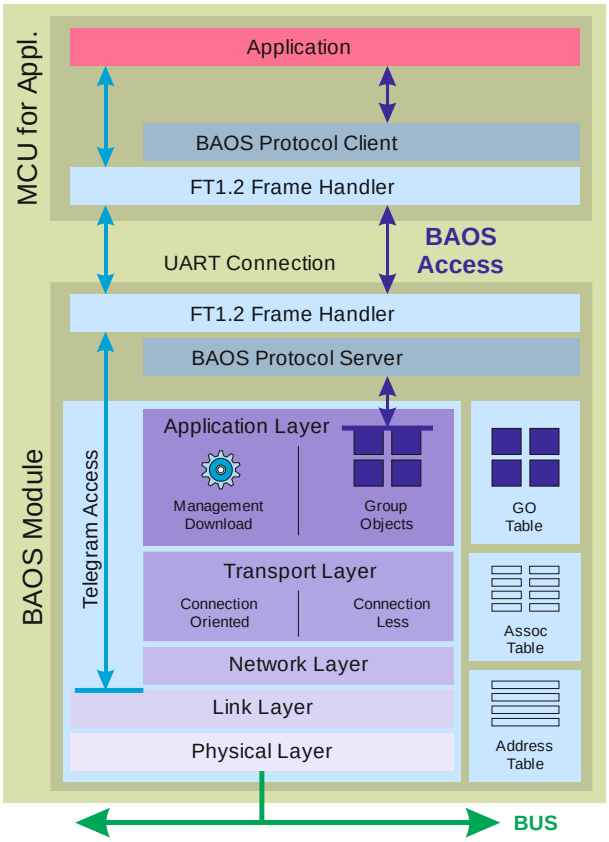
The firmware contains a certified KNX communication stack, which conforms to the OSI model. It also manages the KNX group object table, association table and address table, so the application does not need to care about them.

The BAOS protocol Server handles up to 1000 data points and up to 70 kByte for parameters. The data points can be modified by the application and by KNX telegrams. The application is automatically notified about a change of a data point value.

The FT1.2 Frame Handler, which embeds the BAOS Protocol, ensures data integrity.

Warning: Do not alter or program the micro-controller of the KNX BAOS Module. You might render your device to be permanently unusable.

KNX Device with BAOS Module



6.6 Reset all Configurations of the BAOS Module to Default

ETS can configure the KNX BAOS Module as any KNX device. It is possible to reset the configuration without ETS. This might be necessary if it cannot be reset to default state with ETS. To do this, a **master reset** must be performed.

- The module must be disconnected from power for a few seconds.
- Hold the learning button down and reconnect power.
- Keep the button down for at least 3 seconds, and then release it.

The learning LED flashes for a very short time and about 3 seconds later, the module is up and running again with all configurations reset to default. The individual address is now 15.15.255.

6.7 KNX IP BAOS Devices

KNX IP BAOS devices use an Ethernet/IP interface to the application. If you don't want to use a serial connection to you PC and use an Ethernet/IP interface, the following devices might interest you.

The KNX IP BAOS 771 and 772 offer an Ethernet interface 10Base-T (LAN RJ-45) for connecting a PC or a similar device. The 771 supports 250, and the 772 up to 1000 data points. The BAOS protocol is embedded into IP instead of FT1.2.

More info is available at http://www.weinzierl.de.

The KNX IP BAOS 777 offers the same interface as the 771/772 and up to 2000 data points. It features an internal web server which enables a PC to manage the KNX devices via the 777 with a web browser. This browser can be used to visualize a building structure and its connected sensors and actuators. Furthermore a display in front of the 777 can be used to configure some basic settings.

More info is available at http://www.weinzierl.de.

7 Commissioning with ETS

In this chapter the demo project of the BAOS Module and its configuration is described.

To use and configure the BAOS Module, the ETS (Engineering Tool Software) is used. ETS runs on computers using the *Microsoft Windows* operating system.

This tool requires so-called product databases, which describes KNX devices. The BAOS Module's product database is available in a generic and an example individual version.

The generic database can be used for configuring the BAOS Module. It allows selecting data point types. As long as we are developing our application this is a useful approach. Later, when the application is ready for release, an individual database can be created, which is more convenient for the commissioner. How to do this is described later in the chapter "Individual ETS Entries". For now the generic database is used.

The demo project uses this generic database and has already configured the BAOS Module. We will use this project, now.

7.1 Install ETS

If ETS is not installed, download and install it from the KNX download page at http://www.knx.org. Version 5 suits fine for working with the KNX BAOS Module.

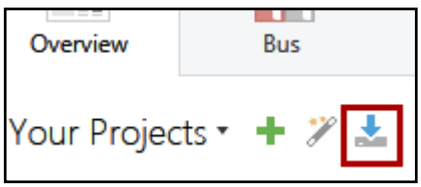
After downloading the executable file, start it (double click) and follow the install instructions.

7.2 Install ETS License

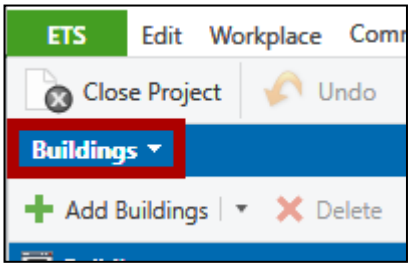
A license might also be necessary to run ETS. The free demo license allows 5 devices per project. If more are needed, a full license is available to purchase at the KNX Online Shop.

7.3 Import a Project

Start ETS and import the test project:



Select the file to import:
Weinzierl_8xx_KNX_BAOS_ETS_Projects_for_Demo/ETS_Project_using_generic_ETS_entry/Project.knxproj.

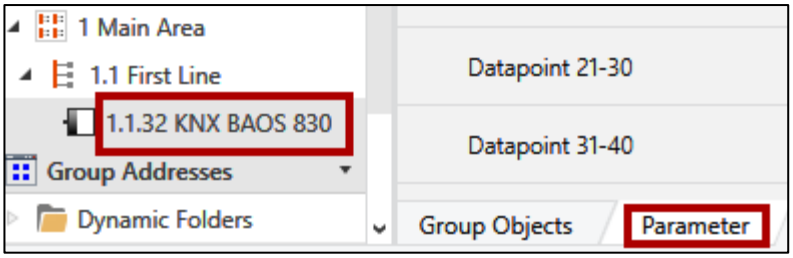


Now open the new Project by double-clicking it. Select **Buildings** and choose **Project Root** to see the whole project. In the Topology, if we expand **1 Main Area** and all its children, we see the device **KNX BAOS 8xx**.

7.4 Topology

The topology is a hierarchical structure which contains the devices of a KNX installation.

7.4.1 Areas



A KNX installation contains some connected devices, each addressed by its own individual address. The individual address is a three number address, like 1.1.1. The first number denotes the area of the network.

Big KNX installations can have up to 15 areas, so the individual address can range from 1.x.x to 15.x.x.

7.4.2 Lines

The second number denotes the line. An area can hold up to 15 lines, so the individual address can range from x.1.x to x.15.x.

7.4.3 Devices

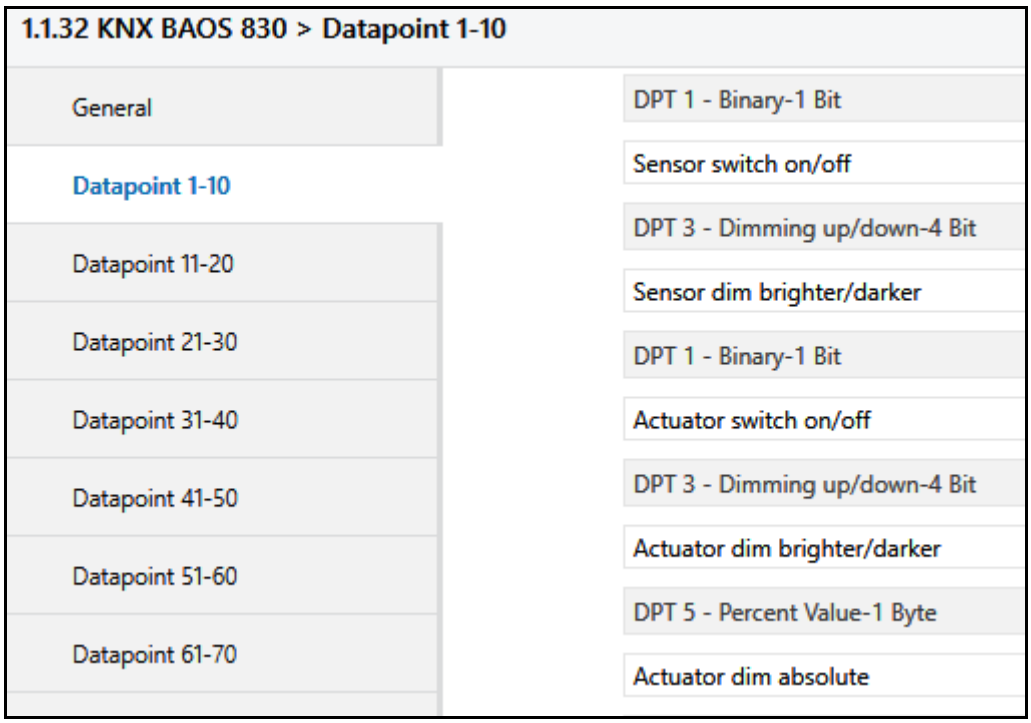
The third number denotes the device. 255 devices can be addressed in the range from x.x.1 to x.x.255.

The device **KNX BAOS 8xx** in this demo project has the individual address 1.1.32.

7.5 Parameters

Now it is time to see the parameters of the BAOS 8xx. Select the device and display its parameter.

According to the demo application of the Development Board, the data points are configured as in the following figure.



The description text of the data point, which is only visible if you enabled a data point, is just a text field which is not downloaded into the KNX device. It stays in the ETS project just for information. It helps the commissioner to give a name to every data point.

The application must know about the usage of the data points, so we use the parameter bytes to inform our application about it. The demo application uses 2 parameter bytes. These parameter bytes can be changed by ETS. The application reads them at start up time and after an ETS download. To access the parameters in the application, the **GetParameterByte.Req** command is used. See **KnxBAOS_Protocol_v2.pdf** for more info.

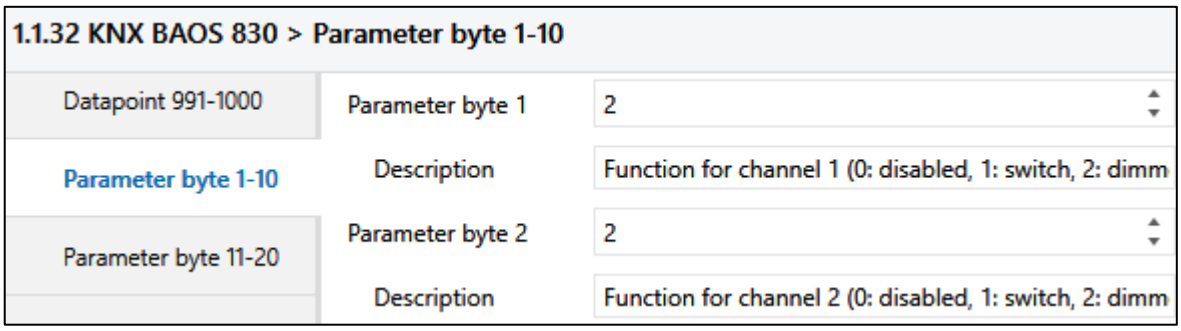
We already read about the interpretation of the parameter bytes in the section "The Demo Application". Here for our case, we use the following values:

Value 2 of parameter #1 means

- Channel 1: data point #1 used as 1 bit switch sensor (S4, S5) and
- Channel 1: data point #2 used as 4 bit dimming sensor (S4, S5).

Value 2 of parameter #2 means

- Channel 2: data point #3 used as 1 bit switch actuator (LED D3),
- Channel 2: data point #4 used as 4 bit relatively dimming actuator (LED D3) and
- Channel 2: data point #5 used as 1 byte absolutely dimming actuator (LED D3).



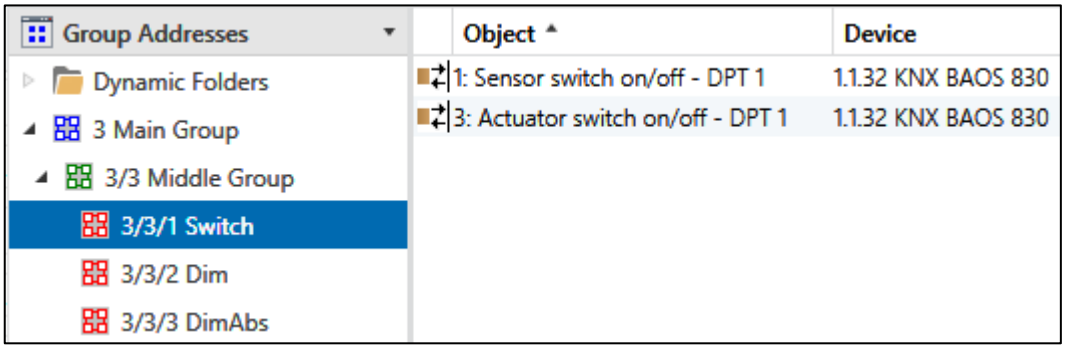
The description of the parameters is also not downloaded into the module. It is only for information to ease the use of the parameter bytes.

7.6 Group Addresses

The data points are linked together by group addresses:

- Data point #1 and data point #3 using the group address 3/3/1.
- Data point #2 and data point #4 using the group address 3/3/2.
- Data point #5 is alone in the group address 3/3/3.

These group addresses contain all data points which have to take action if someone sends a value to this address. There is normally one data point, which sends a value to such a group address and all other data points in this group address listens to this event and act accordingly.



Example: To switch a light on or off, we need one data point from the sensor (the light switch) which sends the 1-bit value. The data point type must be DTP 1 (1 bit) for all connected data points. All other data points in this group address will listen. These are the data points from all the actuators (light bulbs, etc.). The group address 3/3/1 can be a freely chosen address, but it must be unique in the KNX installation.

Group address 3/3/3 (an absolute dimming value) is used in this demo project as test for Net'n Node (see section "Monitoring KNX using Net'n Node"). Some sensors can use such value to tell the actuator to dim a light to a certain value. This value has the type DPT 5 (1 byte) and ranges from 0 to 100. 0 is off and 100 is the brightest light.

7.7 Download

Connect the Development Board to the KNX bus and the PC to a KNX Interface. Press the *Learn button* on the board (red LED must light up). Select the device **KNX BAOS 8xx** and use menu **Download/Full download**.

The device is now configured. Press the buttons S4 and S5. The LED D3 is now switched on and off via group address 3/3/1.

If you press S4 and S5 long, the LED dims lighter or darker (via group address 3/3/2).

The Telegrams can be monitored with Net'n Node, as described in section "Monitoring KNX using Net'n Node".

7.8 Conclusion

The generic database has the advantage, that all data points can be configured as any type. This is the usual way while development. For commissioning it is inconvenient since the installer has to define the correct type for every data point. With an individual database all data point types can be defined like the application expects them and only these data points can be made visible which are really needed.

How to create an individual database we will see in chapter "Individual ETS Entries".

8 Programming the Development Board

This chapter describes how to program the Development Board using an IDE.

The demo application is an example software, which handles some push buttons and some LEDs. Its sources are also provided, which can be used as a starting point for developing own applications. Basic knowledge in programming in C language is assumed.

The source is written in C, suitable for the freely available *WINAVR (GNU) compiler*. You can create own applications based on the Atmel Cortex micro-controller. For information about this controller, see

Atmel ARM-based 32-bit Micro-controller SAM D20

at the Atmel web site.

8.1 Additional Hardware

To program the Development Board the following additional hardware, besides a KNX installation, is needed:

- A JTAG interface (like AVR JTAGICE3) to program the board.
- A PC which has installed Atmel Studio 7 or higher and ETS 5 or higher.

8.2 Installation of IDE and Compiler

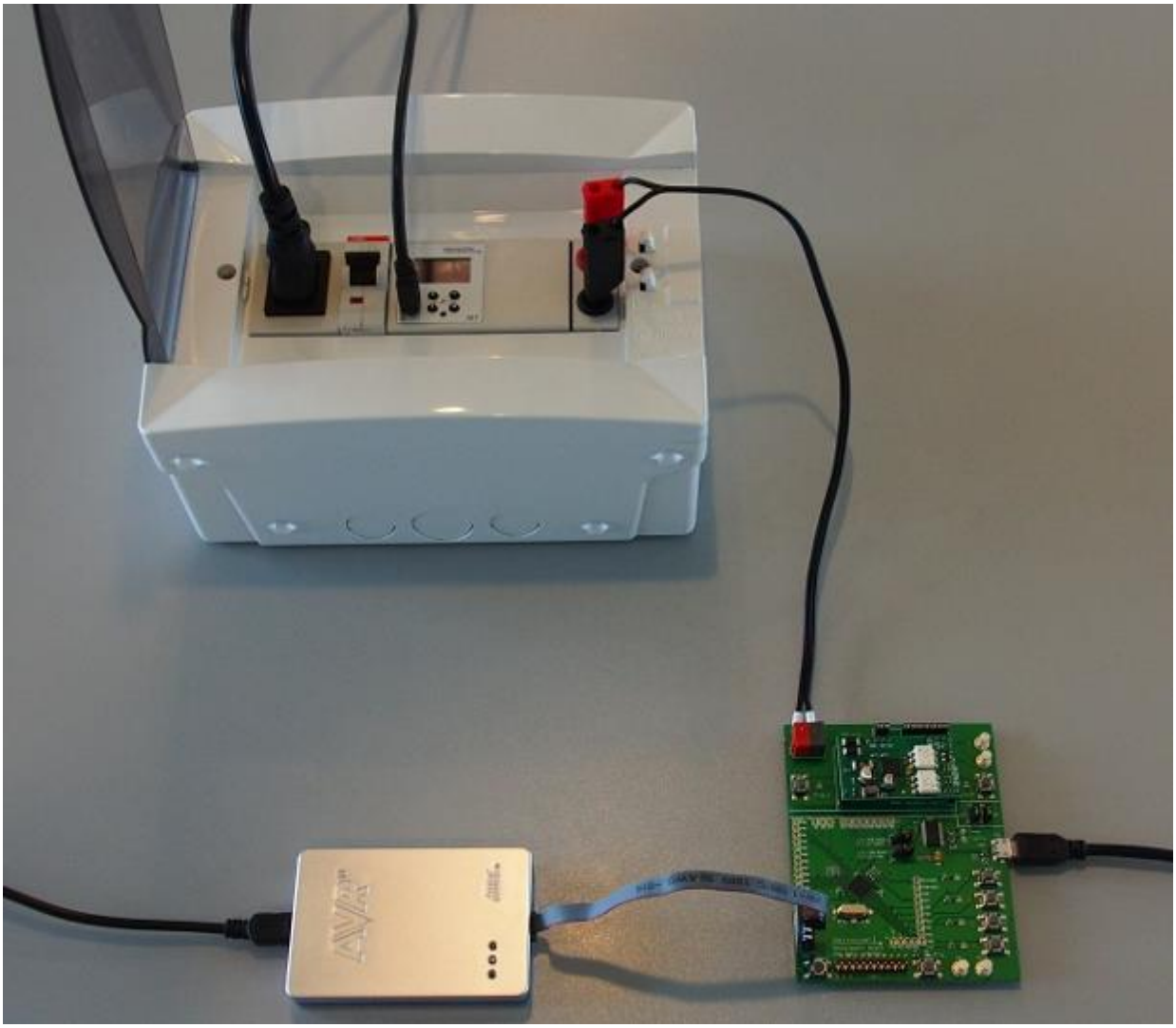
Atmel Studio is the IDE (integrated development environment) based on *Microsoft Visual Studio*. It is available for free at the Atmel web site. It uses the compiler set WINAVR (GNU). If you have already installed Atmel Studio make sure you have at least version 7.

You need a programming tool to write the software to the micro-controller of the Development Board. ATJTAGICE3 can do this and is available at the Atmel web site. Also download Atmel Studio, which is not included in the starter kit.

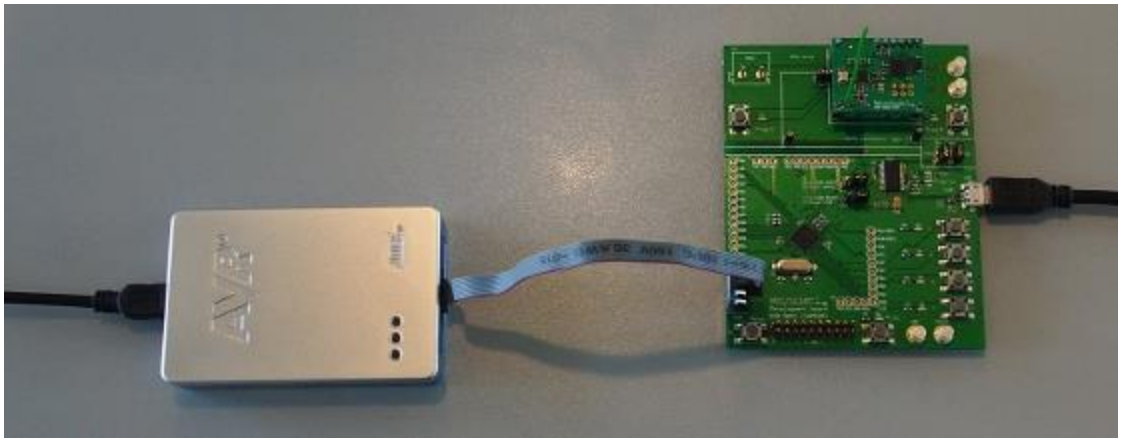
8.3 First Debugging Steps

Connect the SWD connector of the Development Board to the programming tool (ATJTAGICE3). The programming tool is connected to the PC. Furthermore connect the Development Board to the KNX bus. The Development Board gets its power from the USB interface. Connect it to the PC.

The following figure shows the hardware build-up.



In this figure, the KNX bus is provided by the white box. It consists of a power supply including a choke and a *KNX USB Interface*.



The build-up for RF is simpler, but some kind of RF interface is needed for the PC/ETS.

Unpack the archive

Weinzierl_8xx_KNX_BAOS_Demo_Source.zip,

start Atmel Studio and open the project

**Weinzierl_8xx_KNX_BAOS_Demo_Source/
Sources/
Weinzierl_8xx_KNX_BAOS_Demo_Project.atsln**.

Select the project and choose the menu **Project/Properties**. Check whether everything is configured correctly:

Tab	Parameter	Development Board
Device	Device Name	ATSAMD20G18
Tool	Selected debugger/programmer	JTAGICE3 (or similar)
	Interface	SWD
	Programming settings	Erase only program area or Erase entire chip

Now build the demo application with menu **Build/Build Solution** and start it with menu **Debug/Start Debugging and Break**.

A few seconds later the application starts and stops at the entrance of the main() function:

```
int main(void)
{                                   <= EXECUTION STOPS HERE
    App_Init();

    while(TRUE)
    {
        App_Main();
    }

    return 0;
}
```

Now we can go through the program step by step using the keys **F10** and **F11**. To continue the program use **F5**.

8.4 Download a Binary Application

If you want to download a binary application into the micro-controller of the Development Board, do the following:

1. Start Atmel Studio.
2. Choose menu **Extras/Device Programming** which shows a dialog window.

3. Use the correct tool (e. g. JTAGICE3) for the Development Board:
   Device: ATSAMD20G18
   and
   Interface: SWD.
   Push the button **Apply** to establish the connection. Verify it by pushing the button Device signature **Read**. It must show a hexadecimal value, which is the device signature. Example: 0x10001205.
4. Go to tab **Memories**. Load the ELF file. An example is included in the archive

   Weinzierl_8xx_KNX_BAOS_Demo_Source.zip.

   Unpack the archive and use the file

   Weinzierl_8xx_KNX_BAOS_Demo_Source/
   Binary/
   Weinzierl_8xx_KNX_BAOS_Demo.elf

   to program the **Flash** of the Development Board.
5. Make sure **Erase Flash memory before programming** check-mark is set
6. Program the device by pressing the button **Program**.

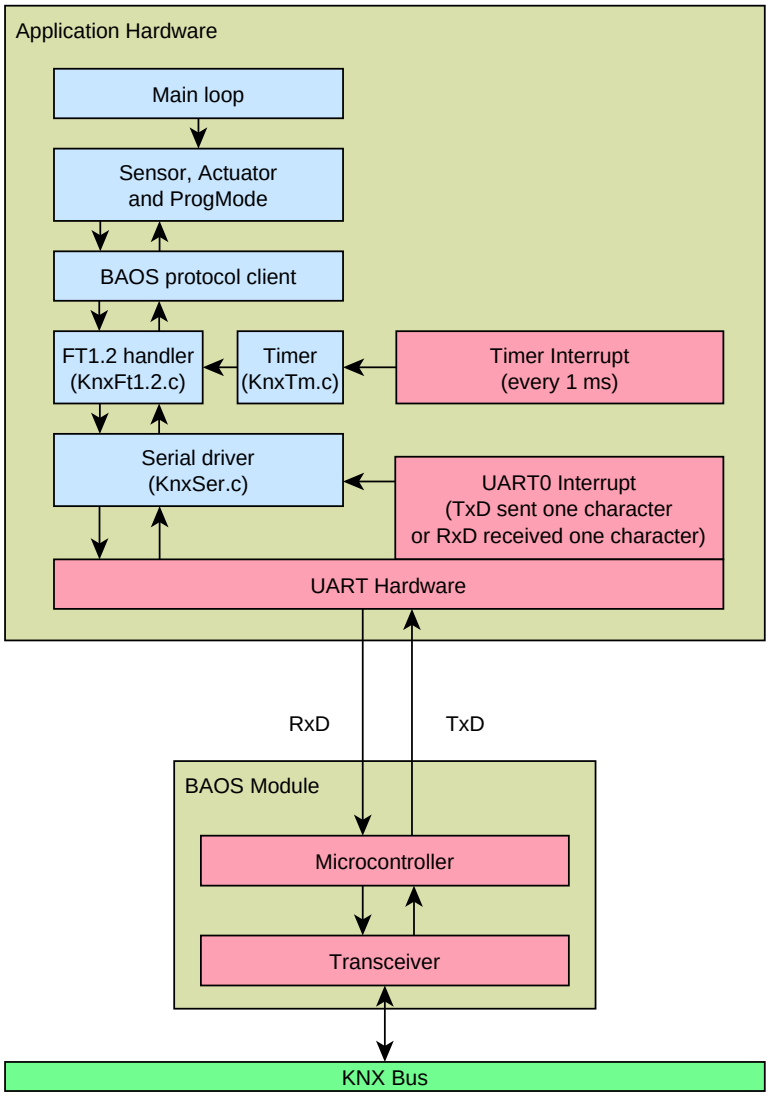
Now the Development Board is programmed with the binary file.

9 The Demo Application

This chapter is about the basics of the demo application and how to use the BAOS frame work.

The application uses a main loop at its top level, which calls the sensor and actuator module cyclically. The sensor part handles the buttons and sends its events to the BAOS protocol client. The actuator part handles the LEDs according to the events received from the BAOS protocol client.

The FT1.2 handler (via timer handler) and the serial driver use 2 interrupts: timer and communication. The communication interrupt occurs for every sent or received character at the UART port. The timer interrupt calls the timer handler every millisecond which manages a global counter and the timeouts for the FT1.2 handler.



9.1 Software Modules

9.1.1 Main Loop Module

The main loop module contains only one source file:

Main.c is the main entry. It initializes all components, like system and application, and enters a never ending loop. This main loop contains the BAOS process and one application, called both cyclically.

```
AppBoard_PortsInit();         // Set board-configurations
KnxBaos_Init();               // Initialize BAOS
App_Init();                   // Set start-configuration for demo-application

while(TRUE)                   // Never ending main loop
{
      KnxBaos_Process();      // Handle KNX BAOS communication
      App_Main();             // Call main loop
}
```

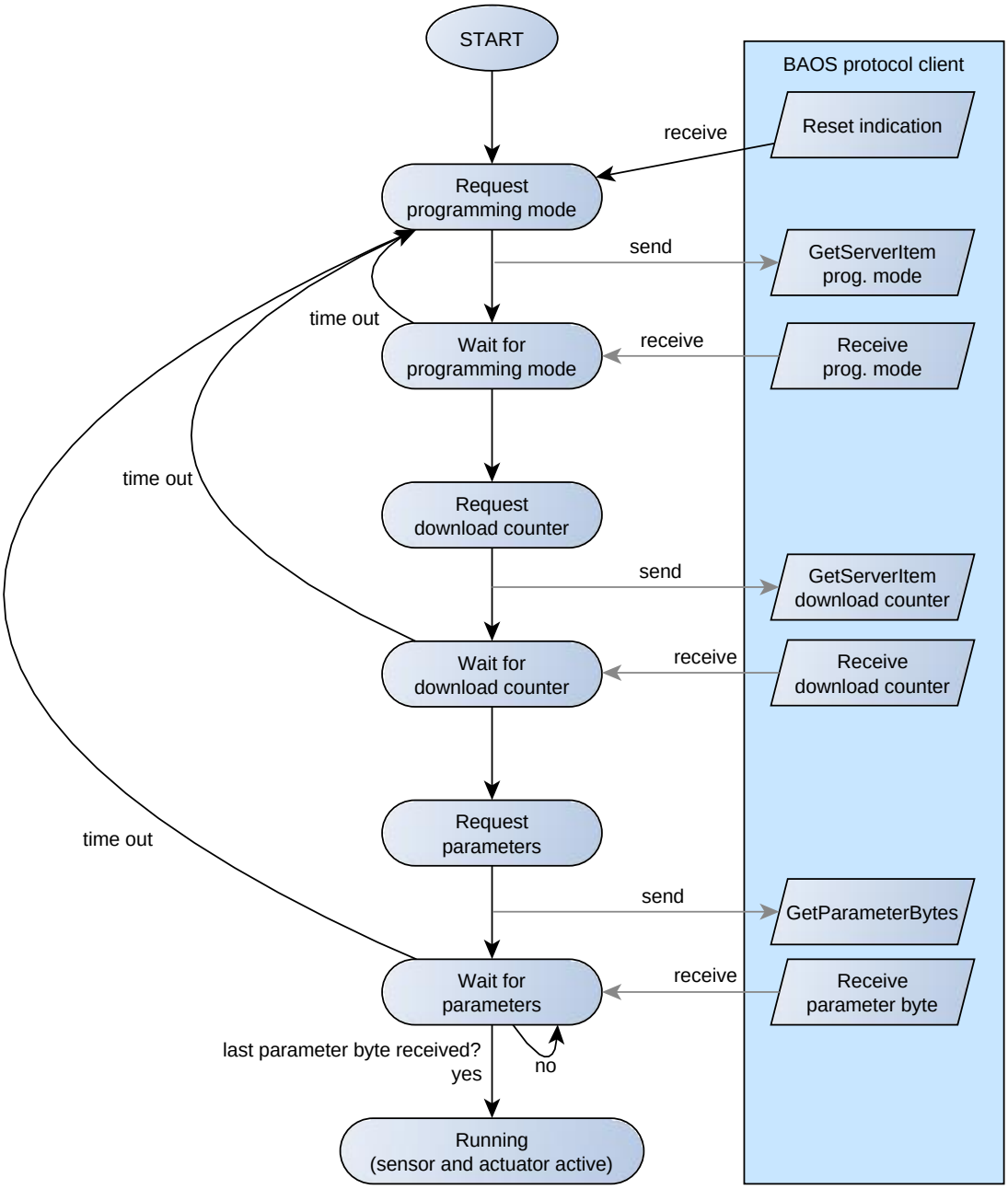
9.1.2 Sensor, Actuator and ProgMode Module

App.c contains the initialization function App_Init(), called from main().

```
AppKey_Init();          // Initialize our two push buttons
AppActuator_Init();     // Initialize actuators
AppSensor_Init();       // Initialize sensors
AppProgMode_Init();     // Initialize programming mode handler
```

In this call, it initializes all components: key handler, actuator, sensor and programming mode parts.

App_Main() is called cyclically from the main loop. It executes a state machine, which does the following:



1. Request the programming mode. The key S3 and the LED D2 on the Development Board are for an application driven programming mode. The state machine gets the current programming mode from the BAOS module and updates the LED D2.
2. Get download counter. The download counter is a ServerItem of the BAOS module which counts the ETS downloads. This value is stored in an internal variable, which can be read by calling the function App_GetDownloadCounter(). In this demo application it is just an example and is not used further.
3. Retrieve some parameter bytes from the BAOS Module. All parameter bytes, the application needs to know for running the sensor and actuator, are requested.

At the end of this initialization, the key handler, actuator part, sensor part and programming mode handler are executed in every loop. This is the application's main work.

In case of a reset indication (e. g. BAOS Module reboots after an ETS download), the application will be notified and the state machine starts again. See **App.c**.

```
/// Handle reset indication.
///
/// BAOS has been reset (could be due to a change of the parameters via ETS).

void App_HandleResetIndication(void)
{
    m_eState = REQUEST_PROG_MODE;    // Initialize state machine
}
```

ETS sets the parameter bytes. In the demo application, we use only two parameter bytes. See **App.h**.

```
/// Parameter byte assignment
///
/// Parameter bytes used by this application:
///
/// PB#1: Configuration for DP#1 and #2
/// PB#2: Configuration for DP#3, #4 and #5

enum PbUsage_tag
{
    PB_UNUSED      =  0,             // PB#0 is never used
    PB_FIRST       =  1,             // First used parameter byte
    PB_SWITCH_TYPE =  1,             // PB#1
    PB_LIGHT_TYPE  =  2,             // PB#2
    PB_MAX         =  2              // All remaining param. bytes not used
};
```

Only PB_SWITCH_TYPE and PB_LIGHT_TYPE are used as parameter bytes. These parameter bytes are used in the actuator and sensor part.

AppActuator.c contains the actuator part of the application. It handles the LEDs according to the BAOS indications. The function AppActuator_HandleDatapointValueInd() gets called for every data point change from the KNX bus.

```
switch(nDpId)
{
      case DP_LED0_SWITCH_I:             // Switch object selected

            if((m_nLightType == LT_SWITCH)
            || (m_nLightType == LT_DIMMER))
            {
                  if(nDpLength == 1)      // We expect 1 byte data length
                  {
                        if(*pData == SWITCH_ON)            // Switch on
                        {
                              AppDim_Switch(TRUE);
                        }
                        else if(*pData == SWITCH_OFF)      // Switch off
                        {
                              AppDim_Switch(FALSE);
                        }
                  }
            }
```

```
            break;

      case DP_LED0_DIM_RELATIVE_I:        // Relative dimming object selected

            if(m_nLightType == LT_DIMMER) // We expect to be a dimmer
            {
                  if(nDpLength == 1)      // We expect 1 byte data length
                  {
                        AppDim_DimRelative(*pData);
                  }
            }

            break;

      case DP_LED0_DIM_ABSOLUTE_I:        // Absolute dimming object selected

            if(m_nLightType == LT_DIMMER) // We expect to be a dimmer
            {
                  if(nDpLength == 1)      // We expect 1 byte data length
                  {
                        AppDim_DimAbsolute(((uint16_t)*pData)*255/100);
                  }
            }

            break;

      default:                            // Ignore all other data points
            break;
}
```

nDpId is the data point number which has been changed. If this number is one of our data points we are handling (DP_LED0_xxx), the corresponding code case is executed.

Next we check whether and how the data point is used by checking the corresponding parameter byte. This parameter byte is stored in m_nLightType. If its value is LT_SWITCH only the data point DP_LED0_SWITCH_I is active. In case of LT_DIMMER all three data points are active.

nDpLength contains the size of the new data point value. It must match the size we expect for our application. If the ETS is misconfigured to use a bigger size than 1 byte, we ignore the indication.

pData id a pointer to a byte array which contains the new data point value. In our case, we accept only 1 byte.

If all these conditions are met, we act on the LED and call the function AppDim_xxx() to switch in on, off or dim. The absolute dimming value from the KNX bus can range from 0 to 100 %. We map this to the range 0 to 255, since the function AppDim_DimAbsolute() accepts this range.

AppSensor.c contains the sensor part of the application. It gets key events and sends BAOS commands.

```
nEvent = AppKey_GetKeyEvent(1);                 // Get event for key S4
```

```
switch(nEvent)
{
      case KEY_EV_LONG:                         // Key long pressed
            if(m_nSwitchType == ST_DIMMER)      // Data point used as dimmer?
            {
                  nValue = DIM_REL_DIRECTION_UP | DIM_REL_100;

                  KnxBaos_SendValue(
                        DP_SWITCH_DIM_OR_MOVE_O, DP_CMD_SET_SEND_VAL, 1,
                         &nValue);

                  // DatapointID: 2, Command: send value, Length: 4 bit
            }
            break;

      case KEY_EV_SHORT:                        // Key short pressed
            if((m_nSwitchType == ST_SWITCH)     // Data point used as switch?
            || (m_nSwitchType == ST_DIMMER))    // or as dimmer?
            {
                  nValue = SWITCH_ON;           // Value: 1 -> LED-ON telegram

                  KnxBaos_SendValue(
                        DP_SWITCH_OR_STEP_O, DP_CMD_SET_SEND_VAL, 1, &nValue);

                  // DatapointID: 1, Command: send value
            }
            break;

      case KEY_EV_RELEASE:                      // Key released after long pr.

            if(m_nSwitchType == ST_DIMMER)      // Data point used as dimmer?
            {
                  nValue = DIM_REL_DIRECTION_UP | DIM_REL_STOP;
                  // DPT_Control_Dimming: stop increasing LED light

                  KnxBaos_SendValue(
                        DP_SWITCH_DIM_OR_MOVE_O, DP_CMD_SET_SEND_VAL, 1,
                         &nValue);

                  // DatapointID: 2, Command: send value
            }
            break;

      case KEY_EV_NONE:                         // State of key not changed
            break;
}
```

AppKey_GetKeyEvent() returns the current key event. The key events can be a long press, a short press, a release and no key pressed. The long press event is followed by the release event after the user releases the switch. A short press is not followed by a release.

Every case contains one statement which check whether a certain data point is in use or not (m_nSwitchType). This is controlled by the parameter bytes, which are retrieved at start of the application and at a reset indication from the KNX BAOS Module.

If the parameter byte for our switch is ST_SWITCH we change the value of data point #1 only by a short key press (simple switch). If the parameter byte is ST_DIMMER, we change data point #1 at a short press or data point #2 if the switch is long pressed (dimming).

nValue contains the new value for the data point. The function KnxBaos_SendValue() needs a byte array for this value because these values can have up to 14 bytes. In our case we use only 1 byte, so we just send the address of nValue to the function.

The key number for the key events is mapped in **AppKey.h**:

```
#define APP_KEY_COUNT       3             // Count of application keys
#define IS_KEY_PRESSED_0    GET_KEY_S3    // Port pin for channel 0
#define IS_KEY_PRESSED_1    GET_KEY_S4    // Port pin for channel 1
#define IS_KEY_PRESSED_2    GET_KEY_S5    // Port pin for channel 2
#define IS_KEY_PRESSED_3    GET_KEY_S6    // Port pin for channel 3
#define IS_KEY_PRESSED_4    GET_KEY_S7    // Port pin for channel 4
```

In this case the key S3 is mapped to key #0, S4 to key #1, and so on. The mapping can be changed along with the number of used keys.

The time (milliseconds) for a long press is also defined in this file:

```
#define KEY_TIME_LONG 300
```

AppProgMode.c contains the programming mode handler of the application. It handles the key S3 and LED D2.

AppDim.c handles the LED. It performs the dimming of the LED as well as the switching on and off. **AppDim.h** defines the maximum value of the LED for 100% brightness and its timing for every dimming step (4 milliseconds delay):

```
#define DIM_MAX_VALUE 0xFF              // Max. Brightness
#define DIM_MIN_VALUE 0x00              // Min. Brightness

#define DIM_RAMP_TIME 0x04              // Delay in ms for every dimming step
```

AppLedPwm.c is also part of the LED handling. It maintains the software PWM which controls the LED.

AppKey.c is the driver for handling the push buttons on the Development Board. It also handles their de-bouncing, long/short presses and releases.

AppBoard.c initializes the ports of our micro-controller at the Development Board. See **AppBoard.h** for which port is connected to the components on the board.

9.1.3 BAOS Protocol Client Module

KnxBaos.c and **KnxBaosHandler.c** contain the implementation of the object server protocol, which is the core protocol of the BAOS. They mainly contain routines for sending and receiving telegrams.

9.1.4 FT1.2 Handler Module

KnxFt12.c handles the FT1.2 protocol which is used by the communication with the BAOS Module.

KnxBuf.c handles receive buffer for FT1.2.

9.1.5 Serial Driver Module

KnxSer.c is the low level serial driver used by the communication with the KNX BAOS Module.

9.1.6 Timer module

KnxTm.c is the system timer and contains some convenience functions for timer usage. The following function returns the current up time in milliseconds. It starts counting after a reset of the application.

```
uint32_t KnxTm_GetTimeMs(void);          /* 0x00000000 - 0xffffffff */
```

The function KnxTm_GetTimeMs() retrieves the current timer value.

Warning: The timer does not start immediately after a reset or power up.

Using KnxTm_SleepMs() is dangerous if it is used for more than 10 ms, since this is a busy wait. The BAOS communication can fail in this case. It is better to perform delays like this in the main loop:

```
{
    static uint32_t nTimeStamp;
    static enum eState_t nState = IDLE;

    switch(nState)
    {
        case IDLE:

            /* Store current time stamp for wainting. */

            nTimeStamp = KnxTm_GetTimeMs();
            nState = WAITING;
            break;

        case WAITING:

            /* Wait for the 200 ms. */
            /* i. e. Return control to the main loop. */

            if(KnxTm_GetDelayMs(nTimeStamp) >= 200)
            {
                nState = RUNNING;
            }
            break;

        case RUNNING:

            /* Time has elapsed. */
            /* Do something here and start waiting again. */

            nState = IDLE;
            break;
    }
```

```
}
```

9.1.7 Header Files

StdDef.h defines some macros which are helpful. It's a good idea if you familiarize yourself with the content of this file since its macros are used in nearly every source file. Some common typedefs are defined there, in case a compiler does not come up with these types:

```
// General type defines. These are normally defined in "stdint.h", which
// comes from the compiler. But if some types are missing, we can define
// them here.

#ifndef ___bool_t_defined
typedef unsigned char       bool_t;      //  1 Bit variable
#define __bool_t_defined 1
#endif

#ifndef ___int8_t_defined
typedef char                int8_t;      //  8 Bit variable
typedef unsigned char       uint8_t;     //  8 Bit variable (unsigned)
#define __int8_t_defined 1
#endif

#ifndef ___int16_t_defined
typedef int                 int16_t;     // 16 Bit variable
typedef unsigned int        uint16_t;    // 16 Bit variable (unsigned)
#define __int16_t_defined 1
#endif

#ifndef ___int32_t_defined
typedef long                int32_t;     // 32 Bit variable
typedef unsigned long       uint32_t;    // 32 Bit variable (unsigned)
#define __int32_t_defined 1
#endif
```

9.2 Creating Own Applications

In order to make your own applications, the files **App.c, AppSensor.c** and **AppActuator.c** are the central place. The function **App_Init()** does the initialization. The function **App_Main()** is cyclically called by **main()**. It must not happen that the function **App_Main()** is blocked because the interrupts would be still enabled, but the processing of received data and the transmission of data would be stopped.

Use the function **AppKey_GetKeyEvent()** to query the buttons and use defines like **SET_LED_D3** from **AppBoard.h** to control the LED.

9.2.1 Use Cases

The most important cases are to set and get the data point values and to get parameter bytes. This can be done by using the BAOS protocol.

9.2.1.1 Set Data Point Value

To set a new value to a data point and send it to the KNX bus, do the following:

```
uint8_t nValue = 0;
KnxBaos_SendValue(2, DP_CMD_SET_SEND_VAL, 1, &nValue);
```

This sets the new value 0 of data point #2 to have one byte size (1). This value change is sent to the KNX bus. Take care to correctly configure this data point to one byte size by ETS.

Keep always in mind the function KnxBaos_SendValue() accepts a byte array as value. Since we have here only one byte, we can use &nValue. But sending a value of more bytes might require a conversion of endianess, since network communication is always defined in big endian, whereas some micro-controllers use little endian. In memory a 4 byte value 0x11223344 is stored in big endian like this:

```
11 22 33 44
```

The same value is stored in little endian like this:

```
44 33 22 11
```

So we must be careful for values longer than 1 byte.

9.2.1.2 Get Data Point Value

If a data point value gets changed by the KNX bus the routine **App_HandleDatapointValueInd()** is called:

```
/// Handle the DatapointValue.Ind data.
///
/// A KNX telegram can hold more than one data. This functions gets called
/// for every single data in a telegram array.
///
/// @param[in] nDpId Current data point ID from telegram
/// @param[in] nDpState Current data point state from telegram
/// @param[in] nDpLength Current data point length from telegram
/// @param[in] pData Pointer to byte data from telegram
///
void AppActuator_HandleDatapointValueInd(
    uint16_t nDpId, uint8_t nDpState,
    uint8_t nDpLength, uint8_t* pData)
{
    switch(nDpId)
    {
        case 1:
            [...]
            break;
        case 2:
            [...]
            break;
        case 3:
            [...]
    }
```

```
}
```

Note: The first data point is always 1 (not 0).

It is always a good idea to check the size of the data point value stored in the byte array pData. Unless we do not check nDpLength, the access to pData can cross the array boundary.

If we do not want to wait for a notification whether a data point has changed (indication), we can explicitly request the current data point value, like this:

```
KnxBaos_GetDpValue(nDpId, nN);
```

This requests the current values of data point nDpId and the following nN data points. After sending this request we will be informed by **App_HandleGetDatapointValueRes()**:

```
/// Handle the GetDatapointValue.Res data.
///
/// A KNX telegram can hold more than one data. This functions gets called
/// for every single data in a telegram array.
///
/// @param[in] nDpId Current data point ID from telegram
/// @param[in] nDpState Current data point state from telegram
/// @param[in] nDpLength Current data point length from telegram
/// @param[in] pData Pointer to byte data from telegram
///
void App_HandleGetDatapointValueRes(
    uint16_t nDpId, uint8_t nDpState,
    uint8_t nDpLength, uint8_t* pData)
{
}
```

For every data point we requested, this function gets called. pData contains the value and nDpLength its size.

The parameters of this routine are the same as in **App_HandleDatapointValueInd()**, which is called automatically if a KNX telegram changes a data point value.

9.2.1.3 Get Parameter Byte

ETS can set parameter bytes while download. These parameters can be used to configure device behaviour. E. g. set a time out, a lighting value or a temperature threshold. These parameters are organized byte wise. To access some bytes do the following:

```
KnxBaos_GetParameterByte(nStartByte, nNumberOfBytes);
```

This requests **nNumberOfBytes** starting at byte **nStartByte**. There are 250 bytes available in the generic ETS database, starting at byte #1. After sending this request we will be informed by **KnxBaos_OnGetParameterByteRes()**:

```
/// Handle the GetParameterByte.Res data.
///
/// A KNX telegram can hold more than one data. This functions gets called
```

```
/// for every single data in a telegram array.
///
/// @param[in] nIndex Current byte number (channel)
/// @param[in] nByte Current byte value from telegram
///
void App_HandleGetParameterByteRes(
    uint16_t nIndex, uint8_t nByte)
{
}
```

This routine is called for every byte we requested. The value is delivered in **nByte**. **nIndex** is the current byte index (number) of the request.

10 Programming the Raspberry Pi Board

To use the KNX BAOS 838 kBerry, the Raspberry Pi board must be programmed to communicate to the KNX bus via the BAOS protocol. The software is available at GitHub. Basic knowledge in programming in C++ language and Linux is assumed.

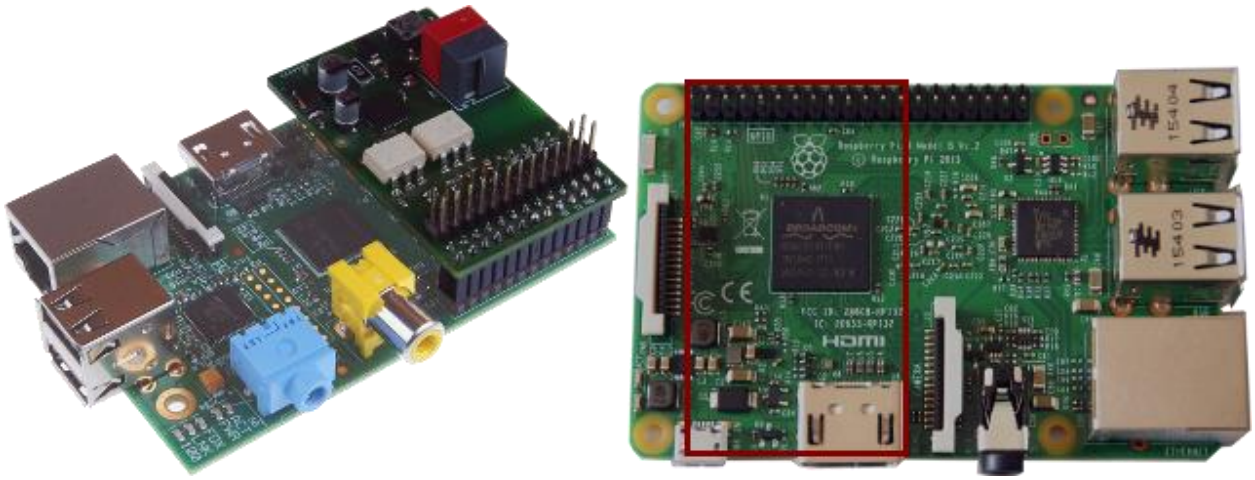
10.1 Download the Operating System

First we have to prepare the operating system of the Raspberry Pi. Download the latest Raspbian Lite at https://www.raspberrypi.org and store it to the SD-card. See README.md for more information about it. You can also use the standard version of Raspbian, but "Lite" is sufficient.

After storing the image to the SD-card, mount it to the Raspberry Pi.

10.2 Connect the Pi and kBerry

Connect the KNX BAOS 838 kBerry Module to the Raspberry Pi as shown in the following pictures. Models with a 40 pin connector have to connect the module as shown on the right picture. For more information about the pinning of the module, see section “Pinning of the KNX BAOS Modules”.



Connect the KNX BAOS 838 kBerry Module to the **KNX Bus** and the Raspberry Pi to an **Ethernet network**. Finally power the Raspberry Pi with at the micro **USB connector**. It should boot now and in a few minutes you can connect it via **ssh pi@raspberrypi**. The password is **raspberry**. If your network does not support network names, you have to use the IP address.

See Raspberry Pi web page for more information about commissioning it.

10.3 Prepare the Operating System

10.3.1 Optionally re-size the File System

If you have successfully connected the Raspberry Pi, re-size the file system, since the image is certainly smaller than the real SD-card. This step is optional, but the free space in the file system will be very small after installing and compiling the kBerry software.

Start the configuration program.

```
sudo raspi-config
```

Select the menu entry **Expand Filesystem** and choose **<Select>** by hitting the *tabulator* key and then *Enter*. A few moments later the message "Root partition has been resized." will appear.

10.3.2 Release the Serial Console

Now we have to release the serial console to use it for communication to the KNX BAOS Module 838. Remove the serial device **ttyAMA0** or **serial0** from the boot command of the kernel and reboot the system. Since we are still running *raspi-config*, we can now select **Advanced Options** and there **Serial** to disable the login shell at the serial port.

Exit the configuration program and do not reboot the system, yet. Change *enable_uart=0* in /boot/config.txt and reboot the system:

```
sudo sed –ie "s/enable_uart=0/enable_uart=1/g" /boot/config.txt
sudo systemctl reboot
```

Warning: The Raspberry Pi 3 does not use the correct baud rate in Raspian out of the box, so kBerry will not work. To fix this, the overlay **pi3-miniuart-bt-overlay** must be activated.

Only Raspberry Pi 3:

```
sudo sh -c "echo dtoverlay=pi3-miniuart-bt-overlay >>/boot/config.txt"
```

All Raspberry Pi (including 3):

```
sudo sed -ie "s/console=[a-Z]*0,[0-9]* //g" /boot/cmdline.txt
sudo systemctl reboot
```

10.3.3 Install Software

A few minutes later, reconnect to the Raspberry Pi and install the software we need. For this, the Pi must have internet access.

```
sudo apt-get update
sudo apt-get install git cmake libboost-dev screen
```

screen is not really necessary, but it makes it more convenient to work on the Pi.

```
screen
```

Now we are in a screen environment and can switch to more console as we like: Enter a new console with **CTRL-A c**, Switch to the next console with **CTRL-A n**, to the previous with **CTRL-A p**, and so on. See **man screen** for more info.

10.4 Install BAOS Software

Download the BAOS software from GitHub and compile it (remove the multiple thread build "-j5" from the make command, or the Pi will certainly run out of memory).

```
git clone https://github.com/weinzierl-engineering/baos
cd baos
sed -ie "s/-j[0-9]*//g" build_unix.sh
sh build_unix.sh
```

This might take quite a lot of time (about 3 hours).

10.5 Use BAOS Software

10.5.1 Read a Server Item

The samples include in the BAOS SDK include one for the serial port. This can be used for communicating to the kBerry Module. Edit file **samples/c++/BaosSerial.cpp**. Look for

```
connector->open("COM8");
```

and change **COM8** to the correct serial port: **/dev/ttyAMA0**. Compile the software again and execute it.

```
cd build_unix
make && make install
bin/sample_BaosSerial
```

It prints the serial number of the KNX BAOS 838 kBerry Module.

```
13:52:04:651 [] Console Logger Started
13:52:04:701 [BaosSerial] Serial Number: 00 C5 00 00 00 00
13:52:04:758 [BaosSerial] 0 items found
```

This sample application reads the data point configurations, also. To do, we need to configure the kBerry with ETS.

10.5.2 First Commissioning with ETS

Configure ETS to use the KNX USB Interface in the **Bus** folder. Look into the list of **Discovered Interfaces** and select the one, which is connected to the KNX bus. **Test** and **Select** the interface. Go back to the **Overview** folder.

For demonstration a simple project for 830 is available in the archive

Weinzierl_83x_KNX_BAOS_ETS_Projects_for_Demo.zip.

Unpack it and import the file

Weinzierl_83x_KNX_BAOS_ETS_Projects_for_Demo/
ETS_Project_using_generic_ETS_entry/
Project.knxproj

in ETS.

The project configures five data points:

1. Sensor switch on/off (1 bit)
2. Sensor dimming up/down (4 bits)
3. Actuator switch on/off (1 bit)
4. Actuator dimming up/down (4 bits)
5. Actuator dimming absolute (1 byte)

Open the project, select **Project Root** as view, select the device **KNX BAOS 8xx** with right mouse button and **Download/Full download**.

Execute the software again.

```
bin/sample_BaosSerial
```

It prints the serial number of the KNX BAOS 838 kBerry Module and the five data point configurations.

```
10:35:16:760 [] Console Logger Started
10:35:16:831 [BaosSerial] Serial Number: 00 C5 00 00 00 00
10:35:17:510 [BaosSerial] 5 items found
10:35:17:513 [BaosSerial] Id: 1, Datapoint type 1, Size: 1 Bits
10:35:17:516 [BaosSerial] Id: 2, Datapoint type 3, Size: 4 Bits
10:35:17:519 [BaosSerial] Id: 3, Datapoint type 1, Size: 1 Bits
10:35:17:522 [BaosSerial] Id: 4, Datapoint type 3, Size: 4 Bits
10:35:17:524 [BaosSerial] Id: 5, Datapoint type 5, Size: 1 Bytes
```

10.5.3 Listening to Data Points

The sample BaosEventListener listens to events. So if any configured data point is changed from the KNX bus, this application recognises it.

Edit file **samples/c++/BaosEventListener.cpp**. Look for

```
std::string name = "Baos-Sample";
ScopedBaosConnection connection(name, true);
```

and replace these lines with

```
std::string name = "/dev/ttyAMA0";
```

```
ScopedSerialBaosConnection connection(name, true);
```

Compile the software again and execute it.

```
cd build_unix
make && make install
bin/sample_BaosEventListener
```

The output looks like this:

```
13:34:19:822 [] Console Logger Started
13:34:19:829 [kdrive.baos.BaosConnection] Connect /dev/ttyAMA0
```

So far, the program starts and connects to the serial port.

```
13:34:19:846 [kdrive.baos.ProtocolDecoder] Tx : A7
13:34:19:868 [kdrive.baos.ProtocolDecoder] Rx : A8 FF FF 00 C5 00 00 00 00 00 04
```

The program requests an PEI identification (A7), which is answered by an confirm (A8) with the following information:

- Individual KNX address (0xFFFF = 15.15.255) of the kBerry BAOS module.
- Serial number (0x00C500000000)
- Reserved byte (0x00)
- Supported features (0x04 = cEMI)

The next telegrams are BAOS specific. The description of data point #1 is requested and the response tells about its configuration (0x57 = transmit, write, communication, low priority) and type (0x01 = 1 bit). See chapter "BAOS Protocol" for more information.

```
13:51:13:823 [kdrive.baos.ProtocolDecoder] Tx : RequestFunctions::GetDatapointDescription
F0 03 00 01 00 01
13:51:13:846 [kdrive.baos.ProtocolDecoder] Rx : ResponseFunctions::GetDatapointDescription
F0 83 00 01 00 01 00 01 00 57 01
```

The value of data point #1 is requested. Its value is 0.

```
13:51:13:856 [kdrive.baos.ProtocolDecoder] Tx : RequestFunctions::GetDatapointValue
F0 05 00 01 00 01 00
13:51:13:878 [kdrive.baos.ProtocolDecoder] Rx : ResponseFunctions::GetDatapointValue
F0 85 00 01 00 01 00 01 00 01 00
13:51:13:881 [BaosEventListener] Received datapoint value for id 1 00
```

Start Net'n Node (see section "Monitoring KNX using Net'n Node" for more info about Net'n Node and the installation of a license file).

Connect Net'n Node to your KNX bus (left vertical tool bar: **Scan** for interfaces and **open** the one, which connects you to your KNX bus). Select Menu **Send KNX/Group Value Write/DPT 1 – Binary – 1 bit**. In the dialog use **Group Address 3/3/1** and **send** the **value TRUE**.

The application running at the Raspberry Pi shows some output, like this.

```
13:59:56:888 [kdrive.baos.ProtocolDecoder] Rx :
IndicationFunctions::DatapointValueIndication
```

```
F0 C1 00 03 00 02 00 03 18 01 01 00 01 18 01 01
13:59:56:925 [BaosEventListener] Received datapoint value for id 3 01
13:59:56:928 [BaosEventListener] Received datapoint value for id 1 01
```

Since ETS has configured data point #1 and #3 to be in group address 3/3/1, the send request of Net'n Node causes the update (value indication) of both data points.

Press **Enter** to exit the application.

For more information about the BAOS SDK, see Weinzierl web page for kBerry.

To shut down the system, exit all screen sessions and enter

```
sudo systemctl poweroff
```

11 BAOS Protocol

In this chapter describes the basics of the BAOS protocol.

11.1 BAOS Frame

The BAOS protocol is a protocol between the Development Board and the BAOS Module. Its format is as follows:

BAOS protocol															
Byte 0								Byte 1							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Main service 0xF0								Sub service							

BAOS protocol																											
Byte 2								Byte 3								Byte 4								Byte 5							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Start item																Number of items															

BAOS protocol																											
Byte 6								Byte 7								Byte 8								...	Byte n-1						
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0
Error code or First item ID																First item data															

BAOS protocol																											
Byte n								Byte n+1								Byte n+2								...	Byte m-1						
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0
Next item ID																Next item data															

... More items ...

BAOS protocol																											
Byte q								Byte q+1								Byte q+2								...	Byte l-1						
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0
Last item ID																Last item data															

The main service for BAOS is always 0xF0. The sub service is the command, whether to read, write an item. The items are the datapoints/parameter bytes/server items/etc.

11.2 Some Important Services and their Responses

11.2.1 GetDatapointValue.Req

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

11.2.2 GetDatapointValue.Res

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 data points in this response
+6	First DP ID	2		ID of first data point
+8	First DP state	1		State byte of first data point
+9	First DP length	1		Length byte of first data point
+10	First DP value	1 – 14		Value of first data point
...	...	...	...	...
+n – 4	Last DP ID	2		ID of last data point
+n – 2	Last DP state	1		State byte of last data point
+n – 1	Last DP length	1		Length byte of last data point
+n	Last DP value	1 – 14		Value of last data point

Example: Read values of data point #1 and #2

Data point #1 is configured as a 1 bit value (= 0) and #2 is a 2 byte value (= 0x8899).

```
APP:  F0 05 0001 0002                                  GetDatapointValue.Req
BAOS: F0 85 0001 0002 0001 00 01 00 0002 00 02 88 99   GetDatapointValue.Res
```

11.2.3 DatapointValue.Ind

DatapointValue.Ind is not a request/response service. It is an automatic notification if a value of a data point changes.

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0xC1	Subservice code
+2	StartDatapoint	2		ID of first data point
+4	NumberOfDatapoints	2		Number of data points in this indication

+6	First DP ID	2		ID of first data point
+8	First DP state	1		State byte of first data point
+9	First DP length	1		Length byte of first data point
+10	First DP value	1 – 14		Value of first data point
...	...	...	...	...
+n – 4	Last DP ID	2		ID of last data point
+n – 2	Last DP state	1		State byte of last data point
+n – 1	Last DP length	1		Length byte of last data point
+n	Last DP value	1 – 14		Value of last data point

Example: Data point #3 has been changed by a KNX message.

It is configured as a one byte value (= 0x55).

```
BAOS:          F0 C1 0003 0001 0003 81 55                    DatapointValue.Ind
```

11.2.4 SetDatapointValue.Req

Offset	Field	Size	Value	Description
+0	MainService	1	0xF0	Main service code
+1	SubService	1	0x06	Subservice code
+2	StartDatapoint	2		Lowest ID of data points to set
+4	NumberOfDatapoints	2		Number of data points to set
+6	First DP ID	2		ID of first data point
+8	First DP command	1		Command byte of first data point
+9	First DP length	1		Length byte of first data point
+10	First DP value	1 – 14		Value of first data point
...	...	...	...	...
+n – 4	Last DP ID	2		ID of last data point
+n – 2	Last DP command	1		Command byte of last data point
+n – 1	Last DP length	1		Length byte of last data point
+n	Last DP value	1 – 14		Value of last data point

11.2.5 SetDatapointValue.Res

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	0x0000	
+6	ErrorCode	1	0x00	

Example: Set value of data point #5

Data point #5 is configured as a 1 bit value and will be changed to 1. The new value will also be sent to the KNX bus.

```
APPLICATION:  F0 06 0005 0001 0005 03 01 01          SetDatapointValue.Req
BAOS:         F0 86 0005 0000 00                     SetDatapointValue.Res
```

11.2.6 GetParameterByte.Req

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

11.2.7 GetParameterByte.Res

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

Example: Read parameter bytes #8 - #16

The parameter bytes have the values 0x11, 0x22, ... 0x88.

```
APPLICATION:  F0 07 0008 0008                        GetParameterByte.Req
BAOS:         F0 87 0008 0008 11 22 33 44 55 66 77 88  GetParameterByte.Res
```

The BAOS protocol offers more services. For complete information about these services, commands, error codes, etc. see document **KnxBAOS_Protocol_v2.pdf**.

11.3 BAOS Server Items

Server items are internal data of the BAOS module. These data deliver information about internal states of the module and some of them can be changed to alter the behaviour of the BAOS module.

11.3.1 GetServerItem.Req

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

11.3.2 GetServerItem.Res

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

Example: Read the serial number

This reads the server item #8 (serial number). The answer from the BAOS contains the serial number 0x112233445566.

```
APPLICATION:  F0 01 0008 0001                          GetServerItem.Req
BAOS:         F0 81 0008 0001 0008 06 11 22 33 44 55 66    GetServerItem.Res
```

The following table shows all available server items for the 8xx BAOS module.

ID	Description	Size	Access	Indication
1	**Hardware type** Used to identify the hardware type. The coding is manufacturer specific. It is mapped to the property PID_HARDWARE_TYPE in the device object.	6	R	No
2	**Hardware version** Version of the ObjectServer hardware. Coding Example: 0x10 = Version 1.0	1	R	No
3	**Firmware version** Version of the ObjectServer firmware. Coding Example: 0x10 = Version 1.0	1	R	No
4	**KNX manufacturer code DEV** KNX manufacturer code of the device, not modified by the ETS. It is mapped to the property PID_MANUFACTURER_ID in the device object.	2	R	No
5	**KNX manufacturer code APP** KNX manufacturer code loaded by ETS. It is mapped to bytes 0 and 1 of the property PID_APPLICATION_VER in the application object.	2	R	No
6	**Application ID (ETS)** ID of the application loaded by ETS. It is mapped to bytes 2 and 3 of the property PID_APPLICATION_VER in the application object.	2	R	No
7	**Application version (ETS)**	1	R	No

	Version of the application loaded by ETS. It is mapped to byte 4 of the property PID_APPLICATION_VER in the application object.			
8	**Serial number** Serial number of the device. It is mapped to the property PID_SERIAL_NUMBER in the device object.	6	R	No
9	**Time since reset** Uptime of the module in milliseconds.	4	R	No
10	**Bus connection state** State of the KNX bus connection (0 = disconnected, 1 = connected). This ServerItem sends an indication every time its value has changed.	1	R	Yes
11	**Maximum buffer size** Maximum size of BAOS buffer size in bytes.	2	R	No
12	**Length of description string** The BAOS module does not support the storing of description strings for the group objects, because it would lead to very long download times. So this valus will always return 0.	2	R	No
13	**Baudrate** The current baudrate of the BAOS protocol: 0 = unknown, 1 = 19200 baud, 2 = 115200 baud. If this value is changed, the communication to the BAOS module must be established again. Care must be taken changing the baud rate, since the module will reset it to 19200 baud at a reset (also after ETS download).	1	R/W	No
14	**Current buffer size** Current size of BAOS buffer size in bytes.	2	R/W	No
15	**Programming mode** Current state of the programming mode (LED) of the BAOS module. This ServerItem sends an indication every time its value has changed.	1	R/W	Yes
16	**Protocol Version (Binary)** Version of the ObjectServer binary protocol. Coding Example: 0x20 = Version 2.0	1	R	No
17	**Indication Sending** Is sending of indications active? 0 = off, 1 = active. This controlls the sending of both indications: ServerItems and Datapoints.	1	R/W	No
20	**Individual Address** The individual KNX address of the device.	2	R/W	Yes
37	**Device friendly name** A NULL terminated string of the user given name of this device. Maximum of 30 bytes.	30	R/W	Yes
38	**Maximum datapoints** Maximum number of datapoints this module supports.	2	R	No
39	**Configured datapoints**	2	R	No

	Number of configured datapoints.			
40	**Maximum parameter bytes** Number of available parameter bytes.	2	R	No
41	**Download counter** ETS download counter.	2	R	No

Important: All values which are longer than one byte are big-endian interpreted.

12 About KNX

In this chapter describes the basics of KNX and its usage in the BAOS Modules.

The standardized bus system *KNX* plays a more and more important role in building automation. A lot of devices use this protocol.

In general the KNX system is a bus system for building control. All connected devices are communicating over the same bus. The information is transported via a communication stack. Every single device connected to this bus has its own micro-controller on board. There is no central control device. The bus is structured completely decentralized.

One main advantage of this design is fault tolerance. An error within one device has little effect on the others. All connected participants are operating independently.

Connected to the KNX bus are sensors and actuators. The sensors are generating telegrams. The actuators receive these messages and act accordingly. It's also possible a device is a sensor and an actuator. So it's possible to send and receive data.

12.1 KNX Twisted Pair Bus System

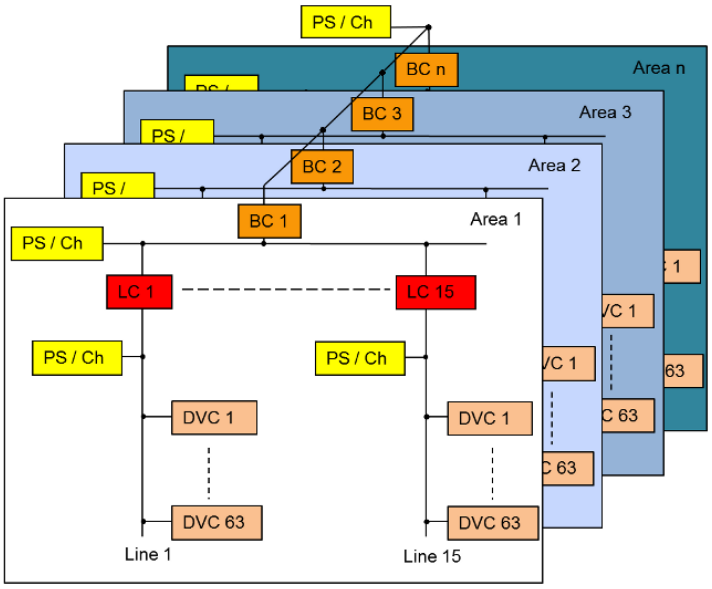
A minimum TP1 KNX installation consists of the following components:

- a KNX **power supply (PS)** unit containing a **choking coil (Ch)**
- **bus devices (DVC):** some sensors (at least one)
- **bus devices (DVC):** some actuators (at least one)

A more complex KNX installation has additionally the following components:

- **Line coupler (LC)** to connect more devices (DVC) to a line.
- **Backbone coupler (BC)** to connect more areas.

The KNX bus in its maximum expansion can hold many devices. The topology is basically like this:



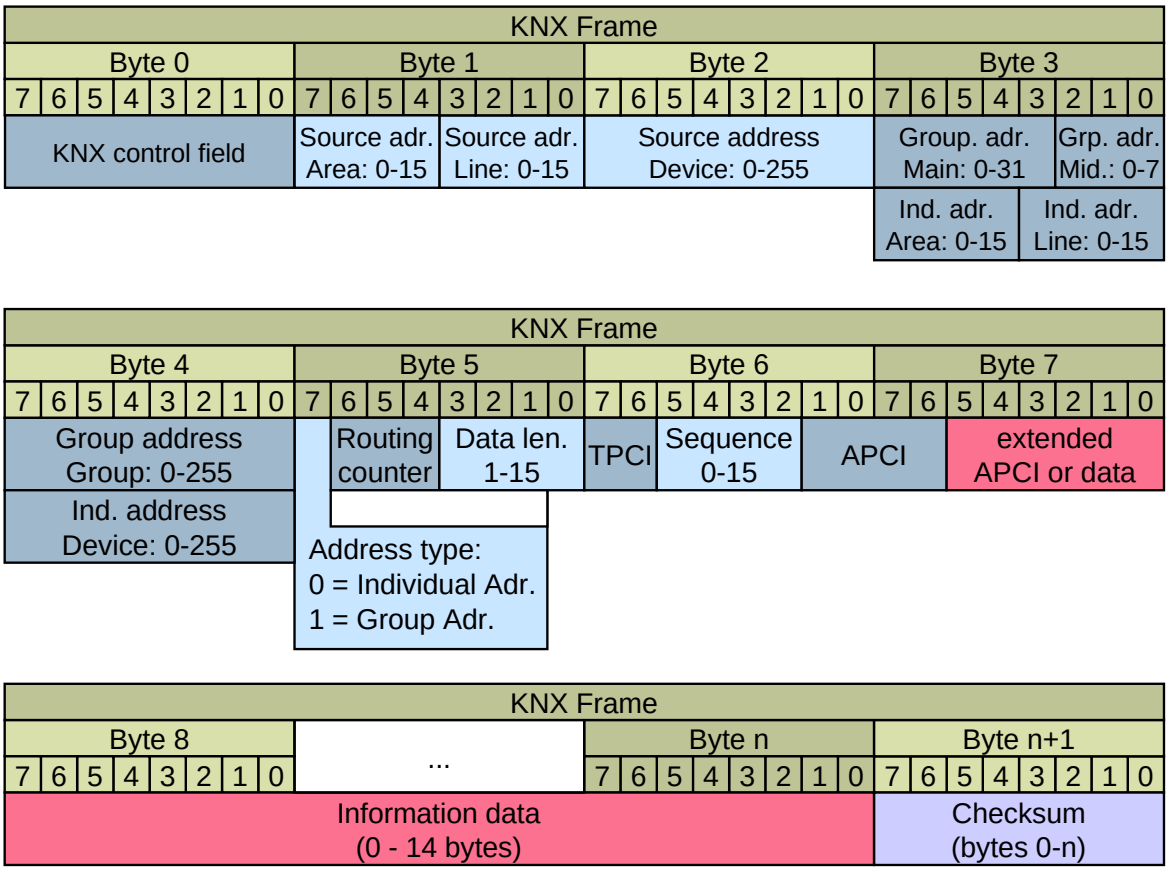
The KNX twisted pair bus system (KNX TP) provides to all connected devices data and the operating voltage over the same two-wire line. The nominal bus voltage is 29 V.

The bus transfer rate is 9600 bit/s respective about 50 telegrams per second transfer rate.

12.1.1 KNX Twisted Pair Telegrams

The information exchange between KNX devices is based on telegrams. A telegram is a clear defined sequence of bytes. It is segmented in several fields. Here is a standard connection less KNX telegram. A standard connection less telegram is mainly used for data exchange between KNX devices (sensors and actuators). Connection oriented telegram are generally used for downloads by the ETS. Standard telegrams have a data length up to 15 bytes. For more data an extended frame is used, which can hold up to 254 bytes of data.

Standard frame:



The fields of this telegram are:

The **KNX control field** has the following information:

- **Bit #7**: Frame type:
  - 1 = L_Data_Standard frame
  - 0 = L_Data_Extended frame
- **Bit #6**: Poll data/data:
  - 0 = L_Data frame
  - 1 = L_Poll_Data frame
- **Bit #5**: Repeat flag:
  - 0 = repeated L_Data_Standard frame
  - 1 = not repeated L_Data_Standard frame
- **Bit #4**: unused, must be 1
- **Bit #3 - 2**: Priority, leave this at 11
- **Bit #1 - 0**: unused, must be 00

The **source address** describes the sender address (individual address).

The **group address** determines which bus devices will receive the telegram. The target address is a group address, which can address many devices at the same time. Byte #5, bit #7 determines the **address type**: 1 = group address.

The **individual address** determines which bus devices will receive the telegram. The target address is an individual address, which addresses exactly one device. Byte #5, bit #7 determines the **address type**: 0 = individual address.

For more info about addressing, see Section “Addressing Modes”.

The **routing** counter determines how many hops remain. The counter is decremented every time a frame passes a coupler. At the value 0 the frame will be removed.

The **data length** field describes the number of information bytes in this frame (starting at byte #7).

The **extended application protocol control information (APCI)** is the service code from/for the application layer. But it can also contain information data. This is the case for ValueWrite telegrams. In this case the remaining information data bytes are not necessary and are skipped.

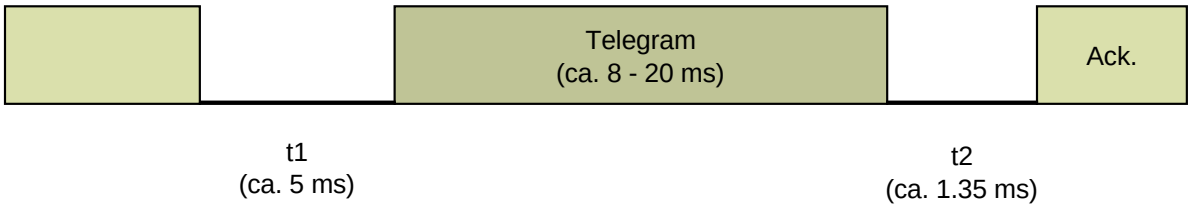
The **information data** are the telegram payload. The size of this field can range from zero to 14 bytes for standard frames. In case of zero bytes the extended APCI can hold the data.

The **checksum** validates the frame. It is calculated by xor'ing all bytes and at last by 0xFF: **chksum = b0 xor b1 xor b3 xor ... xor bn xor 0xFF**.

For more information see **KNX System Specifications/03_02_02 Communication Medium TP 1** available at the KNX Specifications page.

12.1.2 Telegram Timings

When an event occurs (e. g. push-button is pressed), the bus device sends a telegram to the bus. The transmission starts after the bus has been unoccupied for at least the time period t1. Once the transmission is complete, the telegram must be acknowledged (ACK) after time t2. All addressed bus devices acknowledge the reception of the telegram simultaneously.



12.1.3 Bus monitoring with Net'n Node

It is possible to monitor the KNX communication with Net'n Node. In the **Acess Port Configuration** panel use **Busmon** as Layer. In this case all telegrams (including acknowledgements) are visible in the telegram list. In this mode no routing and filtering is active since this is done in upper layers. Sending telegrams in Net'n Node is not possible in bus monitor mode.

Selecting **Link Layer** again, shows all telegrams from the link layer and enables to edit and send telegrams again.

12.2 KNX Radio Frequency Bus System

Radio Frequency RF is the wireless alternative in the KNX standard. In locations that are ideally suited for cabling KNX RF is used for wireless data transmission within a floor or a complete building.

KNX RF uses a Frequency Shift Keying (FSK) for data modulation frequency with a central frequency of 868.3 MHz. With data rates of 16384 baud a similar count of frames can be transmitted as on TP.

12.2.1 KNX Radio Frequency Telegrams

Block 1			
Byte 0	Byte 1	Byte 2	Byte 3
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Length (Byte 1 - 9)	C-Field 0x44	Esc 0xFF	RF-Info

Block 1			
Byte 4	Byte 5	Byte 6	Byte 7
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Serial number or domain address			

Block 1			
Byte 8	Byte 9	Byte 10	Byte 11
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Serial number or domain address		CRC hi (Byte 0 - 9)	CRC lo (Byte 0 - 9)

Block 2				
Byte 12	Byte 13		Byte 14	Byte 15	
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	
KNX control field	Source adr. Area: 0-15	Source adr. Line: 0-15	Source address Device: 0-255	Group. adr. Main: 0-31	Grp. adr. Mid.: 0-7
				Ind. adr. Area: 0-15	Ind. adr. Line: 0-15

Block 2				
Byte 16	Byte 17	Byte 18	Byte 19	
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	
Group address Group: 0-255	L/NPCI 0xE6	TPCI	APCI	extended APCI or data
Ind. address Device: 0-255				

Block 2			
Byte 20	...	Byte n	Byte n+1
7 6 5 4 3 2 1 0		7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Information data (0 - 8 bytes)		CRC hi (Byte 12 - n-1)	CRC lo (Byte 12 - n-1)

The first data block follows with some control information and the serial number or the domain address. The first block has a fixed length of 10 data bytes and an own checksum of 2 bytes.

The application data starts in block 2, which has a maximum length of 16 byte plus 2 bytes checksum. For longer telegrams additional blocks may follow. The coding of the data in block 2 and following are according to the telegram content used for twisted pair.

The fields of this telegram are:

The **C-Field** has the following information:

- Always Send/No reply = 0x44

The **RF Info** has the following information:

- **Bit #7 - 4**: unused, must be 0000
- **Bit #3 - 2**: Signal strength:
  - 00 = no data
  - 01 = weak
  - 10 = medium
  - 11 = strong
- **Bit #1**: State of battery:
  - 0 = weak
  - 1 = OK
- **Bit #0**: Unidirectional flag:
  - 0 = sent by bidirectional device
  - 1 = sent by unidirectional device

The Serial number or domain address is used to separate more RF networks. All connected devices must have the same domain address.

The **KNX control field** has the following information:

- **Bit #7 - 4**: usually 0000
- **Bit #3 - 0**: Extended frame format (EFF):
  - 0000 = L_Data_Extended frame
  - 01xx = LTE-HEE extended address type

The **source address** describes the sender address (individual address).

The **group address** determines which bus devices will receive the telegram. The target address is a group address, which can address many devices at the same time. Byte #17, bit #7 determines the **address type**: 1 = group address.

The **individual address** determines which bus devices will receive the telegram. The target address is an individual address, which addresses exactly one device. Byte #17, bit #7 determines the **address type**: 0 = individual address.

For more info about addressing, see Section “Addressing Modes”.

The **L/NPCI** has the following information:

- **Bit #7**: Destination address flag (DAF):
  - 1 = Group address
  - 0 = Individual address
- **Bit #6 - 4**: Repetition counter.

- **Bit #3 - 1**: Data Link Layer Frame Number.
- **Bit #0**: Serial number/domain address:
  - 0 = Block 1 contains the serial number
  - 1 = Block 1 contains the domain address

The **TPCI** has the following information:

- **Bit #5 - 2**: Sequence number.
- **Bit #1 - 0**: High bits of APCI.

The **transport protocol control information (TPCI)** is a code from the transport layer.

The **application protocol control information (APCI)** is a code from the application layer. There are two different sizes for the APCI. Some services support a four bit APCI (like A_GroupValue_Write), but the most (management) services requires ten bits and the APCI reaches up to the end of byte #7. If only four bits are used, the last 6 bits can be used as data. This is used in a write or in a response of a group object, when the group object value size is equal or less than six bits.

The **extended application protocol control information (APCI)** is a code from the application layer. But it can also contain information data. This is the case for ValueWrite telegrams. In this case the remaining information data bytes are not necessary and are skipped.

The information data are the telegram payload. The size of this field can range from zero to 8 bytes. In case of zero bytes the extended APCI can hold the data.

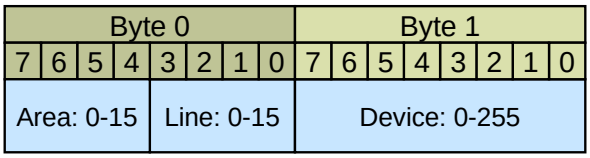
The **checksum** validates the block. It is calculated according to IEC 870-5.

For more information see **KNX System Specifications/03_02_05 Communication Medium RF** available at the KNX Specifications page.

12.3 Addressing Modes

KNX provides mainly three different addressing modes.

- An **individual address** must be unique within a KNX installation. A device is addressed this way while configuring it via ETS.



- Communication between devices in an installation is carried out via **group addresses**.

The default in ETS is a 3-level (main group/middle group/subgroup) structure.

Byte 0								Byte 1							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Main: 0-31					Mid: 0-7			Sub: 0-255							

- The group address 0/0/0 is reserved for **broadcast messages**. This message is for all available devices and is used for downloading an individual address.

12.4 Data Point Types

The data point type describes the size, the range and its representation of a value. The most common types are:

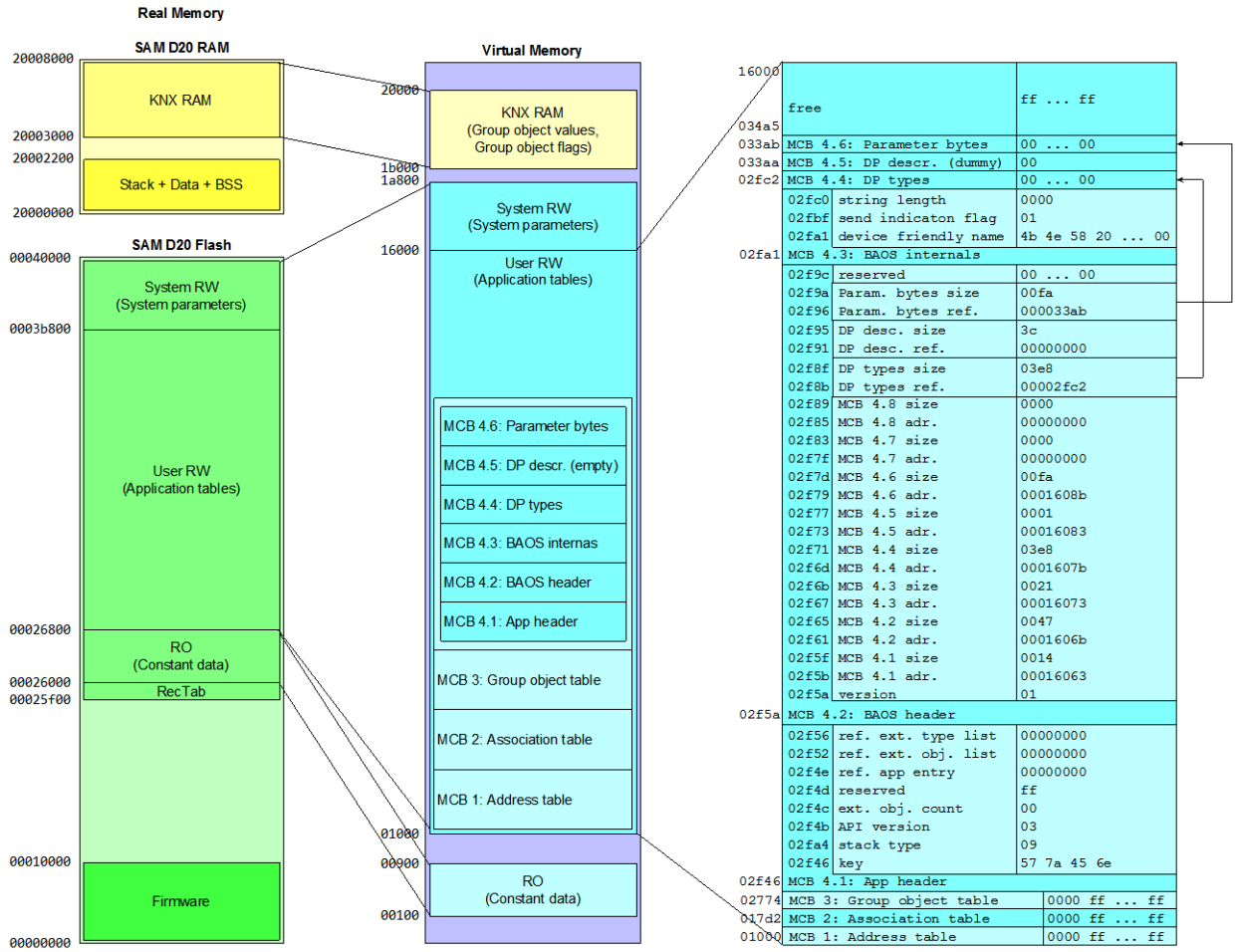
Base type	DPT	Name	Encoding (Representation)	Value range	Size (bits)
boolean	1.001	DPT_Switch	0 = Off, 1 = On	0 – 1	1
	1.007	DPT_Step	0 = Decrease, 1 = Increase	0 – 1	1
	1.008	DPT_UpDown	0 = Up, 1 = Down	0 – 1	1
uint4	3.007	DPT_Control_Dimming	0x08 - 0x0F = Increase, 0x01 - 0x07 = Decrease, 0x00 = Stop dimming	0x00 - 0x0F	4
uint8	4.001	DPT_Char_ASCII	0 - 127 = ASCII Character	0x00 - 0x7F	8
	4.002	DPT_Char_8859_1	0 - 255 = Latin 1 Character (ISO 8859.1)	0x00 - 0xFF	8
	5.001	DPT_Scaling	0 - 255 = Scaling in Percent (128 = 50%)	0x00 - 0xFF	8
	5.004	DPT_Percent_U8	0 - 255 = Scaling in Percent (0% - 255%)	0x00 - 0xFF	8
int8	6.001	DPT_Percent_V8	-128 - 127 = Relative value in Percent (-128% - 127%)	0x00 - 0xFF	8
	6.010	DPT_Value_1_Count	-128 - 127 = Counter pulse	0x00 - 0xFF	8
uint16	7.001	DPT_Value_2_Ucount	0 - 65535 = Counter value	0x0000 - 0xFFFF	16
int16	8.001	DPT_Value_2_Count	-32768 - 32767 = Counter value	0x0000 - 0xFFFF	16
	8.010	DPT_Percent_V16	-32768 - 32767 = Value in percent (-327.68% - 327.67%)	0x0000 - 0xFFFF	16
float16	9.001	DPT_Value_Temp	-273 - 670760 = Temperature value (Celsius)	-671088.64 - 670760.96	16
unint32	12.001	DPT_Value_4_Ucount	0 - 4294967295 = Counter value	0x00000000 - 0xFFFFFFFF	32
int32	13.001	DPT_Value_4_Count	-2147483648 - 2147483647 = Counter value	0x00000000 - 0xFFFFFFFF	32

For more info see **KNX System Specifications/03_07_02 Datapoint Types** available at the KNX Specifications page.

12.5 Virtual Memory Map of the BAOS Module

KNX uses a virtual memory mapping mechanism to access certain memory regions via the bus. These memory regions contain information mainly configured by the ETS.

The following figure shows the real memory map of the KNX BAOS Module and its virtual memory map. The virtual memory map is for the ETS to read and write the configuration. The most important part for the application is the User RW memory. It is divided in MCB parts.



The KNX Stack uses three configuration tables to handle group communication:

- MCB 1: Address table
- MCB 2: Association table
- MCB 3: Group object table

The tables are downloaded by ETS.

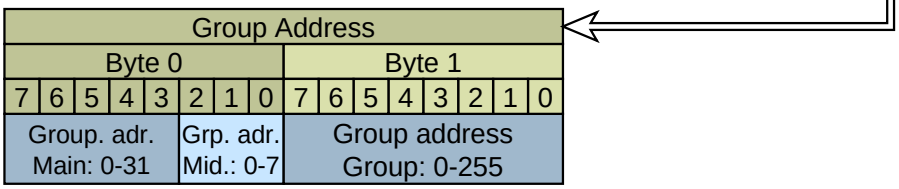
The following sections show the MCB parts (User RW) and their addresses.

12.5.1 Address Table (MCB 1)

- Virtual address range: 0x01000 - 0x017D1
- Real address range: 0x00026800 – 0x00026FD1
- Size: 0x007D2

The Address Table contains all group addresses, which are used by the device to send or receive the group telegrams via Group Objects (also called Communication Objects or Data Points). The Address Table has a two byte length field followed by the Group Addresses. The Group Addresses have to be sorted.

Address Table:	Example:
MCB 1 + 0: Number of entries	1400: 00 05 5 entries
MCB 1 + 2: Group Address #1	1402: 15 0d Group Address #1: 2/5/13
MCB 1 + 4: Group address #2	1404: 15 0e Group Address #2: 2/5/14
MCB 1 + 6: Group Address #3	1406: 15 0f Group address #3: 2/5/15
MCB 1 + 8: Group address #4	1408: 1a 01 Group Address #4: 3/2/1
MCB 1 + a: etc.	140a: 1a 02 Group address #5: 3/2/2



A Group Address is a two octet value which is sent on the medium as it is stored in the Group Address Table.

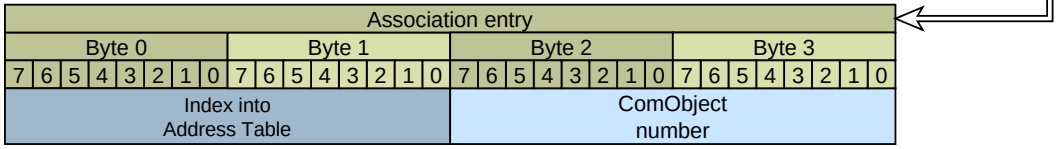
12.5.2 Association Table (MCB 2)

- Virtual address range: 0x017D2 – 0x02773
- Real address range: 0x00026FD2 – 0x00027F73
- Size: 0x00FA2

The Association Table holds the relations between the Group Addresses and the Group Objects (see below). Every entry in this table is a link of a Group Address to a Group Object. So each entry consists of a connection number and Group Object number.

The table has a two byte length field followed by the associations.

Association Table:	Example:
MCB 2 + 00: Number of entries	1500: 00 07 7 entries
MCB 2 + 02: Index into Address Table | ComObject #	1502: 00 04 00 01 AT entry #4 | Object #1
MCB 2 + 06: Index into Address Table | ComObject #	1506: 00 05 00 02 AT entry #5 | Object #2
MCB 2 + 0a: Index into Address Table | ComObject #	150a: 00 04 00 05 AT entry #4 | Object #5
MCB 2 + 0e: Index into Address Table | ComObject #	150e: 00 05 00 06 AT entry #5 | Object #6
MCB 2 + 12: Index into Address Table | ComObject #	1512: 00 03 00 07 AT entry #3 | Object #7
MCB 2 + 16: Index into Address Table | ComObject #	1516: 00 01 00 05 AT entry #1 | Object #5
MCB 2 + 1a: etc.	151a: 00 02 00 06 AT entry #2 | Object #6



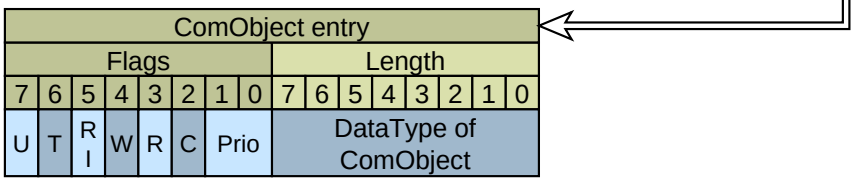
12.5.3 Group Object Table (MCB 3)

- Virtual address range: 0x02774 – 0x02F45
- Real address range: 0x00027F74 – 0x00028745
- Size: 0x007D2

Group Objects (also known as Communication Objects) represent the KNX Data Points which are used for runtime communication. The Group Object Table stores the descriptors of the Group Objects.

Group Objects are part of the KNX application layer and represent the interface between the application and the KNX Stack. Using Group Objects, the application is able to send data to and receive data from the KNX bus via Group Telegrams. The application accesses the Group Objects via their indices, the Group Object numbers. The application has no knowledge about the Group Addresses used on the network.

ComObjects:	Example:
MCB 3 + 0: Number of ComObjects	1600: 00 05 5 ComObjects
MCB 3 + 2: ComObject #1: flags | length	1602: 57 00 -T-W-C | 1 bit
MCB 3 + 4: ComObject #2: flags | length	1604: 57 03 -T-W-C | 4 bit
MCB 3 + 6: ComObject #3: flags | length	1606: 57 00 -T-W-C | 1 bit
MCB 3 + 8: ComObject #4: flags | length	1608: 57 03 -T-W-C | 4 bit
MCB 3 + a: ComObject #5: flags | length	160a: 57 07 -T-W-C | 1 byte
MCB 3 + c: etc.	160c: etc.



The Group Object Table is a sorted list of Group Objects. The index of the Group Object Table is the Group Object number. The first Group Object has the index #1. The Group Object numbers are referenced in the Association Table.

The Group Object Table has a two byte length field. Each Group Object descriptor consists of a byte holding configuration flags and a byte indicating the size of the Group Object value.

The **flags** have the following coding:

- **Bit #7**: **U**pdate (from response frame):
  - 0 = Value will not be changed by a response frame from the bus (use this as default).
  - 1 = Value will be changed by a response frame from the bus (flags W, C must also be enabled).
- **Bit #6**: **T**ransmit (send):

  - 0 = Changed value will not be sent to the bus.
  - 1 = Changed value will be sent to the bus (flag C must also be enabled).
- **Bit #5**: **R**ead on **I**nit.:
  - 0 = Value is not read at initialization time.
  - 1 = Value is read at initialization time (flags U, T, W, C must also be enabled).
- **Bit #4**: **W**rite via bus:
  - 0 = Value is not writable.
  - 1 = Value is writable from bus (flag C must also be enabled)
    Input data points (actuators) should set this to 1, so the value can be written by a KNX telegram.
- **Bit #3**: **R**ead via bus:
  - 0 = Value is not readable (use this as default).
  - 1 = Value is readable from bus (use this only if you want to use GroupValueRead requests, flag C must also be enabled). Input data points (actuators) should set this to 1, so the value can be read by other KNX devices. Only one actuator in every Group Address should enable this.
- **Bit #2**: **C**omObject (Data Point) enabled:
  - 0 = disabled
  - 1 = enabled (i.e. linked by ETS to group address)
- **Bit #1 - 0**: **Prio**rity, leave it to 11 (low).

The **length** field defines the data length of the Group Object.

Length Field Type	Value	Size	Length Field Type	Value	Size
VTYPE_BIT_1	0	1 Bit	VTYPE_BYTE_5	15	5 Bytes
VTYPE_BIT_2	1	2 Bits	VTYPE_BYTE_7	16	7 Bytes
VTYPE_BIT_3	2	3 Bits	VTYPE_BYTE_9	17	9 Bytes
VTYPE_BIT_4	3	4 Bits	VTYPE_BYTE_11	18	11 Bytes
VTYPE_BIT_5	4	5 Bits	VTYPE_BYTE_12	19	12 Bytes
VTYPE_BIT_6	5	6 Bits	VTYPE_BYTE_13	20	13 Bytes
VTYPE_BIT_7	6	7 Bits	VTYPE_BYTE_15	21	15 Bytes
VTYPE_BYTE_1	7	1 Bytes	VTYPE_BYTE_16	22	16 Bytes
VTYPE_BYTE_2	8	2 Bytes	VTYPE_BYTE_17	23	17 Bytes
VTYPE_BYTE_3	9	3 Bytes	VTYPE_BYTE_18	24	18 Bytes
VTYPE_BYTE_4	10	4 Bytes	VTYPE_BYTE_19	25	19 Bytes
VTYPE_BYTE_6	11	6 Bytes	VTYPE_BYTE_20	26	20 Bytes
VTYPE_BYTE_8	12	8 Bytes	VTYPE_BYTE_21	27	21 Bytes
VTYPE_BYTE_10	13	10 Bytes	…	…	…
VTYPE_BYTE_14	14	14 Bytes	VTYPE_BYTE_54	60	54 Bytes

12.5.4 Application Header (MCB 4.1)

- Virtual address range: 0x02F46 – 0x02F59
- Real address range: 0x00028746 – 0x00028759
- Size: 0x00014

The Application Header contains some internally used data, like stack type, etc.

12.5.5 BAOS Header Block (MCB 4.2)

- Virtual address range: 0x02F5A – 0x02FA0
- Real address range: 0x0002875A – 0x000287A0
- Size: 0x00047

The BAOS Header Block contains pointer and sizes of data used by the BAOS firmware.

12.5.6 BAOS Internals (MCB 4.3)

- Virtual address range: 0x02FA1 – 0x02FC1
- Real address range: 0x000287A1 – 0x000287C1
- Size: 0x00021

The BAOS Internals contains information and flags for the BAOS firmware. The device friendly name is stored there and the flag for sending indications.

12.5.7 Data Point Types (MCB 4.4)

- Virtual address range: 0x02FC2 - 0x033A9
- Real address range: 0x000287C2 – 0x00028BA9
- Size: 0x003e8

The Data Point Types are used for the generic ETS database. In the generic database each data point can be set to a certain type (disabled, 1 bit, 4 bit, 1 byte, etc.). There are these types stored.

If an individual database for ETS is used, this region is free to use.

12.5.8 Data Point Descriptions (MCB 4.5)

- Virtual address range: 0x033AA – 0x033AA
- Real address range: 0x00028BAA – 0x00028BAA
- Size: 0x0001

The Data Point Descriptions are not used for the KNX BAOS 830, 832 and 838 Modules. It would lead to very long ETS download times. So the descriptions remain in the ETS database only. This Sub-MCB contains only one dummy byte.

The KNX BAOS 777 device, for example, uses description strings.

12.5.9 Parameter Bytes (MCB 4.6)

- Virtual address range: 0x033AB – 0x034A4
- Real address range: 0x00028BAB – 0x00028CA4
- Size: 0x00FA

All Parameter Bytes of the BAOS module are stored in this region. The generic ETS database offers 250 bytes. If an own database is created, the number of parameter bytes can be changed as long as virtual memory is available.

12.5.10 Free Virtual Memory

- Virtual address range: 0x034A5 – 0x15FFF
- Real address range: 0x00028CA5 – 0x0003B7FF
- Size: 0x12B5B

This region is free to use. It can be used to store more parameter bytes. For this, the creation of an ETS database is necessary. See chapter "Individual ETS Entries".

12.6 Access Protection

Nearly every item in the KNX world can be read and write protected. Most of the properties are writing protected by an access key and readable by everyone. KNX defines 4 access levels, each to reading and writing:

- **Level 0**: Access for system manufacturer. This access level is completely restricted and only accessible for Weinzierl Engineering. Do not use it.
- **Level 1**: Access for device manufacturer. This access level is mainly for changing production properties, like serial number, manufacturer ID, etc. Its access key is 0x12345678. This can be changed as described below. It is recommended to change this key and to keep it secret.
- **Level 2**: Access for tools. This access level is used by tools like the ETS. Its access key is 0xFFFFFFFF. This can be changed as described below. It is recommended to leave this key as it is. It can be changed by the end customer as he likes. In ETS this key must also be set (BCU Key).
- **Level 3**: Access for everyone. This access level is not protected in any way. So it is accessible for everyone and has no key.

An access key can be changed if you have access to its level. I. e. you know the key.

12.6.1 Access via Net'n Node

Start Net'n Node and connect it to the KNX bus which is also connected to the KNX BAOS Module. See section "Monitoring KNX using Net'n Node" for more information about this tool.

Select menu **Tools/Access Protection** to manage access keys. In the dialog enter the correct individual address and the key FFFFFFFF in **Authorize with device/Key**. Press Test Key and look at the **Returned Level**. It should read 2, so you have access for level 2.

Enter the key for level 1: 12345678 and hit **Test Key** again. Now it should read level 1. Change the key in **New Key** as you like (e. g. 11223344) and remember it very hard. Hit **Set Key** and the new authorization key for level 1 is stored.

This key is, for example, needed to change some properties of the KNX BAOS Module. See next section.

12.7 Important Properties

Every KNX device has its own set of properties. Some of them can only be read and store just information, some can be written and some will act accordingly. Here are the most important ones.

12.7.1 The 0 - Device Object

11 - Serial Number: Serial number of the device. This 6 bytes value can be written if the access key level 1 is set. The first 2 bytes of the serial number must be identical to the manufacturer identifier. The remaining 4 bytes are manufacturer specific, but they must be unique. The prefixed manufacturer ID ensures, that all serial numbers in the KNX world are unique.

12 - Manufacturer Identifier: ID of the manufacturer given from the KNX Association. This 2 bytes value can be written if the access key level 1 is set. **Warning**: This value must match to the ETS database or else the download will fail.

15 - Order Info: This 10 bytes value is an identifier for your product a catalogue. It is used for ordering more much products. This value is shown by ETS in *Diagnostic/Device Info*. To change the value access key level 1 is required.

19 - Manufacturer Data: This 4 bytes value can be some manufacturer specific information about this device, e. g. manufacturing date. Access key level 1 is required.

21 - Description: This 30 bytes value is a string, which stores the device friendly name, also seen in ETS. It is an ASCII string and defaults to "KNX BAOS 83x" or "KNX BAOS 840".

25 - Version: This 2 bytes value is a read only value. It shows the version of the BAOS firmware.

30 - Download Counter: This 2 bytes value is a read only value. It shows the number of ETS downloads.

54 - Programming Mode: This 1 byte value shows the state of the programming mode. It has only two possible values: 0 = off, 1 = on. To write a new individual address to a KNX device, the programming mode must be active (on). This is normally done by pushing the Learn key. Writing this property is an alternative to switch the programming mode by key.

57 - Subnetwork Address: This 1 byte value shows the first part of the individual address. It can also be written to change the individual address. 0x12 means 1.2.x. The last part is stored in the next property.

58 - Device Address: This 1 byte value shows the second part of the individual address. It can also be written to change the individual address. 0x20 means x.x.32.

78 – Hardware Type: This 6 byte value is the hardware type identifier for the device. The first byte must be 0, since all other values are for future use and will be defined by the KNX Association. The next 2 bytes are the manufacturer identifier, the next 2 bytes identify the BAOS module (0x0800) and the last byte the variant (TP = 0x03, RF = 0x04).

82 – RF Domain Address: This 6 byte value is the domain address used by RF medium. It is used to separate groups of devices from other groups. For TP the line and area topology do this, but since RF is sending its telegrams over air, the domain address is the key to separate such groups. This is, of course, also important to protect the own installation from another KNX RF installation in the neighbourhood.

12.7.2 The 1 - Address Table Object

The Address Table is written and managed by the ETS. Its values can be read, but care must be taken to write anything.

12.7.3 The 2 - Association Table Object

The Association Table is written and managed by the ETS. Its values can be read, but care must be taken to write anything.

12.7.4 The 9 - Group Object Table Object

The Group Object Table is written and managed by the ETS. Its values can be read, but care must be taken to write anything.

12.7.5 The 3 - Application 1 Object

The Application 1 Object stores some information of the BAOS application.

12.7.6 The 4 - Application 2 Object

The Application 2 Object is an unused entry. It is specified for an alternative application, but it is not used.

12.7.7 The 8 - cEMI Server Object

The Common EMI server object id used to communicate and change the mode of the BAOS Module. It can be used by Net'n Node to read and write KNX telegrams via the BAOS Module. It can also be used to switch the BAOS Module to certain communication modes (like bus monitor).

51 – Medium Type: This 2 byte value shows all the media types, the device supports. Every medium is represented by its designated bit:

- Bit #5: 1 = IP
- Bit #4: 1 = RF
- Bit #1: 1 = TP

For BAOS Modules 830, 832 and 838, only TP (= 0x0002) is supported.

52 – Communication Mode: Current communication mode of the BAOS Module. This is used to switch the module to different communication modes, like sending/receiving own KNX telegrams via cEMI protocol.

Normally the module is in BAOS mode (0xF0). To use own KNX telegrams via cEMI protocol, the mode must be changed to Link Layer mode (0x00).

See also section "cEMI in the Application" for an example using this property.

54 – Additional Information Types: If present, holds all supported additional info types for cEMI telegrams. In case of KNX BAOS RF 840, only the type 0x02 is supported, which contains one byte RF Info, 6 bytes domain address or serial number and one byte RF LFN.

For more info see **KNX System Specifications/03_06_03 EMI_IMI** available at the KNX Specifications page.

64 – cEMI Supported Communication Modes: This 16 bit mask shows all the supported communication modes, which can be used:

- Bit #3: 1 = Local Transport Layer mode available
- Bit #1: 1 = Link Layer bus monitor mode available
- Bit #0: 1 = Link Layer mode available

For BAOS Modules 830, 832 and 838, the mask is 0x000B.

12.7.8 The 19 – RF Medium Object

56 – RF Domain Address: This property is a copy of property "0 – Device Object/82 – RF Domain Address".

13 How to Change Production Parameters

Every KNX BAOS Module contains a set of parameters to identify it as a product. These parameters can be changed permanently, so even a master reset cannot restore them to default values. Such parameter, like the serial number, manufacturer data, etc. should never be changed after shipping out the device which contains the KNX BAOS Module.

13.1 BAOS Module Config Tool

To change these production values permanently, a tool is available at the BAOS web page. Download from http://www.weinzierl.de, install KnxBAOS Module Config and start it.

This tool can change the production values. It has its own manual.

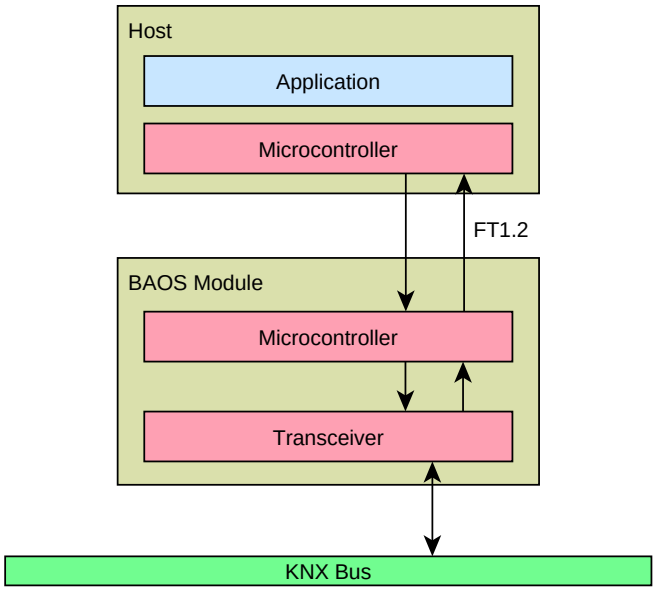
13.2 Net'n Node

It is also possible to change these production values with Net'n Node, since they are all accessible via properties. See sections "Access Protection" and "Important Properties":

- Start Net'n Node and connect it to the KNX bus which is also connected to the KNX BAOS Module.
- Use menu **Tools/Edit Properties** to read the current production values. Enter the individual address of the KNX BAOS Module (default 15.15.255) and hit **Scan**. After a while the dialog window shows the properties of the module.
- Unfold **0 Device Object**, select it and read all values of the properties by hitting **Read**. After a while the values show up.
- Change some values (see section "Important Properties" for more info about these properties):
  11 Serial Number,
  12 Manufacturer Identifier,
  15 Order Info,
  19 Manufacturer Data and
  78 Hardware Type.
- Enter the level 1 access key in **Authorize**: Enable **Use authorize** and enter the key.
- Select each changed property and hit **Write** to write its value back to the module.
- All values should now be updated. Even a master reset cannot change them.

14 FT1.2 Protocol

The communication between the Development Board and the KNX BAOS Module uses the ObjectServer or the cEMI protocol. These protocols are encapsulated in the FT1.2 protocol for reliability reasons. The typical device architecture looks like this:



14.1 General

The FT1.2 protocol is based on the international standard IEC 60870-5-1 and 60870-5-2. An asymmetric transmission procedure is used. I. e. the host initiates a message transfer and the KNX BAOS Module sends a response. The protocol is restricted to point-to-point (no address field) communication.

14.2 Physical

14.2.1 Interface

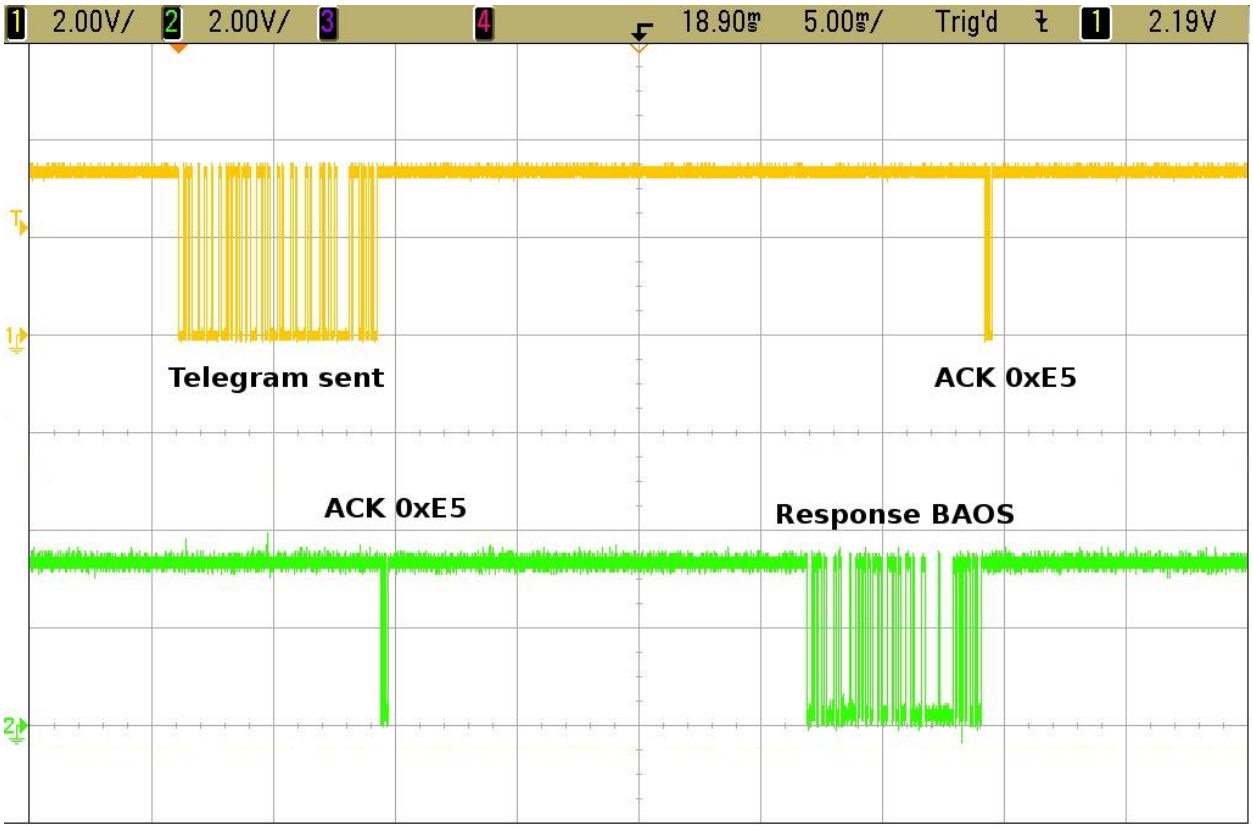
The module and application are connected via a 3-wire connection:

- **RxD**: Received data
- **TxD**: Transmit data
- **GND**: 0 V Ground
- additional for 830 and 840: **Vcc**: Power supply

Data transmission is performed with 8 data bits, even parity and 1 stop bit. The default transmission rate is 19200 baud. Frames have a fixed or variable length.

14.2.2 Timings

The timing of the FT1.2 communication between the application and the KNX BAOS Module are shown in the next figure. The application sends an FT1.2 frame to the KNX BAOS Module which acknowledges it. After a while the module sends a response frame to the application which also acknowledges it.



14.3 FT1.2 Frame Format

The FT1.2 protocol ensures data integrity by using a defined header, a check-sum and a stopping character. There are two frame variants: a fixed frame for reset requests and a variable frame length for containing data.

An FT1.2 reset frame looks like this:

FT1.2 Frame			
Byte 0	Byte 1	Byte 2	Byte 3
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Start character 0x10	Reset 0x40	Checksum 0x40	Stop character 0x16

To request a reset of internal counters and states, send an FT1.2 reset request:

10 40 40 16	RESET_REQ

Note: This is **not** a reboot of the KNX BAOS Module software. It only resets internal registers and states of the stack.

An FT1.2 data frame looks like this:

FT1.2 Header Frame																															
Byte 0								Byte 1								Byte 2								Byte 3								Byte 4							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Start character 0x68								Length (bytes 4-n)								Length (repeated)								Start character 0x68 (repeated)								FT 1.2 control field							

FT1.2 Information Frame																FT1.2 End Frame															
Byte 5								...								Byte n								Byte n+1								Byte n+2							
7	6	5	4	3	2	1	0									7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Information data (0 - n bytes)																								Checksum (bytes 4-n)								Stop character 0x16							

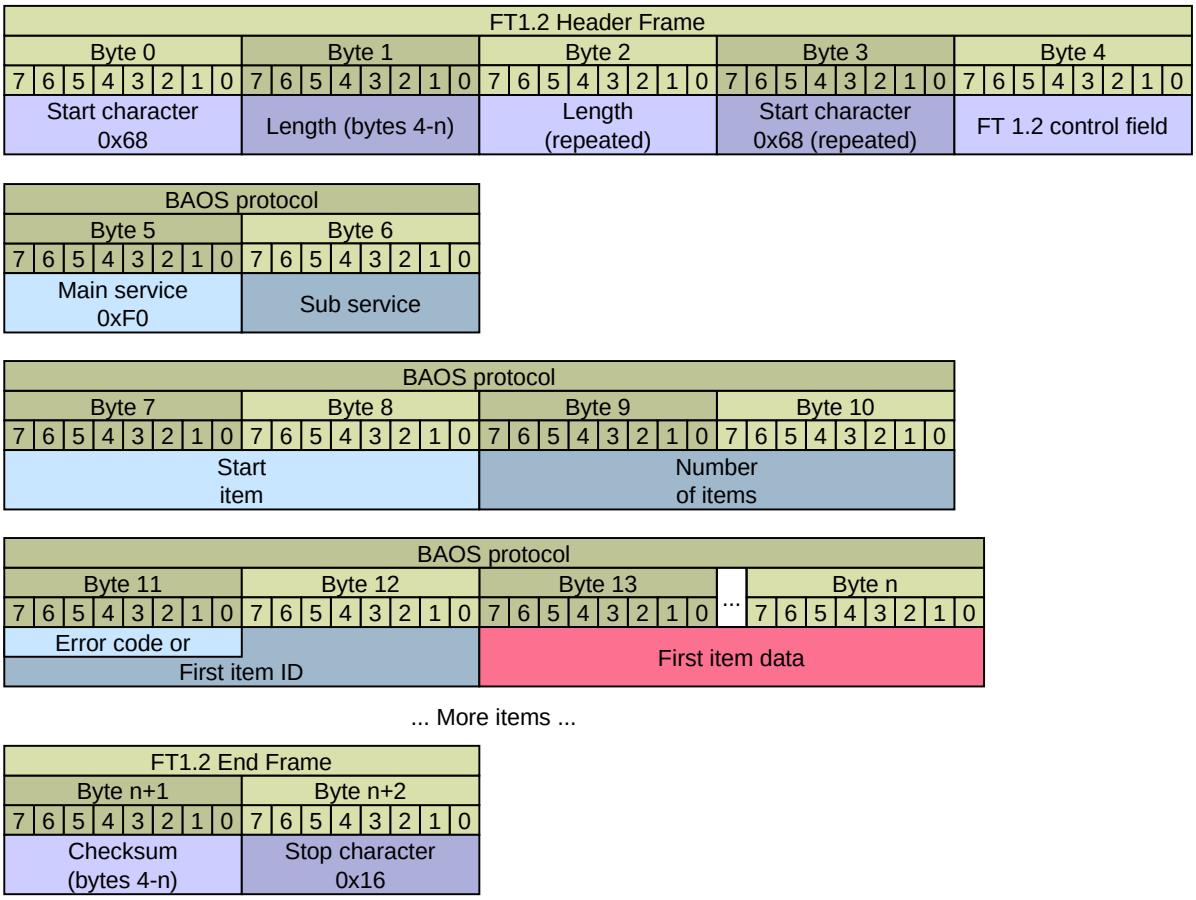
The **FT1.2 control field** has the following information:

- **Bit #7**: Direction:
  0 = Host (application) to KNX BAOS Module
  1 = KNX BAOS Module to Host (application)
- **Bit #6**: 1
- **Bit #5**: Frame count bit (toggled at each new message), must be 1 after a reset.
- **Bit #4 - 0**: 10011

15 BAOS Frame Embedded in an FT1.2 Frame

The BAOS protocol is encapsulated into the FT1.2 frame for data integrity.

An FT1.2 frame using the BAOS protocol looks as follows:



For the FT1.2 control field bits, see section “FT1.2 Frame Format”.

16 Common EMI Protocol

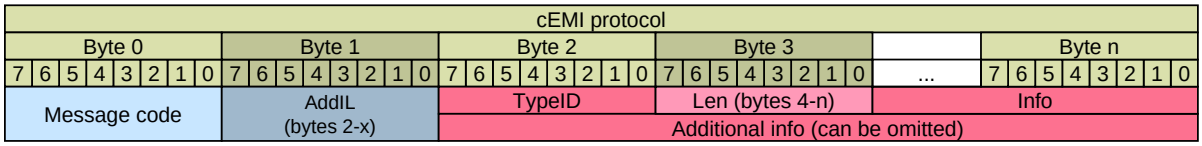
In this chapter, we learn how to access the link layer of the stack and its management via the cEMI protocol with examples.

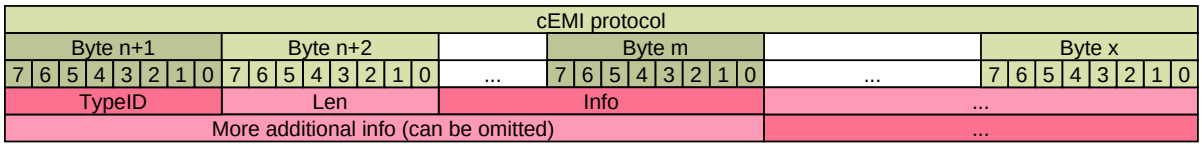
The message routing is implemented in the module stack. It offers the possibility to disable the BAOS protocol and to send/receive telegrams directly to/from a certain layer of the communication stack. In principal there are two reasonable short cuts:

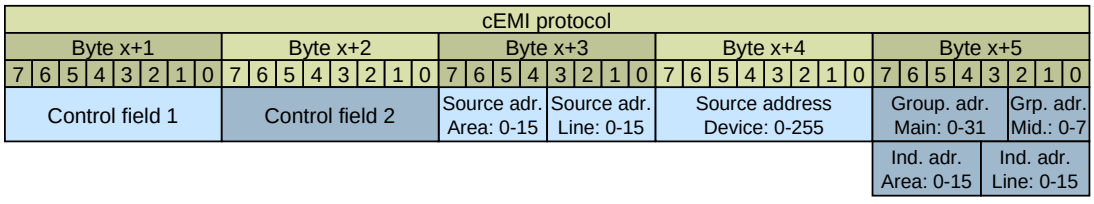
- **Telegram access via the link layer (LL).** This access also gives/takes telegrams like the network layer access. The group address filtering is also active. So group messages which are not intended for this device will be filtered out. To disable this filtering the length of the group address table must be set to zero.
  Broadcast messages can also be received as well as all directly addressed telegrams for this device.
- **Access to the management server.** This is not really an access to the KNX bus, because the management server is responsible for internal organization of the device. So the cEMI telegrams are not sent to the bus. They are handled internally.

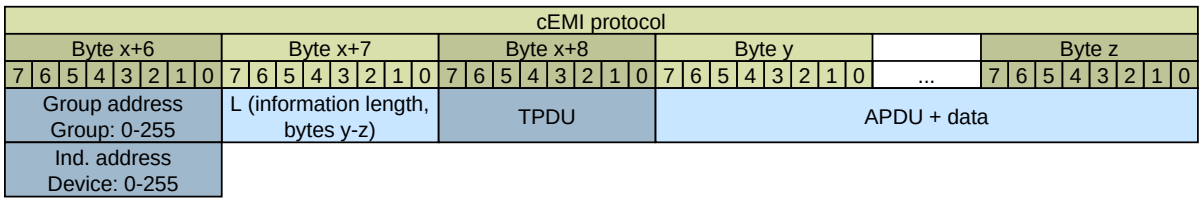
16.1 Link Layer Access

The cEMI telegram format for the *link layer* contains the KNX telegram as follows:









The **Message code** determines the kind of message:

- L_Data.req: 0x11
- L_Data.con: 0x2E
- L_Data.ind: 0x29
- L_Busmon.ind: 0x2B

The **AddIL** is the length of the additional info. This additional info can be omitted in which case the AddIL is 0. The additional info has the following structure:

- TypeID: Determines the type of additional information.
  - 0x00: reserved
  - 0x02: RF medium information (Len = 8)
  - 0x03: Bus-monitor status information (Len = 1)
  - 0x04: Relative time stamp (Len = 2)
- Len: length of current additional info.
- Info: Current additional info.

Control field 1:

- **Bit #7**: Frame type:
  - 1 = Standard frame
  - 0 = Extended frame
- **Bit #6**: unused, must be 0
- **Bit #5**: Repeat flag:
  - 0 = repeat frame if error
  - 1 = do not repeat
- **Bit #4**: System broadcast flag:
  - 0 = system broadcast
  - 1 = broadcast
- **Bit #3 - 2**: Priority, leave this to 11
- **Bit #1**: ACK request (= 1)
- **Bit #0**: Confirm flag (1 = error)

Control field 2:

- **Bit #7**: Destination address type:
  - 0 = Individual address
  - 1 = Group address
- **Bit #6 - 4**: Hop count
- **Bit #3 - 0**: EFF (extended frame format):
  - 0000 = Standard frame (long frame: APDU > 15 bytes)
  - 01xx = LTE frame

L: Information length of data (TPDU not included).

16.1.1 cEMI in the Application

Common EMI is used to send read and write requests directly to the KNX stack without the use of the BAOS protocol. Such a read or write request does not involve a data point.

To achieve this, the BAOS Module must be in the communication mode *Link Layer*. This is done by sending an M-PropWrite Request.

16.1.1.1 Set Module to Link Layer Mode

To set the BAOS Module into Link Layer mode, send the following telegram:

```
uint8_t aBuffer[] =
{
    0x08,                // Length of this array
    0xF6,                // cEMI service code: M_PropWrite.req
    0x00, 0x08,          // cEMI Object type (cEMI server)
    0x01,                // cEMI Object instance
    0x34,                // cEMI Property ID (Communication Mode)
    0x10,                // cEMI Number of elements (high nibble)
    0x01,                // cEMI Start index (incl. low nibble of prev. byte)
    0x00                 // cEMI Data: Link Layer Mode (=00)
};

KnxFt12_Write(aBuffer);  // Send M_PropWrite.req
```

16.1.1.2 Set Module back to BAOS Mode

Now the Module accepts own telegrams in the cEMI format. To use the BAOS protocol again, use this M_PropWrite.req again, but change the data to 0xF0:

```
uint8_t aBuffer[] =
{
    0x08,                // Length of this array
    0xF6,                // cEMI service code: M_PropWrite.req
    0x00, 0x08,          // cEMI Object type (cEMI server)
    0x01,                // cEMI Object instance
    0x34,                // cEMI Property ID (Communication Mode)
    0x10,                // cEMI Number of elements (high nibble)
    0x01,                // cEMI Start index (incl. low nibble of prev. byte)
    0xF0                 // cEMI Data: BAOS Mode (=F0)
};

KnxFt12_Write(aBuffer);  // Send M_PropWrite.req
```

16.1.1.3 Write Value to a Group Object

As long the module is in Link Layer mode, we can create own telegrams.

Writing a value to a data point normally causes a KNX telegram to be sent to a certain group object. This group object delivers the value to the actuator(s). It is also possible to write to such a group object without any data point involved.

As an example if a group object **3/3/3** is configured as absolute dimming value (1 byte) to a dimmer input, we can write a value 0 - 100 to change the light. Technically we can use values up to 255, but an absolute dimming value is defined from 0 - 100.

```
uint8_t aBuffer[] =
{
    0x0C,                   // Length of this array
    0x11,                   // cEMI service code: L_Data.req
    0x00,                   // cEMI Additional info length
    0x9C,                   // cEMI KNX control field 1
    0xE0,                   // cEMI KNX control field 2
    0x11, 0x20,             // cEMI source address 1.1.32
    0x1B, 0x03,             // cEMI group address 3/3/3
    0x00,                   // cEMI TPDU
    0x00,                   // cEMI APDU + APCI
    0x80,                   // cEMI APCI
    0x64                    // cEMI data byte (= 100)
};

KnxFt12_Write(aBuffer);    // Send L_Data.req
```

This writes the value 100 to the group object **3/3/3**.

16.1.1.4 Read Value from a Group Object

Reading a value from a group object, the following example can do this:

```
uint8_t aBuffer[] =
{
    0x0B,                   // Length of this array
    0x11,                   // cEMI service code: L_Data.req
    0x00,                   // cEMI Additional info length
    0x9C,                   // cEMI KNX control field 1
    0xE0,                   // cEMI KNX control field 2
    0x13, 0x03,             // cEMI source address 1.3.3
    0x1A, 0x03,             // cEMI group address 3/2/3
    0x00,                   // cEMI TPDU
    0x00,                   // cEMI APDU + APCI
    0x00                    // cEMI APCI
};

KnxFt12_Write(aBuffer);    // L_Data.req
```

The response can be handled in **KnxBaos_Process()**, File: **KnxBaos.c**:

```
void KnxBaos_Process(void)
{
    bool_t bAccept;                     // Accept the current telegram?

    // First proceed the receive job
    if(KnxFt12_Read(m_pBaosRcvBuf))     // Poll telegrams from BAOS
    {
        bAccept = FALSE;                // Initialize telegram to be ignored

        [...]                           // Check telegram service code
        [...]                           // Set bAccept = TRUE, if accepted
```

```
        if(bAccept == TRUE)             // Have we accept the telegram?
        {
            [...]                       // Handle telegram data
        }
        else
        {
            // Here we can handle all received telegrams in
            // m_pBaosRcvBuf[]. m_pBaosRcvBuf[0] stores the length,
            // the following bytes are cEMI telegrams.
        }
    }
    [...]
}
```

The buffer **m_pBaosRcvBuf[]** will contain this response (the length byte at index 0 is not part of cEMI):

```
05 AB 01 40 00 15                                       PC_GetValue.con
```

Note: More information about cEMI telegrams can be found in the **KNX System Specifications/03_06_03 EMI_IMI** available at the KNX Specifications page.

16.1.2 Send Group Telegrams using Net'n Node

Start Net'n Node, use **View/Access Port Configuration** and **Create new...** port access and use **KNX FT1.2 Serial**. If it successfully **Open**s the port, choose **Select Send KNX Group Value Write**. In the dialog enter the values (see examples below) and **Send** it. A response will be displayed in the Telegram View.

- DPT 01 - Binary - 1 bit
  - Group Address: 4/3/2
  - Data: TRUE

The telegrams send and received look like these:

```
PC:         11 00 BC E0 00 00 23 02 01 00 81    L_Data.req
Module:     2E 00 BC E0 FF FF 23 02 01 00 81    L_Data.con
```

Net'n Node can also monitor the KNX bus. For this, additionally open a KNX bus interface (e. g. USB TP). The telegram we caused on the bus looks like this:

```
KNX-Bus:    29 00 BC E0 FF FF 23 02 01 00 81    L_Data.ind
```

Here we sent a telegram to group address 4/3/2 containing a 1 bit value of TRUE (1).

- DPT 05 - 8-Bit Unsigned Value - 1 byte
  - Group Address: 8/4/3
  - Data: 170

The telegrams send and received look like these:

```
PC:         11 00 BC E0 00 00 44 03 02 00 80 AA   L_Data.req
```

```
KNX-Bus:     29 00 BC E0 FF FF 44 03 02 00 80 AA    L_Data.ind
Module:      2E 00 BC E0 FF FF 44 03 02 00 80 AA    L_Data.con
```

Here we sent a telegram to group address 8/4/3 containing a 1 byte value of 0xAA (170).

- DPT 08 - 2-Octet Signed Value - 2 bytes
  - Group Address: 15/7/254
  - Data: -2

The telegrams send and received look like these:

```
PC:          11 00 BC E0 00 00 7F FE 03 00 80 FF FE    L_Data.req
KNX-Bus:     29 00 BC E0 FF FF 7F FE 03 00 80 FF FE    L_Data.ind
Module:      2E 00 BC E0 FF FF 7F FE 03 00 80 FF FE    L_Data.con
```

Here we sent a telegram to group address 15/7/254 containing a 2 byte value of 0xFFFE (-2).

If we want to read a value, we have to use a group address which is connected to a real data point of a real KNX device. For this, select **Send KNX Group Value Read**. In the dialog enter the group address (example 1/2/1) and select the **Send** button. A response will be displayed in the Telegram View.

```
PC:          11 00 9C E0 00 00 0A 01 01 00 00    L_Data.req
KNX-Bus:     29 00 BC E0 FF FF 0A 01 01 00 00    L_Data.ind
KNX-Bus:     29 00 BC E0 11 01 0A 01 01 00 41    L_Data.ind
Module:      2E 00 BC E0 FF FF 0A 01 01 00 00    L_Data.con
Module:      29 00 BC E0 11 01 0A 01 01 00 41    L_Data.ind
```

Why are there two telegrams on the KNX bus and two from the module?

The first telegram from the PC is an L_Data request to read the value in group address 1/2/1. This request is sent to the bus (first telegram on the KNX bus). It origins from our BAOS module (hence the individual address 15.15.255 (0xFFFF). The second telegram on the bus is the answer. There is a KNX device (1.1.1) whose data point has the 1 bit value 1 (coded into the last byte 0x41). The two last telegrams from the BAOS module are the corresponding telegrams from the KNX bus. The first one is the confirmation of our request. The second one is the indication of the answer to our request. This answer contains the value we wanted to know.

If we query a group address (example 8/2/1) which is not connected to any device, the telegrams would look like these:

```
PC:          11 00 9C E0 00 00 42 01 01 00 00    L_Data.req
KNX-Bus:     29 00 BC E0 FF FF 42 01 01 00 00    L_Data.ind
Module:      2E 00 BC E0 FF FF 42 01 01 00 00    L_Data.con
```

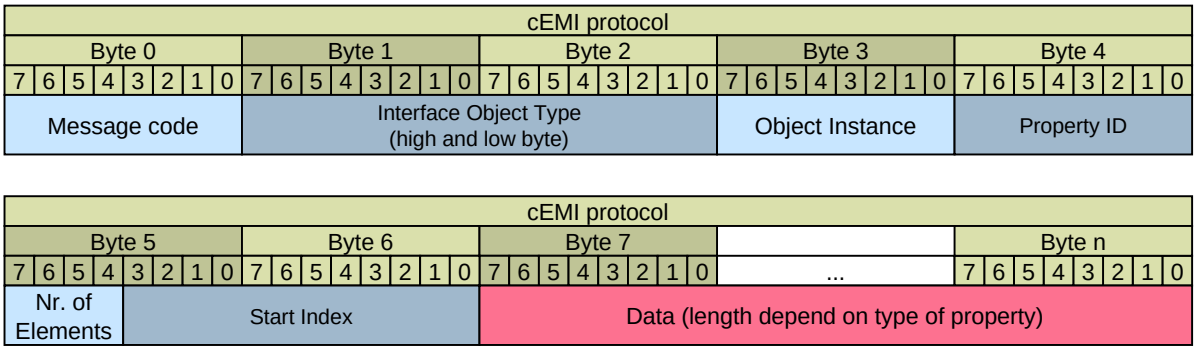
We see the two answer telegrams are missing, since there is no device which can respond our request.

16.2 Management Server Access

Properties are a quite central information and management part of KNX devices. They hold lots of information about it and some of them can be changed to make an effect. They can be accessed via the KNX bus but it is also possible to access them via the serial (UART) port using cEMI.

The host interface of the KNX BAOS Modules 83x allows access to the KNX Properties via the so-called M_Prop Services. M_Prop Services are part of the KNX specification for common EMI servers. All property elements which are present in the KNX Module are accessible via the bus as well as via the host interface. Due to historical reasons the protocols for the two ways are different. Mainly the addressing of the interface objects is different. An interface objects is a container for properties. The first object in a KNX device is always the so-called device object. It is addressed by the object index which is 0, because it is the first one. And there is an object instance number which starts at 1. Each property is addressed as an array with number of elements (4 bit) and a start index (12 bit).

The cEMI telegram format for the *management server* contains the service as follows:



The **Message code** determines the kind of message:

- M_PropRead.req: 0xFC
- M_PropRead.con: 0xFB
- M_PropWrite.req: 0xF6
- M_PropWrite.con: 0xF5
- M_Reset.req: 0xF1
- M_Reset.ind: 0xF0

Interface Object Type, **Object Instance** and **Property ID** determine the access to the management service.

Number of Elements and **Start Index** determine the structure and size of the **Data**.

16.2.1 Property Access Examples with Net'n Node

Example: Read the property Programming Mode.

Start Net'n Node, use **View/Access Port Configuration** and **Create new...** port access and use **KNX FT1.2 Serial**. If it successfully **Open**s the port, choose **Send KNX Local services M_PropRead.req**. In the dialog enter the values (see below) and **Send** it. A response will be displayed in the Telegram View.

- M_Prop**Read**.Req/Con
  - InterfaceObjectType: 0 (Device Object)
  - ObjectInstance: 1
  - PropertyId: 54 (Programming Mode)
  - NrOfElem: 1
  - StartIndex: 1

The telegrams send and received look like these:

```
PC:          FC 00 00 01 36 10 01         M_PropRead.Req
Module:      FB 00 00 01 36 10 01 00      M_PropRead.Con
```

The answer can be interpreted by Net'n Node. Just click on the second telegram to see it. The answer shows the data = 0 which means the programming mode is off.

To send this in the application, we have to add the length byte:

```
uint8_t aBuffer[] =
{
    0x07,             // Length of this array
    0xFC,             // cEMI service code: M_PropRead.req
    0x00, 0x00,       // cEMI Interface Object Type 0
    0x01,             // cEMI Object Instance 1
    0x36,             // cEMI PropertyId: 54 (Programming Mode)
    0x10,             // cEMI Number of Elements 1, Start Index 1 (high part)
    0x01,             // cEMI Start Index 1 (low part)
};

KnxFt12_Write(aBuffer);   // M_PropRead.req
```

Example: Write the property. We will activate the programming mode this way.

- M_Prop**Write**.Req/Con
  - InterfaceObjectType: 0 (Device Object)
  - ObjectInstance: 1
  - PropertyId: 54 (Programming Mode)
  - NrOfElem: 1
  - StartIndex: 1
  - Data/Length: 01/1

The telegrams send and received look like these:

```
PC:          F6 00 00 01 36 10 01 01      M_PropWrite.Req
Module:      F5 00 00 01 36 10 01         M_PropWrite.Con
```

The red LED must be on now.

Example: Read the serial number of the module. The serial number is located in the device object (type = 0x0000, instance = 0x01) with the property identifier (PID) PID_SerialNumber = 0x0B).

M_PropRead and M_PropResp Services in cEMI Format:

```
PC:          FC 00 00 01 0B 10 01                        M_PropRead.Req
Module:      FB 00 00 01 0B 10 01 00 C5 00 00 00 00      M_PropRead.Con
                                 ^^^^^^^^^^^^^^^^^
                                   serial number
```

Some more PIDs of interest for BAOS applications (device object = 0, instance = 1):

PID name	PID#	Value type
PID_SERIAL_NUMBER	11	PT_GENERIC06
PID_MANUCATURER_ID	12	PT_UINT
PID_ORDER_INFO	15	PT_GENERIC10
PID_VERSION	25	PT_VERSION
PID_DOWNLOAD_COUNTER	30	PT_UINT
PID_PROGMODE	54	PT_BITSET8
PID_SUBNET_ADDR	57	PT_UCHAR
PID_DEVICE_ADDR	58	PT_UCHAR
PID_HARDWARE_TYPE	78	PT_GENERIC06

Example: Reset some states of the BAOS Module.

- M_Reset.Req/Ind

The telegrams send and received look like these:

```
PC:          F1      M_Reset.Req
Module:      F0      M_Reset.Ind
```

The module resets some counters and states (e. g. clears programming mode) and sends an indication to confirm the reset.

For an overview about important properties, see section “Important Properties”.

16.3 cEMI Frame Embedded in an FT1.2 Frame

The cEMI protocol is also encapsulated into the FT1.2 frame for data integrity. Net'n Node encapsulates these cEMI telegrams automatically into FT1.2 frames. Keep this in mind if you want to communicate with your own micro controller to the BAOS module with cEMI. The demo code includes routines for this.

An FT1.2 frame using the cEMI protocol looks as follows:

FT1.2 Header Frame				
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Start character 0x68	Length (bytes 4-z)	Length (repeated)	Start character 0x68 (repeated)	FT 1.2 control field

cEMI protocol					
Byte 5	Byte 6	Byte 7	Byte 8		Byte n
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	...	7 6 5 4 3 2 1 0
Message code	AddIL (bytes 7-x)	TypeID	Len (bytes 9-n)	Info	
		Additional info (can be omitted)			

cEMI protocol					
Byte n+1	Byte n+2		Byte m		Byte x
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	...	7 6 5 4 3 2 1 0	...	7 6 5 4 3 2 1 0
TypeID	Len	Info		...	
More additional info (can be omitted)				...	

cEMI protocol					
Byte x+1	Byte x+2	Byte x+3	Byte x+4	Byte x+5	
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	
Control field 1	Control field 2	Source adr. Area: 0-15	Source address Device: 0-255	Group. adr. Main: 0-31	Grp. adr. Mid.: 0-7
		Source adr. Line: 0-15		Ind. adr. Area: 0-15	Ind. adr. Line: 0-15

cEMI protocol					
Byte x+6	Byte x+7	Byte x+8	Byte y		Byte z
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	...	7 6 5 4 3 2 1 0
Group address Group: 0-255	L (information length, bytes y-z)	TPDU	APDU + data		
Ind. address Device: 0-255					

FT1.2 End Frame	
Byte z+1	Byte z+2
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Checksum (bytes 4-z)	Stop character 0x16

The FT1.2 Control Field bits, see section “FT1.2 Frame Format”.

The Message Code, TypeID, Control field 1 and 2, see chapter “Common EMI Protocol”.

For more information about the FT1.2 protocol see **KnxBAOS_Protocol_v2.pdf**, Appendix E.

The example of reading the serial number uses these cEMI telegrams, which are embedded into an FT1.2 frame:

```
PC:     68 08 08 68 53 FC 00 00 01 0B 10 01 6C 16                     M_PropRead.Req
Module: 68 0E 0E 68 D3 FB 00 00 01 0B 10 01 00 C5 00 00 00 00 B0 16   M_PropRead.Con
                                           ^^^^^^^^^^^^^^^^^^
                                             serial number
```

17 Individual ETS Entries

This chapter shows how to create an individual ETS entry as an example.

To create individual entries, the following is required:

- The KNX Manufacturer Tool (MT) and a valid license.
- The KNX Engineering Tool Software (ETS) and a valid license.
- This Weinzierl KNX BAOS Starter Kit.

The MT is used to define and create own Object Lists, Parameters and group objects for ETS. The result of this tool is a test project containing the product.

This test project can be used while development. After development it can be registered and a product is derived from this project.

To achieve this, the following is required:

- KNX membership
- ISO 9001 certification
- Certification of the product

Facts good to know about the KNX BAOS Module for working with MT:

- **Profile Class**: System B
- **Interface**: Serial asynchronous
- **Protocol**: FT1.2 based
- **Address Table**: starting at address 0x1000
- **Association Table**: starting at address 0x17D2
- **ComObjects Table**: starting at address 0x2774, 1000 pre-defined
- **Parameters**: starting at address 0x33AB, 250 parameters each 1 byte.
  (MCB 4 starts at 0x2F46 and is segmented in SUB_MCBs. The parameters start at SUB_MCB 5, which has an offset of 0x465. So their address start at 0x2F46 + 0x465 = 0x33AB. See also section “Binary Import”.)

GO number	DPT	Length
0	not available	not available
1 – 1000	01, 02, ... 18	1 bit ... 14 bytes

Note: The terms *Group Object* and *Communication Object* are synonyms. Communication Object is used in Manufacturer Tool, ETS and other tools. Group Object is the only term used in the rest of the KNX specifications and is therefore considered as the only correct one. Both terms will however be used here because it is here were practice and theory meet. Communication Object will only be used when absolutely necessary, e. g. in the context of Manufacturer Tool.

17.1 Example for Creating an Individual ETS Database

To create a database, start MT and do the following:

17.1.1 Project

Create a new project by selecting menu **File/New/Project...**, select **KNX MT Project** and browse for a location to store the project, enter a name (e. g. **My_KNX_BAOS**) and hit **OK**.

In the dialog window it is recommended to select Target ETS Version **ETS 4** (knxprod) since a lot of users still use ETS 4. KNX BAOS RF 840 needs **ETS 5**, since RF is not fully supported in ETS 4. For manufacturer select either your own name or **M-00FA KNX Association** and hit **OK**.

The KNX BAOS Modules have the manufacturer code 00C5, which is the code for Weinzierl Engineering GmbH. This code must match to the one stored in the device's ETS database. If we create a device with MT using a different code (here 00FA) we must also change this code in the KNX BAOS Module. ETS refuses to download the configuration if the manufacturer codes mismatch. See chapter “How to Change Production Parameters” for changing the manufacturer code.

17.1.2 Create New Application

Now it is necessary to create an application. This describes the configuration of the KNX BAOS Module.

Click the project name in the Solution Explorer (**My_KNX_BAOS**) and use the menu **Project/Add New Item...**, select **Application Program**, edit its name (e. g. **My_KNX_BAOS_App.mtxml**) and hit **Add**. Enter values in the dialog window:

- Application Number: 0001h
- Application Version: 01h
- Name: My_KNX_BAOS_App
- Mask Version: [07B0h] System B

17.1.3 Create New Hardware

The product is not only a software application. It is also a hardware which we must add, too.

Click the project name in the Solution Explorer (**My_KNX_BAOS**) and use the menu **Project/Add New Item...**, select **Hardware**, edit its name (e. g. **My_KNX_BAOS_Hw.mtxml**) and hit **Add**. Enter values in the dialog window:

- Serial Number: SN2016-02-16
- Version Number: 0001h
- Hardware Name: My_KNX_BAOS_Hw
- Order Number: ON2016-02-16
- Product Name: My_KNX_BAOS_Product

and hit **OK**.

Link the hardware to the application: Open the hardware by double clicking its name in the Solution Explorer. In the newly opened tab select the hardware name (the name containing the serial number). Use right mouse button menu **Add new Hardware2Program** and select the application program **[0001 10] My_KNX_BAOS_App**. Choose the correct medium type (**TP** or **RF**) and hit **OK**.

For TP, enter the bus current by selecting the **Hardware**. In the main table enter **3** (for 3 ampere) at the **Bus Current** column.

17.1.4 Binary Import

Unpack the archive

Weinzierl_8xx_KNX_BAOS_ETS_Projects_for_Demo.zip

Edit the application by selecting the application tab and then its name (e. g. **My_KNX_BAOS_App**). Use right mouse button menu **Import binary data**, browse to the s37-file which is in the archive.

Weinzierl_8xx_KNX_BAOS_ETS_Projects_for_Demo/
ETS_Project_using_individual_ETS_entry/
MT_Import_Files/
APP_BAOS_8xx_255_Parameter_Bytes.s37,

open it and hit **OK**.

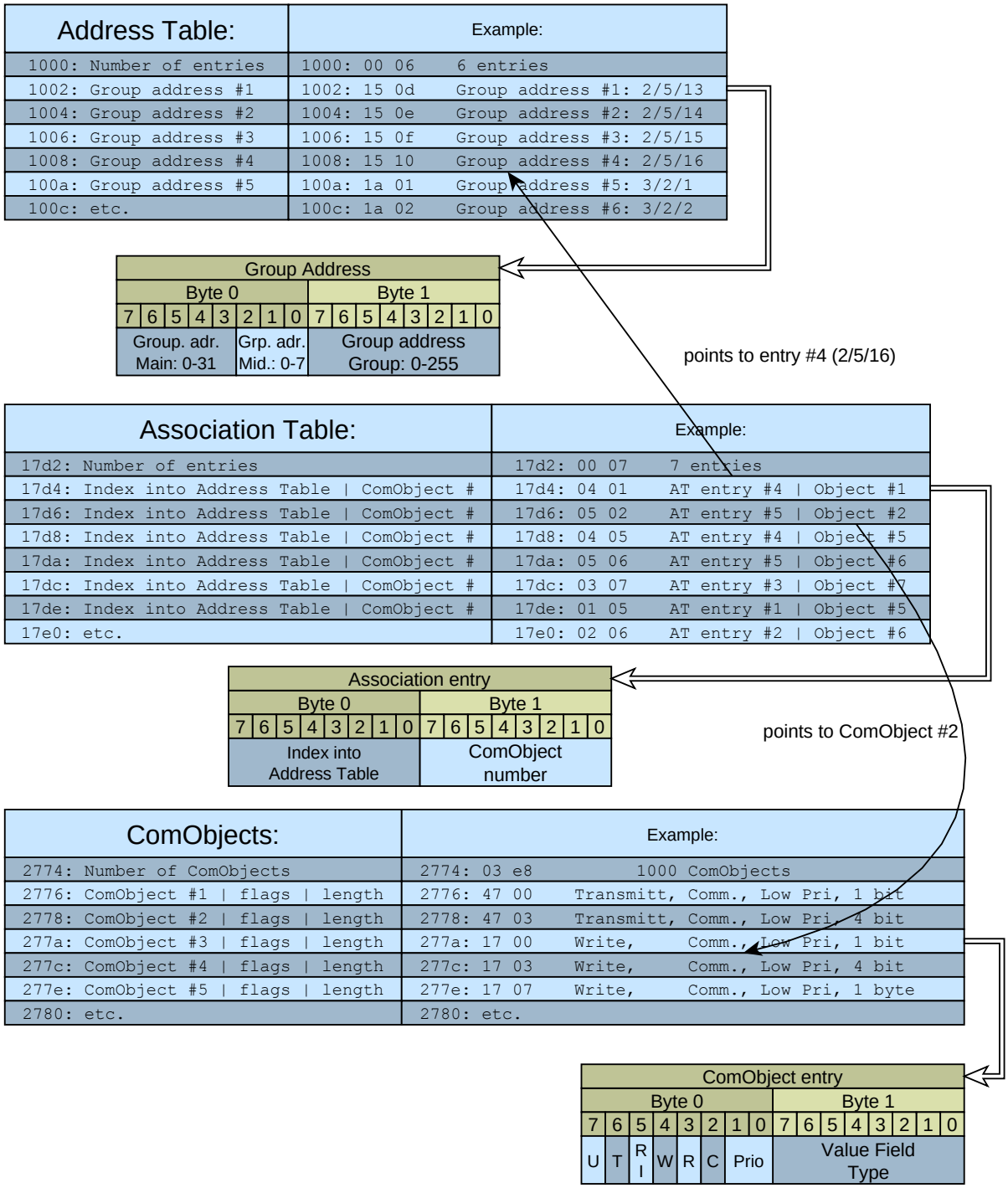
Note: s37 file with 1000 parameter bytes: Don't use the **APP_BAOS_8xx_1000_Parameter_Bytes.s37** file. It contains 1000 parameter bytes, which are used in the example below in section “Example to create more Parameter Bytes”.

This adds a code segment to the application. Check this by selecting the application window and unfold the tree until the static entry shows its children. There is a **Load Procedures** entry. Click it and see a small table. It contains three Merge-Ids. Merge-Id 02h contains MCB 4.

Index	Action	Parameters		AppliesTo
1	RelSegment	lsm=#04h, size=055Fh, mode=01h,		full
2	RelSegment	lsm=#04h, size=055Fh, mode=00h,		par
3	WriteProp	obj=#04h, prop=1Bh, start=0001h,	data={00 00 00 14 00 32 00 00 00 00}	auto
4	WriteProp	obj=#04h, prop=1Bh, start=0002h,	data={00 00 00 47 00 32 00 00 00 00}	auto
5	WriteProp	obj=#04h, prop=1Bh, start=0003h,	data={00 00 00 21 01 33 00 00 00 00}	auto
6	WriteProp	obj=#04h, prop=1Bh, start=0004h,	data={00 00 03 E8 00 32 00 00 00 00}	auto
7	WriteProp	obj=#04h, prop=1Bh, start=0005h,	data={00 00 00 01 00 32 00 00 00 00}	auto
8	WriteProp	obj=#04h, prop=1Bh, start=0006h,	data={00 00 00 FA 00 32 00 00 00 00}	auto

obj=#04 denotes MCB 4. This MCB is segmented in SUB_MCBs, which are denoted by **start=000xh**. Every SUB_MCB has a certain length. This length is defined in the first 4 bytes of **data={xx xx xx xx...}**.

- SUB_MCB 1 "Application Header" starts at offset 0x00000000 and has a length of 0x00000014.
- SUB_MCB 2 "BAOS Header" starts at offset 0x00000014 and has a length of 0x00000047.
- SUB_MCB 3 "BAOS Internal Data" starts at offset 0x0000005B and has a length of 0x00000021.
- SUB_MCB 4 "Data point Types" starts at offset 0x0000007C and has a length of 0x000003E8.
- SUB_MCB 5 "Data point Descriptions" starts at offset 0x00000464 and has a length of 0x00000001. Is not used to store the descriptions into the KNX BAOS Module, since it would take too much time for downloading them and it would consume too much memory. So the descriptions remain in the ETS.
- SUB_MCB 6 "Parameter Bytes" starts at offset 0x00000465 and has a length of 0x000000FA. This offset we need later. So the resulting address for the first Parameter Byte is 0x2F46 + 0x0465 = 0x33AB.



The **ComObject flags** (byte 0) and **ComObject Value Field Type** (byte 1) coding, see section "Group Object Table (MCB 3)".

The **ComObject RAM flags** are for run time use. Each ComObject has one byte. This byte indicates whether everything is OK, an error occurred or a transmission is in progress, etc.

Parameters:	Example:
33ab: Parameter #1	33ab: 02
33ac: Parameter #2	33ac: 02
33ad: Parameter #3	33ad: 00
33ae: etc.	33ae: etc.

ComObject RAM Flags:	Example:
1b000: DataPointType #1	1b000: 00
1b001: DataPointType #2	1b001: 02
1b002: DataPointType #3	1b002: 08
1b003: etc.	1b003: etc.

The **Parameters** are application specific. Each parameter is one byte. They can be changed via ETS. The application can use the BAOS protocol to read the values.

In case of the KNX BAOS Module the following parameters are used:

Address	Usage
0x33AB	Sensor: 0 = disabled, 1 = switch, 2 = dimmer, 3 = shutter
0x33AC	Actuator: 0 = disabled, 1 = switch, 2 = dimmer
0x33AD - 0x34A3	Not used

The **ComObject RAM Flags** determine the data point current state. The values are as follows:

- **Bit #7**: Used for return value:
  - 0 = OK
  - 1 = Error
- **Bit #6 - 5**: unused
- **Bit #4**: Value valid:
  - 0 = Validity of value is unknown
  - 1 = Object has already been received
- **Bit #3**: External update:
  - 0 = No external update
  - 1 = Object has been updated via telegram
- **Bit #2**: Read request:
  - 0 = No read request
  - 1 = Object wants to send a read request
- **Bit #1 - 0**: Status:
  - 00 = OK
  - 01 = Error
  - 10 = Transmit in progress
  - 11 = Transmit requested

17.1.5 Create Address and Association Table

Edit the application by selecting the application tab and then its name (e. g. **My_KNX_BAOS_App**). Use right mouse button menu **Add new/Address Table** and enter the Properties at the right side:

- Max Entries: 03e8h

This creates place for up to 1000 (= 0x3e8) table entries.

Use right mouse button menu at the application name again **Add new/Association Table** and enter the Properties at the right side:

- Max Entries: 03e8h

17.1.6 Create Visible Data Points

The KNX BAOS Module has 1000 ComObjects.

To create a light switch which can handle one LED (switching it on/off and dimming it), we declare the following ComObjects:

Object #1 is switch output, connected to object #3 which is LED switching input.

Object #2 is dimming output, connected to object #4 which is LED dimming input.

So object #1 and #3 are 1 bit (DPT 1.001) and object #2 and #4 are 4 bit (DPT 3.007).

Define wanted data points.

Go to the application by selecting the application tab and then its name (e. g. **My_KNX_BAOS_App**). Use right mouse button menu **Add new/ComObject** and create the following ComObjects:

Function Text	Internal Name	Number	Object Flags	Object Size	Text	Create ComObjectRef
on/off	Obj1	0001h	Output	1 Bit	Sensor switch	True
brighter/darker	Obj2	0002h	Output	4 Bit	Sensor dim	True
on/off	Obj3	0003h	Input	1 Bit	Actuator switch	True
brighter/darker	Obj4	0004h	Input	4 Bit	Actuator dim	True
absolute	Obj5	0005h	Input	1 Byte	Actuator dim	True

Define types of wanted data points.

Go to the **ComObjects** table and define the data point types of every ComObject as follows:

ComObject Number	Data point Type
0001h	[1.1] DPT_Switch
0002h	[3.7] DPT_Control_Dimming
0003h	[1.1] DPT_Switch
0004h	[3.7] DPT_Control_Dimming
0005h	[5.1] DPT_Scaling

17.1.7 Button for Switching and Dimming

Add parameter types for button.

To add parameter types for our wanted data points, select Static in the left tree and choose right mouse button menu **Add new/ParameterTypeRestriction**. Enter these values in the dialog window:

- Internal Name: ButtonFunction_t

and hit **OK**. A variable names "ButtonFunction_t" of enum type exists now in **Parameter Types**. We must add now the possible values for this type. Select **ButtonFunction_t** in the tree and choose right mouse button menu **Add new Enumeration Value**. Enter these values in the dialog window

- Text: Disabled
- Value: 0000h

and hit **OK**. Add two more values

- Text: Switch
- Value: 0001h

hit **OK** and add the last value

- Text: Dimmer
- Value: 0002h

and hit **OK**.

Add memory parameters.

If the left tree shows an entry **Parameters** or **ParameterRefs**, select **Parameters** and delete all entries in the table. Do the same with **ParameterRefs**.

To add a memory parameter (i. e. parameter byte), select **Static** in the left tree and choose right mouse button menu **Add new/Memory Parameter**. Enter these values in the dialog window:

- Access: ReadWrite
- Bit Offset: 00h
- Code Segment: [Rel#04h]
- Internal Name: Button
- Offset: 0465h
- Parameter Type: ButtonFunction_t
- Text: Function
- Create ParameterRef: True

and hit **OK**.

Add two more memory parameters for each button.

To add a memory parameter, select **Static** in the left tree and choose right mouse button menu **Add new/Memory Parameter**. Enter these values in the dialog window:

- Access: None
- Bit Offset: 00h
- Code Segment: [Rel#04h]

- Internal Name: DPT-Value_GO1
- Offset: 0467h
- Parameter Type: ButtonFunction_t
- Text: DPT-Value_GO1
- Create ParameterRef: True

and hit **OK**. Create the second parameter with these values

- Access: None
- Bit Offset: 00h
- Code Segment: [Rel#04h]
- Internal Name: DPT-Value_GO2
- Offset: 0468h
- Parameter Type: ButtonFunction_t
- Text: DPT-Value_GO2
- Create ParameterRef: True

and hit **OK**.

Unfold the dynamic part. **Channel 1** should already exist. If not create the channel by selecting Dynamic and choose right mouse button menu **Add new Channel**. Enter **Channel 1** and number **1**.

If **Channel 1** already contains a ParameterBlock, delete it. Select **Channel 1** and choose right mouse button menu **Add new/ParameterBlock**. Enter these values in the dialog window

- Name: PageButton
- Text: Sensor

and hit **OK**.

Add a parameter to the Parameter Block. Select **PageButton** and choose right mouse button menu **Add new/ParameterRefRef**. Select **Button** in the dialog window and hit **OK**.

Add a choice to the Parameter Block. Select **PageButton** and choose right mouse button menu **Add new/Choose**. Enter these values in the dialog window

- Parameter: [0001h] Button
- Create all Whens (enum only): False
- Create Default when: False

and hit **OK**.

Select the newly created **Button** choice and choose right mouse button menu **Add new When**. Enter these values in the dialog window

- Default: False
- Test: [0000h] Disabled

and hit **OK**. Add two more tests **[0001h] Switch** and **[0002h] Dimmer**.

Add some parameters and ComObjects to the tests. The test (**Disabled**) does not contain anything, so we skip to the next. Select (**Switch**) and choose right mouse button menu

Menu	Item to select
Add new/ComObjectRefRef	**[0001h 0001h] Obj1-on/off**
Add new/ParameterRefRef	**[0002h] DPT-Value_GO1**

Go to **Static/ComObjectRefs** and locate **Obj1** (Sensor switch, on/off) there. Select it and choose right mouse button menu **Copy**. Choose right mouse button menu **Paste**. This creates a copy of Obj1. It is located at the end of the list.

Go to **Static/ParameterRefs** and copy/paste **DPT-Value_GO1** in the same way.

In the Dynamic section, select (**Dimmer**) and choose right mouse button menu

Menu	Item to select
Add new/ComObjectRefRef	**[0001h 0006h] Obj1-on/off**
Add new/ParameterRefRef	**[0004h] DPT-Value_GO1**
Add new/ComObjectRefRef	**[0002h 0002h] Obj2-brighter/darker**
Add new/ParameterRefRef	**[0003h] DPT-Value_GO2**

17.1.8 Light for Switching and Dimming

Add parameter types for light.

To add parameter types for our wanted data points, select Static in the left tree and choose right mouse button menu **Add new/ParameterTypeRestriction**. Enter these values in the dialog window:

- Internal Name: LightFunction_t

and hit **OK**. A variable names "LightFunction_t" of enum type exists now in **Parameter Types**. We must add now the possible values for this type. Select **LightFunction_t** in the tree and choose right mouse button menu **Add new Enumeration Value**. Enter these values in the dialog window

- Text: Disabled
- Value: 0000h

and hit **OK**. Add two more values

- Text: Switch
- Value: 0001h

hit **OK** and add the last value

- Text: Dimmer

- Value: 0002h

and hit **OK**.

Add memory parameters.

To add a memory parameter, select **Static** in the left tree and choose right mouse button menu **Add new/Memory Parameter**. Enter these values in the dialog window:

- Access: ReadWrite
- Bit Offset: 00h
- Code Segment: [Rel#04h]
- Internal Name: Light
- Offset: 0466h
- Parameter Type: LightFunction_t
- Text: Function
- Create ParameterRef: True

and hit **OK**.

Add two memory parameters for each Light.

To add a memory parameter, select **Static** in the left tree and choose right mouse button menu **Add new/MemoryParameter**. Enter these values in the dialog window:

- Access: None
- Bit Offset: 00h
- Code Segment: [Rel#04h]
- Internal Name: DPT-Value_GO3
- Offset: 0469h
- Parameter Type: LightFunction_t
- Text: DPT-Value_GO3
- Create ParameterRef: True

and hit **OK**. Create the second parameter with these values

- Access: None
- Bit Offset: 00h
- Code Segment: [Rel#04h]
- Internal Name: DPT-Value_GO4
- Offset: 046Ah
- Parameter Type: LightFunction_t
- Text: DPT-Value_GO4
- Create ParameterRef: True

and hit **OK**. Create the third parameter with these values

- Access: None

- Bit Offset: 00h
- Code Segment: [Rel#04h]
- Internal Name: DPT-Value_GO5
- Offset: 046Bh
- Parameter Type: LightFunction_t
- Text: DPT-Value_GO5
- Create ParameterRef: True

and hit **OK**.

Create channel 2 by selecting **Dynamic** and choose right mouse button menu **Add new Channel**. Enter **Channel 2** and number **2**.

Select **Channel 2** and choose right mouse button menu **Add new/ParameterBlock**. Enter these values in the dialog window

- Name: PageLight
- Text: Actuator

and hit **OK**.

Add a parameter to the Parameter Block. Select **PageLight** and choose right mouse button menu **Add new/ParameterRefRef**. Select **Light** in the dialog window and hit **OK**.

Add a choice to the Parameter Block. Select **PageLight** and choose right mouse button menu **Add new/Choose**. Enter these values in the dialog window

- Parameter: [0005h] Light
- Create all Whens (enum only): False
- Create Default when: False

and hit **OK**.

Select the newly created **Light** choice and choose right mouse button menu **Add new When**. Enter these values in the dialog window

- Default: False
- Test: [0000h] Disabled

and hit **OK**. Add two more tests **[0001h] Switch** and **[0002h] Dimmer**.

Add some parameters and ComObjects to the tests. The test (**Disabled**) does not contain anything, so we skip to the next. Select (**Switch**) and choose right mouse button menu

Menu	Item to select
Add new/ComObjectRefRef	**[0003h 0003h] Obj3-on/off**
Add new/ParameterRefRef	**[0006h] DPT-Value_GO3**

Go to **Static/ComObjectRefs** and locate **Obj3** (Switch, on/off) there. Select it and choose right mouse button menu **Copy**. Choose right mouse button menu **Paste**. This creates a copy of Obj3. It is located at the end of the list.

Go to **Static/ParameterRefs** and copy/paste **DPT-Value_GO3** in the same way.

Select (**Dimmer**) and choose right mouse button menu

Menu	Item to select
Add new/ComObjectRefRef	**[0003h 0007h] Obj3-on/off**
Add new/ParameterRefRef	**[0009h] DPT-Value_GO3**
Add new/ComObjectRefRef	**[0004h 0004h] Obj4-brighter/darker**
Add new/ParameterRefRef	**[0007h] DPT-Value_GO4**
Add new/ComObjectRefRef	**[0005h 0005h] Obj5-brighter/darker**
Add new/ParameterRefRef	**[0008h] DPT-Value_GO5**

17.1.9 Hide Unwanted Data Points

To hide all unwanted data points (if there are any) select **Dynamic** and choose right mouse button menu **Add new Channel**. Enter **Channel 0** and number **0**.

Select **Channel 0** and choose right mouse button menu **Add new/ParameterBlock**. Enter these values in the dialog window

- Name: HiddenPage
- Text: Hidden Page

and hit **OK**.

Create a flag for hiding the data points: create an enum type by selecting Static and choose right mouse button menu **Add new/ParameterTypeRestriction**. Enter these values in the dialog window

- Base: Value
- Internal Name: HideFlag_t
- Size in bit: 0001h

and hit **OK**. Enter values for enum type by selecting its name and choose right mouse button menu **Add new Enumeration Value**. Enter these values in the dialog window

- Text: Shown
- Value: 0000h

and hit **OK**. Add one more value

- Text: Hidden
- Value: 0001h

and hit **OK**. Create the parameter by selecting **Parameters and Unions** and choose right mode button menu **Add new/VirtualParameter**. Enter these values in the dialog window

- Access: None
- Internal Name: HideFlag
- Parameter Type: HideFlag_t
- Text: HideFlag
- Create ParameterRef: True

and hit **OK**.

Add this parameter to the page **HiddenPage** by selecting the page name and choose right mouse button menu **Add new/ParameterRefRef**. Enter the parameter **[000Ah] HideFlag** and hit **OK**.

Add a choice to the **HiddenPage**. Choose right mouse button menu **Add new/Choose**. Enter these values in the dialog window

- Parameter: [000Ch] HideFlag
- Create all Whens (enum only): False
- Create Default when: False

and hit **OK**.

Select the newly created **HideFlag** choice and choose right mouse button menu **Add new When**. Enter these values in the dialog window

- Default: False
- Test: [0000h] Shown

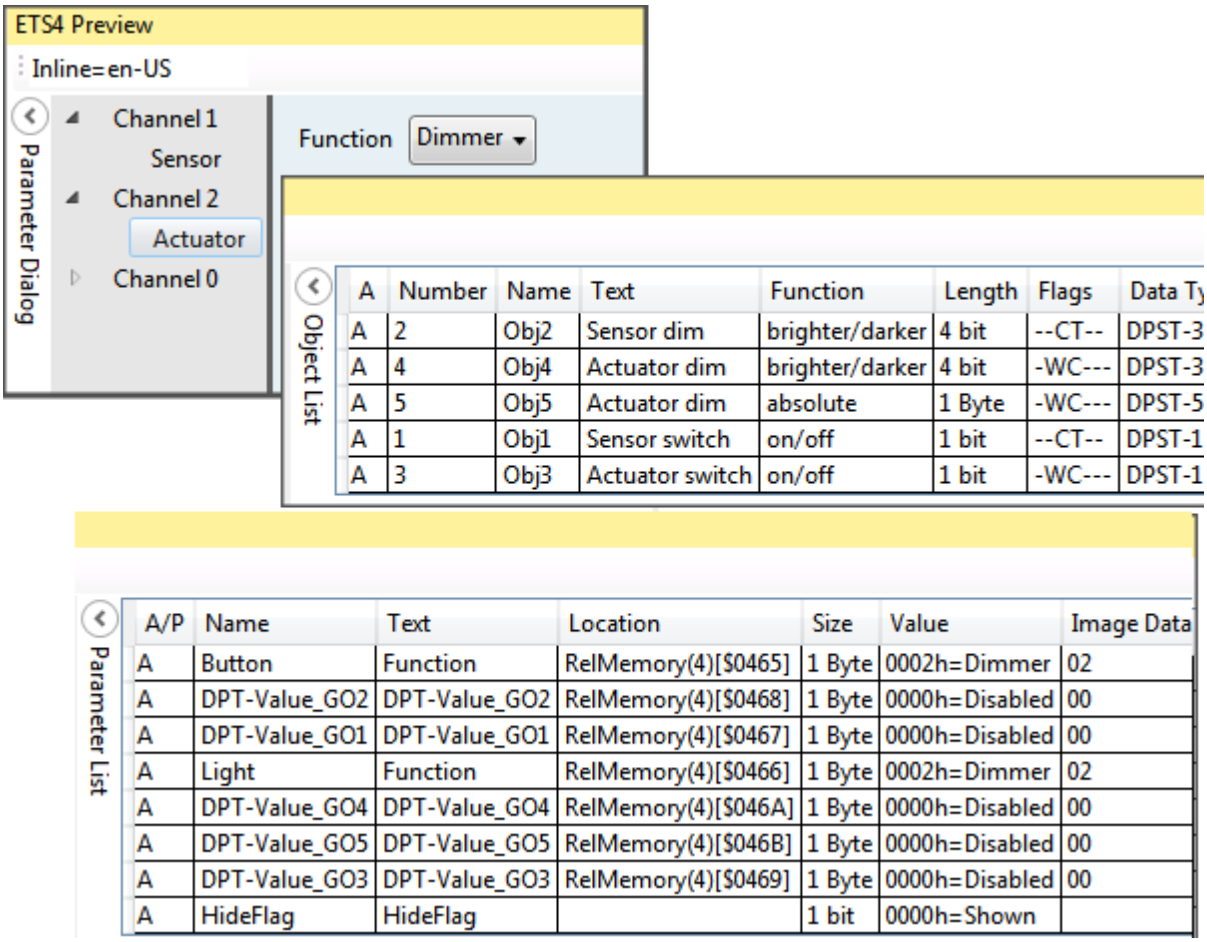
and hit **OK**. Add one more test **[0001h] Hidden**.

Select **ComObjectRefs**, go to the list and select all items which should be invisible (item #0, #8 to #1000, if there are any). Copy (right mouse button at first (empty) column) and paste them to (**Hidden**).

17.1.10 Preview the Work so far

To check the work so far, select the name of the application and choose menu **View/ETS4 Preview**.

Select **ObjectList** and change some parameters (e. g. Lighting Actuator/Function). See if the ObjectList changes. Some entries should appear, some vanish according to our configuration in MT.



Close the preview window and continue.

17.1.11 Create New Product

Click the project name in the Solution Explorer (**My_KNX_BAOS**) and use the menu **Project/Add New Item...**, select **Catalog**, edit its name (e. g. **My_KNX_BAOS_Cat.mtxml**) and hit **Add**.

Select **Catalog** in the newly opened tab and use right mouse button menu **Add New CatalogSection**. Enter values in the dialog window:

- Name: My_KNX_BAOS
- Number: CN2016-10-26

and hit **OK**. Enter values in the second dialog window and hit **OK**. Now the line **Delete me after creating...** can be deleted.

Select the newly created entry and use right mouse button menu **Add New CatalogItem**. Select your values (since we have only one hardware/program/product we cannot change anything) in the dialog window and hit **OK**.

17.1.12 Export the Project

Export project for testing

Select name of the application (e. g. **My_KNX_BAOS_App**) and choose menu **View/ETS4 Preview** and check this preview.

Build all choosing menu **Build/Build Solution** and create a test project using menu **Edit/Create test project**. Select your catalog item and hit **OK**. Save the knxproj-file. This is the test project which can be imported in ETS.

Clean up

Delete **Readme.txt**.

Note: A test project is needed to import the unregistered product database entry in ETS.

17.2 Test the Individual ETS Database in ETS

To test the database, start ETS and do the following:

Import the test project created by MT into ETS. Select the Projects tab and **Import...** from the tool bar. Follow the dialog window and load the project file.

Open the imported project and create a topology in the **Devices** view: Select the new created device and set the individual address in the Properties panel (e. g. 1.1.32). This automatically creates the topology **2 New area** and **1.1 New line** containing the device at 1.1.32.

Create two Group Addresses in the **Group Addresses** view: **New main group/New middle group/Switch** and **New main group/New middle group/Dim**.

Go back to the device, view and enable all data points in the **Parameters tab**:

Enable both **Lighting Sensor** and **Lighting Actuator** as **Dimmer**.

Drag the data points **1: Sensor switch** and **3: Actuator switch** to the group address **Switch**.

Drag the data points **2: Sensor dim** and **4: Actuator dim** to the group address **Dim**.

Finally press the learning key on the Development Board and select **Download/Full download**.

Switching and dimming of the LED is possible now.

17.3 Example to create more Parameter Bytes

Sometimes 250 parameter bytes are not enough. The BAOS Module has quite a lot flash memory which can be used for more parameter bytes. This example shows how to create such a database.

17.3.1 Project

Create a new MT project containing

- Hardware.mtxml
- Catalog.mtxml
- Application.mtxml

as shown in section, “Example for Creating an Individual ETS Database”.

17.3.2 Binary Import

Edit the application by selecting the application tab and then its name (e. g. **My_KNX_BAOS_App**). Use right mouse button menu **Import binary data**, browse to the s37-File which is also in the archive mentioned above:

Weinzierl_8xx_KNX_BAOS_ETS_Projects_for_Demo/
ETS_Project_using_individual_ETS_entry/
MT_Import_Files/
APP_BAOS_8xx_1000_Parameter_Bytes.s37,

open it and hit **OK**.

17.3.3 Using Objects

For every active group object you have to set the value of the used DPT in the **DP types block**.

- Location: Object x -> [Rel#04] + 007Ch + (x - 1); (x ≤ 1000)
- Example: Object 2 has DPT 1.007
  Create a hidden memory parameter with default value 0x01 on location [Rel#04] + 007Dh.
- Info: Take care that the stored DPT is always in line with the DPT of the active object.

17.3.4 Using Parameters

Set your parameters in the Parameter bytes block.

- Location: Parameter x -> [Rel#04] + 0465h + (x - 1); (x ≤ 1000)
- Example: Parameter byte 2
  Create a memory parameter on location [Rel#04] + 0466h.

17.3.5 Additional Settings

Device friendly name:

- Location: [Rel#04] + 005Bh
- Size: 30 bytes

Send Indications.

- Location: [Rel#04] + 0079h
- Value: 01h = activated/00h = deactivated
- Default value: 01h = activated

17.3.6 Speed up ETS Download

If you have, for example, 500 parameter bytes you can speed up your ETS download.

- Adapt load procedures and size of code segment.
- Adapt size of SUB_MCB_6 via hidden parameter:
  - Location: [Rel#04] + 0037h (2 byte value)
  - Value: 01F4h = 500 parameter bytes
- Adapt size of Parameter bytes block via hidden parameter:
  - Location: [Rel#04] + 0054h (2 byte value)
  - Value: 01F4h = 500 parameter bytes

For detailed information please contact Weinzierl Engineering GmbH.

18 KNX Certification

In order to ensure compliance with the KNX system requirements, any KNX device, which:

- Has a KNX logo
- Is managed by ETS

must undergo a certification process. In this KNX certification process, the device is tested according to the requirements of the KNX standard.

Following requirements have to be fulfilled for a KNX certification of a product:

- The manufacturer has to be a member (Shareholder or licensee) of the KNX Association. The sign up process is managed by the KNX Association. For more information see knx.org -> KNX members -> Joining/Fees
- The manufacturer must have a quality management system according to the ISO 900x with certificate issued. For more information see **KNX Specification Vol. 5** available at the KNX Specifications page.
- The manufacturer has to provide a CE declaration for his product to ensure hardware requirements according to applicable standards. A KNX device has to comply with the following hardware requirements:
  - Environmental conditions
  - Electrical safety
  - Functional safety
  - Electromagnetic compatibility (EMC)

  All hardware requirements are listed in the **KNX Specification, Vol. 4** available at the KNX Specifications page.
- After product development, the manufacturer has to register the product which shall be certified. The whole registration process is managed by the KNX Association.
- The required tests for system conformity are explained in the **KNX Specification Vol. 8**. They can be done after a completed registration of the product.

If a device is based on KNX BAOS Module 830, 832 or 838, the device inherits the certified status of it. Therefore only the application specific tests (inter-working/functionality) are required.

For further details please contact info@weinzierl.de.

19 Glossary

Acronym	Definition
AL	Application Layer
cEMI	Common External Message Interface
DPT	Data Point Type
EMI	external message interface
ETS	Engineering Tool Software
GO	Group Object
ISR	Interrupt Service Routine
JTAG	Join Test Action Group
KNX	Standard for building automation
KNX-TP	KNX on twisted pair
LL	Link Layer
LSB/MSB	Least Significant Byte/Most Significant Byte
MCB	Memory Control Block
MT	Manufacturing Tool
MTXML	File format of MT (xml)

Acronym	Definition
NL	Network Layer
OSI	Open System Interconnection Reference Model
PID	Property ID
RAM	Random Access Memory
ROI	Read On Initialization
s19, s28, s37	ASCII-base file format to encode binary files from Motorola
S-Mode	System configuration mode (ETS)
System1	Device model of KNX devices (implemented at BCU1)
System2	Device model of KNX devices (implemented at BCU2)
System7	Device model of KNX devices
SystemB	Device model of KNX devices (table count > 255)
TL	Transport Layer
TP	Twisted Pair
TP1	Twisted Pair 1 (KNX medium)
TP-UART-IC	Twisted Pair Universal Asynchronous Receive and Transmit IC (Siemens)
USART	Universal Synchronous Asynchronous Receive and Transmit

Acronym	Definition
WzEn	Weinzierl Engineering
XML	Extensible Markup Language