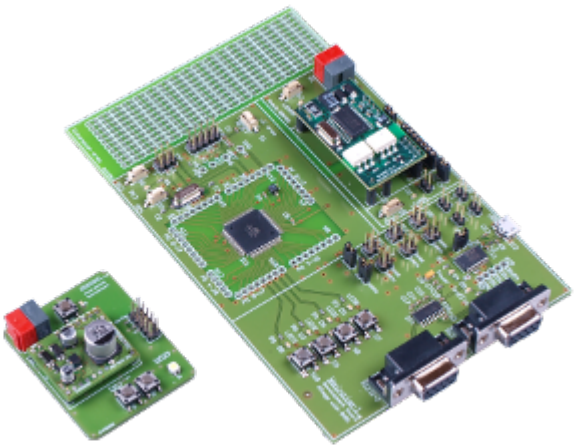
WEINZIERL

KNX BAOS Development Kit 1.4

Users Guide

BAOS Modules 0820/0822



KNX BAOS Development Kit 1.4 Users Guide
BAOS Modules 0820/0822
Edition 1

The KNX BAOS Modules 0820 and 0822 allow a jump start into KNX device development with minimal costs. Both versions include a KNX transceiver and a microcontroller 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 0820 provides electrical isolation and is suitable for devices with separate power supply. The KNX BAOS Module 0822 version offers direct coupling and provides a power supply for the application from the bus.

For the development we provide tools and a generic ETS entry with up to 250 group objects (communication objects). For a quick start in the development, we recommend our starter kit with a demo project in source code for ATmega 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 tool can be found at our *web page*[1].

For products with higher volume an 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* .

[1] http://www.weinzierl.de

Preface

1. Document Conventions

This documentation uses the following conventions to determine normal text from certain important information, like input or output text, listings, tips, hints, warnings, etc.

`Bold`

Used to highlight system input, entered text, commands, file names, directories, name entries and dialog buttons. Also used to highlight key input. For example:

- In Net'n Node use **`Add Port >> KNX via USB`** to connect via the KNX USB 311 device.

`Italic`

Any text written in italics denotes text which is not written literally. It must be replaced by the reader by their meaningful meanings. For example:

- To scan all available properties enter *`Area, Line, Device`* and hit **`Scan`**.

`Pull-quote Conventions`

Terminal output and source code listings are set into a specially formatted box. For example:

The output of the **`date --iso-8601`** command looks like:

```
2013-12-02
```

Example of a source-code listings:

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Hello World!\n");
    return 0;
}
```

`Notes and Warnings`

Last, but not least, this document uses the following visual styles to point out important aspects.

Note

This is a note which is worth to be memorized. It can help the reader in many ways or help him to understand the preceding text.

Important

This is an important note. The reader should consider its content and remember it.

Warning

A warning is the most important note. It keeps the reader from doing things which can cost him a lot of work, time or even damage the hardware/software.

2. 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 MERCHANTABILITY, 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.

2.1. Definitions

"Firmware" and "SDK" mean all provided to you in binary code by Weinzierl Engineering GmbH.

"SDK" is a Software Development Kit, which means all software to develop, write and compile software for its defined target.

"Firmware" is the software already stored into its defined target (generally a micro controller) by Weinzierl Engineering GmbH. This firmware is provided in binary format on its hardware target and may not be altered or changed by the user.

2.2. 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 firmware, software or SDK for the purposes of developing applications that communicate with KNX interface modules or devices from Weinzierl Engineering GmbH.

b. develop and distribute all software done with the SDK provided by Weinzierl Engineering GmbH, but not the SDK itself.

c. reproduce and distribute the SDK and software for the sole purpose of running your application.

2.3. Restrictions

a. The Weinzierl Firmware (installed in ROM, EPROM, EEPROM, flash or any other circuit) is limited to be used on Weinzierl branded modules or devices. It is not allowed to distribute the Weinzierl Firmware without expressed written permission from Weinzierl Engineering GmbH.

b. The Weinzierl software or SDK (available on CD-ROM, or at *the Weinzierl web page*[1].) is also limited to be used on Weinzierl branded devices or for use on appropriate computers in conjunction to these Weinzierl branded devices or development boards. It is not allowed to distribute the Weinzierl software without expressed permission from Weinzierl Engineering GmbH.

c. You may not and you agree not to, or to enable others to, duplicate, decompile, reverse engineer, disassemble, attempt to derive the source code of, decrypt, modify, or create derivative works of the Weinzierl firmware/software, or any part thereof.

2.4. Overview of Restrictions/Permissions

Subject	Restrictions/Permissions
Firmware in module	**`Not`** allowed to copy.
Source code of demo application and framework	Allowed to copy in **`binary`** form in conjunction with **`Weinzierl hardware`**.
Software tools	**`Not`** allowed to copy.

3. About Us

3.1. 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`* .

3.2. Our Services And Products

- **`Devices for KNX`**

  As a solutions provider we offer complete KNX devices for this global standard of building. Convince yourself of the performance of our pre-finished KNX Devices.

[1] http://www.weinzierl.de

- **`Modules for KNX`**

  You want to equip devices with KNX fast, cheap and without major development effort? Then you should take a look at our KNX modules.

- **`Stacks for KNX`**

  KNX describes a complex protocol, which means a considerable effort in the implementation and certification. With our KNX stacks we take a lot of work from you.

- **`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 in conjunction with your development department.

3.3. 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.

Note

For more information about KNX-systems, see the *KNX web site*[2].

4. Feedback

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

[2] http://www.knx.org

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.

The KNX system is organized decentralized, which means there is no central bus management and so the communication is unlikely to fail even if other devices drop out. This makes the whole system fault tolerant and reliable.

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 light switch turns on what lighting? *ETS* is a standardized tool by the KNX organization to commission all devices in a KNX network.

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. lighting, heating, shutter). Some devices can be both, sensors and actuators (e. g. a shutter which reports its movement).

Once the house installation is commissioned it can always be changed by removing, changing or adding new devices. The devices must be configured again to meet the changes. Even if there is no change in the installation the configuration can also be changed if wanted (e. g. to move some lights to another group of lights which are all controlled by a certain switch).

Some in house devices are not KNX capable but must be, to add then to the KNX installation.
To do this, the manufacturer has to add KNX capability to them. To ease this process the *KNX BAOS 0820/0822 Module* offers the possibility to connect a device to the KNX bus. This is done by connecting the device to the BAOS Module by a serial port (UART). The BAOS Module is connected to the KNX bus. It handles the whole KNX communication, configuration and management. The other side, the device, must implement an application which handles the device itself. An example application is included in this package.

The BAOS Module serves as an interface at the *telegram* and *data point/parameter* level. The telegram level is for more experienced usage and offers the possibility to manage KNX messages by the device itself.

A data point represents a 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 device will be notified.

The *parameters* serve as configuration options. These can be read by the device.

The data point/parameter value exchange and notification is realized by the *BAOS Protocol*. *BAOS* stands for "Bus Access and Object Server". It can handle up to 250 data points and 250 parameters. These must be configured by ETS.

ETS needs *product information* about the device (including the BAOS Module) to configure it. This information is included in this package in a generic way. It is sufficient for developing the device but it should be developed into a more user friendly product for the use by an installer.

The resulting product information must be provided alongside with the device (containing the BAOS Module) for certification by the KNX organization.

The Demonstration Board

The *Demonstration Board* is a simple board which demonstrates the usage of KNX by switching or dimming a LED with two buttons. For development projects we recommend the Development Board. (See *Chapter 3, The Development Board*)

Sources for the application (see *Chapter 6, The Application Framework*) and schematics (see *Section 2.5, "Schematics Of The Demo Board"*) are available.

2.1. Introduction

2.1.1. The Demo Board

The *Demo Board*, demonstrates the usage of the BAOS Module. It uses the 0822 module which offers a fully functional communication stack which can be used by the programmer.

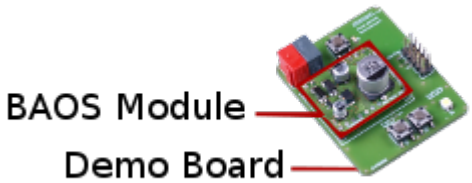


Figure 2.1. The Demo Board and the BAOS Module

The Demo Board contains an 8 bit, 32 kB Flash AVR ATmega328 microcontroller and is targeted for a demonstration of the BAOS usage.

Every Board consists of two parts:

1. The main board is the *Demo Board*. It contains a microcontroller and an *ISP/DebugWire* connector which enables it to be freely programmed by the user.

2. The *BAOS Module*, which is located on the connectors of the main board. This daughter board is responsible for the KNX communication and has its own microcontroller and firmware. The firmware of the module is not alterable by the user.

The demo application is a one channel application for one sensor (2 push buttons) and one actor (LED). This channel can be linked by ETS to other channels of other KNX devices and/or to its own channel.

2.1.2. How To Connect The Device

The BAOS Module is a KNX device using the twisted pair KNX bus for power and communication.

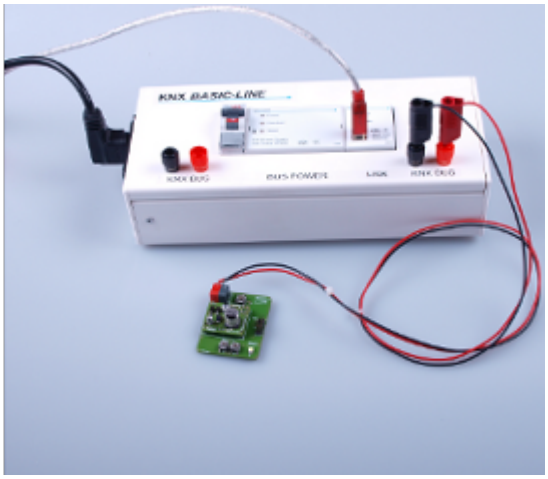


Figure 2.2. How to connect the device

To take the Demo Board in operation state, the following items and steps are necessary.

- A KNX power supply with choke.
- Optionally a KNX USB interface to commission the board via ETS.
- Optionally some sensors and actors to link their channels to the channels of the board.
- Make sure the BAOS Module is correctly connected to the Demo Board.
- Connect the KNX bus (polarity is protected).

2.1.3. Commissioning with ETS

ETS (Engineering Tool Software) is a manufacturer independent configuration tool software to configure intelligent home and building control installations with the KNX system. A short introduction is in *Appendix B, Commissioning With ETS*.

For demonstration a simple project is included in **`KnxBAOS_Demo/KnxBAOS_Demo_ETS_Projects/KnxBAOS_Demo.knxproj`**. Import the project which configures the board to simply handle its own LEDs. Open the project, select the device **`KNX BAOS 82x 87x`** with **`right mouse button`** and **Download** # **Download All**. Press the *Learn button* on the Board (red LED must light up briefly).

After the download is finished, press the two push button S1, S3 on the *Demo Board* to switch the LED on and off.	

2.2. The Demo Application

The demo application of the Demo Board is a two channel dimming actuator and sensor. The state of the actuator channel is shown by the white LED. The buttons S1 and S2 are used as independent output channel.

2.2.1. Data Points/Group Objects

Via KNX bus the data points (DP), also known as group objects, are accessible as follows:

DP#	DPT	Sensor/ Actuator	Demo Board
1	01 - 1 bit	Sensor	• Switch S1, S3 on/off. This simply switch a light on and off. • Shutter step S1, S3 open/close. For moving a shutter just one step.
2	03 - 4 bit	Sensor	• Dimming S1, S3 brighter/darker. This dims a light relatively. • Shutter move S1, S3 up/down. For starting or stopping movement of a shutter.
3	01 - 1 bit	Actuator	Switch LED on/off.
4	03 - 4 bit	Actuator	Dimming LED relatively (up/down).
5	05 - 8 bit	Actuator	Dimming LED absolutely.

The demo project **`KnxBAOS_Demo.knxproj`** uses only data point #1 connected to data point #3, which simply switches the LED. The dimming feature must be activated in the parameter page in the ETS. More information about data point types (DPT), see *Section A.5, "Data Point Types"*.

All these data points can be commissioned via the ETS tool. The Switches and the LEDs can be linked there to perform the desired functions.

Note

In ETS take care of parameter settings as they enable or disable these data points. They can also configure the functionality (i. e. data point types). Cf. data points #1 and #2.

2.2.2. Parameters

The BAOS Module has 250 parameters which can be changed by the ETS. Each parameter is one byte (0-255) an can be read/written by the application via the BAOS Protocol.

`Parameter #1 controls data point #1 and #2 functionality:`

Value	DP#1 Function	DP#2 Function
0	not used	not used
1	used as 1 bit switch sensor (S1, S3)	not used
2	used as 1 bit switch sensor (S1, S3)	used as 4 bit dimming sensor (S1, S3)
3	used as 1 bit shutter step sensor (S1, S3)	used as 1 bit shutter move sensor (S1, S3)

`Parameter #2 controls data point #3, #4 and #5 functionality:`

Value	DP#3 Function	DP#4 Function	DP#5 Function
0	not used	not used.	not used.
1	used as 1 bit switch actuator (D2)	not used.	not used.

Value	DP#3 Function	DP#4 Function	DP#5 Function
2	used as 1 bit switch actuator (D2)	used as 4 bit relatively dimming actuator (D2)	used as 1 byte absolutely dimming actuator (D2)

2.2.3. First Test

The factory default configuration can be changed via ETS. Press the button S3 shortly. The LED should go on. Pressing S1 switches the LED off.

2.2.4. The Demo Application Framework

The demo application is included in this package as source code. To learn about it see *Chapter 6, The Application Framework*.

2.3. Set The Demo Board Back To Delivery State

The demo application is included in this distribution. To reset the firmware back to delivery state, do the following:

1. Install and start *Atmel Studio* (see *Chapter 11, Programming The Base Board*).
2. Choose menu **Tools** # **Device Programming** which shows a dialog window.
3. Use the correct tool (e. g. **JTAGICE3**) for the Demo Board: Device: **ATmega328** and Interface: **ISP**. Push the button **Apply** to establish the connection. Verify it by pushing the button Device signature **Read**. It must show a hexadecimal value.
4. Go to tab **Production file**. Load the *ELF* file which is included in this distribution:

   `KnxBAOS_Demo/KnxBAOS_Demo_Binary_DemoBoard328/DemoBoardSoftware.elf` for the Demo Board.
5. Make sure **Flash** checkmark is set as well as **Erase memory before programming** .
6. Program the device by pressing the button **Program**.

Now the Demo Board should be in delivery state. This process does not only transfer the firmware, it also resets the *FUSE*s. (See data sheets for details.)

Important

Setting the Demo Board back in delivery state does not reset the BAOS Module. So any data point connections, parameter and KNX individual address remain untouched. To reset these use the ETS tool.

2.4. Hardware

This section describes the hardware of the Demo Board, its usage and how to connect the board for usage with the KNX bus.

The *Demo Board* contains an *ATmega328* microcontroller for the user application. It is clocked by a 7.3728 MHz crystal. It contains one PWM-driven LED and two push buttons witch are usable

by the application software. A second LED and additional push button is for the BAOS Module for programming the KNX address.

The BAOS Module (mounted on the connectors of the Demo Board) contains another microcontroller which handles the KNX communicaton.

2.4.1. Demo Board (Base Board)

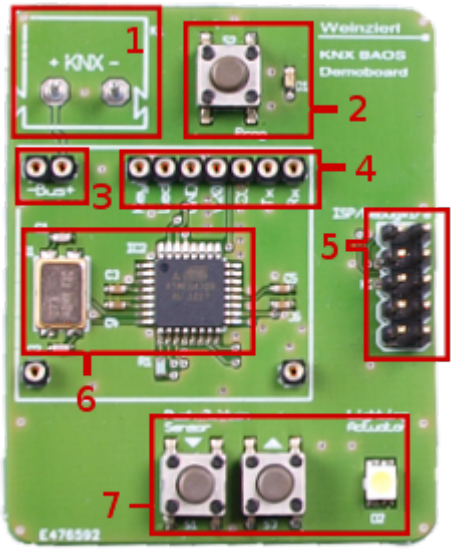


Figure 2.3. Demo Board

The Demo Board contains the following:

1. The connector to the KNX bus.
2. *Learning Key* and LED. This key and LED are not for the application. They are being used to program the KNX individual address. (e. g. 1.1.32).
3. Connection of the KNX bus to the BAOS Module.
4. Serial connection including VCC/GND, LED and Learn Key for data exchange between the the BAOS Module and the Base Board.
5. ISP/DebugWire connector for programming the microcontroller of the Base Board.
6. The microcontroller and its crystal.
7. Two push buttons and LED which are also handled by the user application. It can be dimmed by the microcontroller's PWM.

2.4.2. Fuses Of The ATmega

Fuse bits are a specialty of the ATmega microcontrollers. These bits control the behavior of the MCU and its internal resources like the oscillator or the watch dog. Lock bits can protect some sections in the flash memory to save important sections. Care must be taken to set the these bits correctly of the ATmega328 microcontroller. If your code fails to be executed or uploaded, check the Fuse and Lock bits. They should be set as follows:

```
BODLEVEL = 2V7
RSTDISBL = [ ]
DWEN = [ ]
SPIEN = [X]
WDTON = [ ]
EESAVE = [ ]
BOOTSZ = 2048W_3800
BOOTRST = [ ]
CKDIV8 = [ ]
CKOUT = [ ]
SUT_CKSEL = EXTXOSC_8MHZ_XX_16KCK_14CK_65MS

EXTENDED = 0xFD (valid)
HIGH = 0xD9 (valid)
LOW = 0xFF (valid)

LB = NO_LOCK
BLB0 = NO_LOCK
BLB1 = NO_LOCK

LOCKBIT = 0xFF (valid)
```

2.4.3. BAOS Module

A piggyback board is located on the Demo Board. This is the BAOS Module which is responsible for the whole KNX communication. The BAOS Module comes with a certified KNX stack and can be configured with ETS. It handles the KNX communicaton and is connected via a serial port (UART) to the Demo Board. It is powered by the bus and supplies the Demo Board, too.

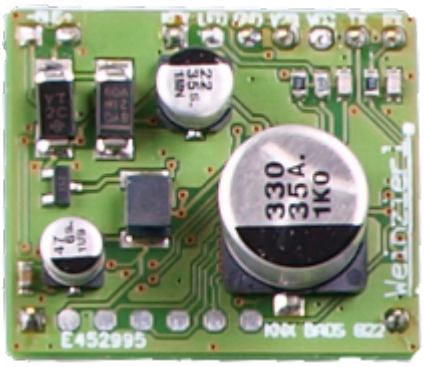
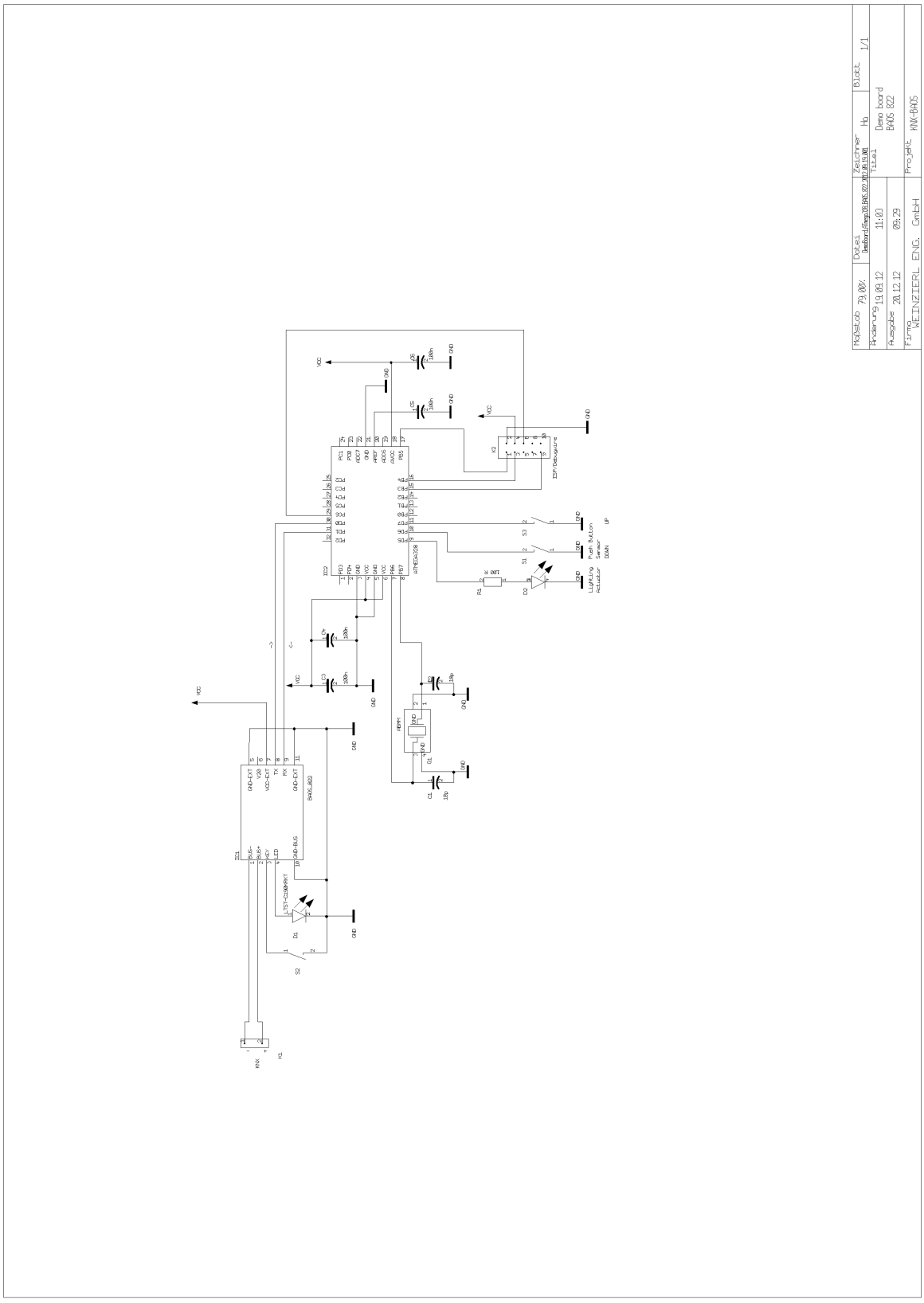


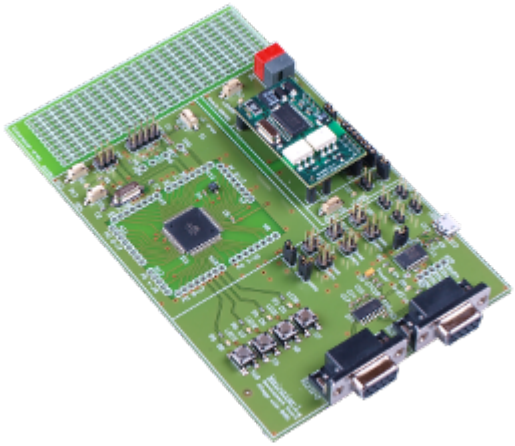
Figure 2.4. BAOS Module 0822

For more information see *Chapter 4, BAOS Modules*.

2.5. Schematics Of The Demo Board



The Development Board



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 microcontroller.

Sources for the application (see *Chapter 6, The Application Framework*) and schematics (see *Section 3.5, "Schematics Of The Development Board"*) are available.

3.1. Introduction

3.1.1. The Development Board

This product, the *Development Board*, makes the entry into the KNX development as easy as possible. It uses either the 0820 or the 0822 module which offers a fully functional communication stack which can be used by the programmer.

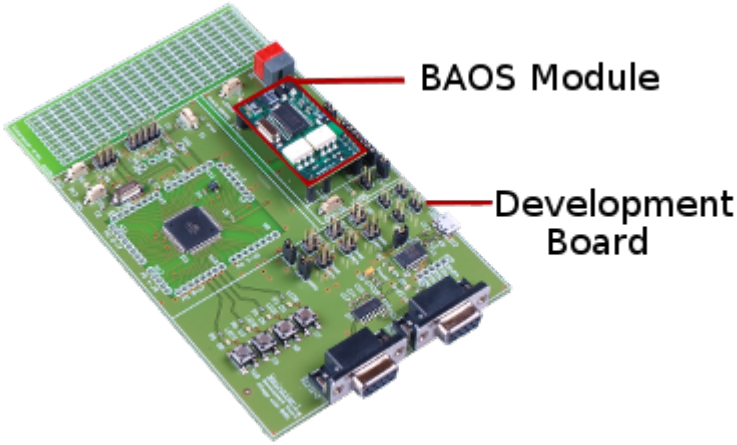


Figure 3.1. The Development Board and the BAOS Module

The Development Board contains an 8 bit, 128 kB Flash AVR ATmega128A microcontroller and is targeted for developing own applications with the BAOS. Additional elements on this board can be used, like the other LEDs, buttons and UART1. Even more elements can be added in the *Extension area* in conjunction with the free port bits of the microcontroller.

Two RS-232 and a USB connector are available to connect various communication partners. These connectors can be activated by the usage of the jumpers (see *Section 3.4.2, "Jumper Usages"*). As

an example a PC can be connected via USB to the BAOS Module. In this case the PC communicates directly to the BAOS Module. This enables development of an application on a PC before porting it to the microcontroller.

Note

In this case the connection of the Development Board to the BAOS Module is disabled (UART0).

A *DemoClient* is included to demonstrate the usage of the *BAOS Protocol*. See *Chapter 7, Connecting PC Via BAOS Interface* for more information.

It is also possible to communicate to the microcontroller of the Development Board (UART1), and thus the own application. This will not break the connection to the BAOS Module which uses UART0.

Note

Take care of the jumper settings as they affect the functionality of the device. See *Section 3.4.2, "Jumper Usages"*.

Every Development Board consists of two parts:

1. The main board is the *Development Board*. It contains a microcontroller, *JTAG* and *ISP* connectors which enable it to be freely programmed by the user.
2. The *BAOS Module*, which is located on the connectors of the main board. This daughter board is responsible for the KNX communication and has its own microcontroller and firmware. The firmware of the module is not alterable by the user.

The demo application is a two channel application for one sensor (1 push button) and one actor (LED). These channels can be linked by ETS to other channels of other KNX devices and/or to its own channels.

3.1.2. How To Connect The Device

The BAOS Module is a KNX device using the twisted pair KNX bus for power and communication.

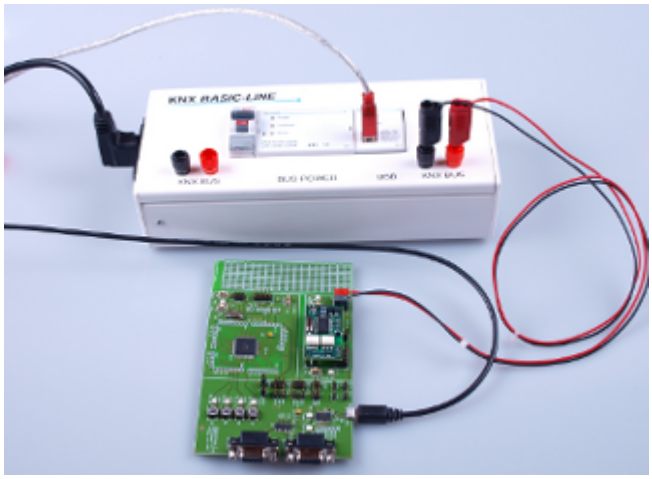


Figure 3.2. How to connect the device

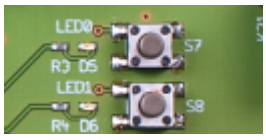
To take the Development Board in operation state, the following items and steps are necessary.

- A KNX power supply with choke.
- Optionally a KNX USB interface to commission the board via ETS.
- Optionally some sensors and actors to link their channels to the channels of the board.
- Make sure the 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 device.

3.1.3. Commissioning With ETS

ETS (Engineering Tool Software) is a manufacturer independent configuration tool software to configure intelligent home and building control installations with the KNX system. A short introduction is in *Appendix B, Commissioning With ETS*.

For demonstration a simple project is included in **`KnxBAOS_Demo/KnxBAOS_Demo_ETS_Projects/KnxBAOS_Demo.knxproj`**. Import the project which configures the board to simply handle its own LEDs. Open the project, select the device **`KNX BAOS 82x 87x`** with **`right mouse button`** and **Download** # **Download All**. Press the *Learn button* on the Board (red LED must light up briefly).

After the download is finished, press the two push button S7, S8 on the *Development Board* to switch the LED0 on and off.	

3.2. The Demo Application

The demo application of the Development Board is a two channel dimming actuator and sensor. The state of the actuator channel is shown by the LED0. The buttons S7 and S8 are used as independent output channel.

3.2.1. Data Points/Group Objects

Via KNX bus the data points (DP), also known as group objects, are accessible as follows:

DP#	DPT	Sensor/ Actuator	Development Board
1	01 - 1 bit	Sensor	• Switch S8, S7 on/off. This simply switch a light on and off. • Shutter step S8, S7 open/close. For moving a shutter just one step.
2	03 - 4 bit	Sensor	• Dimming S8, S7 brighter/darker. This dims a light relatively. • Shutter move S8, S7 up/down. For starting or stopping movement of a shutter.
3	01 - 1 bit	Actuator	Switch LED0 on/off.
4	03 - 4 bit	Actuator	Dimming LED0 relatively (up/down).
5	05 - 8 bit	Actuator	Dimming LED0 absolutely.

The demo project **`KnxBAOS_Demo.knxproj`** uses only data point #1 connected to data point #3, which simply switches the LED. The dimming feature must be activated in the parameter page in the ETS. More information about data point types (DPT), see *Section A.5, "Data Point Types"*.

All these data points can be commissioned via the ETS tool. The Switches and the LEDs can be linked there to perform the desired functions.

Note

In ETS take care of parameter settings as they enable or disable these data points. They can also configure the functionality (i. e. data point types). Cf. data points #1 and #2.

3.2.2. Parameters

The BAOS Module has 250 parameters which can be changed by the ETS. Each parameter is one byte (0-255) and can be read/written by the application via the BAOS Protocol.

`Parameter #1 controls data point #1 and #2 functionality:`

Value	DP#1 Function	DP#2 Function
0	not used	not used
1	used as 1 bit switch sensor (S7, S8)	not used
2	used as 1 bit switch sensor (S7, S8)	used as 4 bit dimming sensor (S7, S8)
3	used as 1 bit shutter step sensor (S7, S8)	used as 1 bit shutter move sensor (S7, S8)

`Parameter #2 controls data point #3, #4 and #5 functionality:`

Value	DP#3 Function	DP#4 Function	DP#5 Function
0	not used	not used.	not used.
1	used as 1 bit switch actuator (LED0)	not used.	not used.

Value	DP#3 Function	DP#4 Function	DP#5 Function
2	used as 1 bit switch actuator (LED0)	used as 4 bit relatively dimming actuator (LED0)	used as 1 byte absolutely dimming actuator (LED0)

3.2.3. First Test

The factory default configuration can be changed via ETS. Press the button S8 shortly. The LED should go on. Pressing S7 switches the LED off.

3.2.4. The Demo Application Framework

The demo application is included in this package as source code. To learn about it see *Chapter 6, The Application Framework*.

3.3. Set The Development Board Back To Delivery State

The demo application is included in this distribution. To reset the firmware back to delivery state, do the following:

1. Install and start *Atmel Studio* (see *Chapter 11, Programming The Base Board*).
2. Choose menu **Tools** # **Device Programming** which shows a dialog window.
3. Use the correct tool (e. g. **JTAGICE3**) for the Development Board: Device: **ATmega128A** and Interface: **JTAG**. Push the button **Apply** to establish the connection. Verify it by pushing the button Device signature **Read**. It must show a hexadecimal value.
4. Go to tab **Production file**. Load the *ELF* file which is included in this distribution:

   `KnxBAOS_Demo/KnxBAOS_Demo_Binary_DevBoard128/DevBoardSoftware.elf` for the Development Board.
5. Make sure **Flash** checkmark is set as well as **Erase memory before programming** .
6. Program the device by pressing the button **Program**.

Now the Development Board should be in delivery state. This process does not only transfer the firmware, it also resets the *FUSE*s. (See data sheets for details.)

Important

Setting the Development Board back in delivery state does not reset the BAOS Module. So any data point connections, parameter and KNX individual address remain untouched. To reset these use the ETS tool.

3.4. Hardware

This section describes the hardware of the Development Board, its usage and how to connect the board for usage with the KNX bus.

The *Development Board* contains an *ATmega128A* microcontroller for the user application. It is clocked by a 3.6864 MHz crystal. The board contains 4 LEDs and 4 push buttons witch are usable by

the application software. Three more LEDs and additional push buttons can be used for programming the KNX address.

The BAOS Module (mounted on the connectors of the Development Board) contains another microcontroller which handles the KNX communicaton.

3.4.1. Development Board (Base Board)

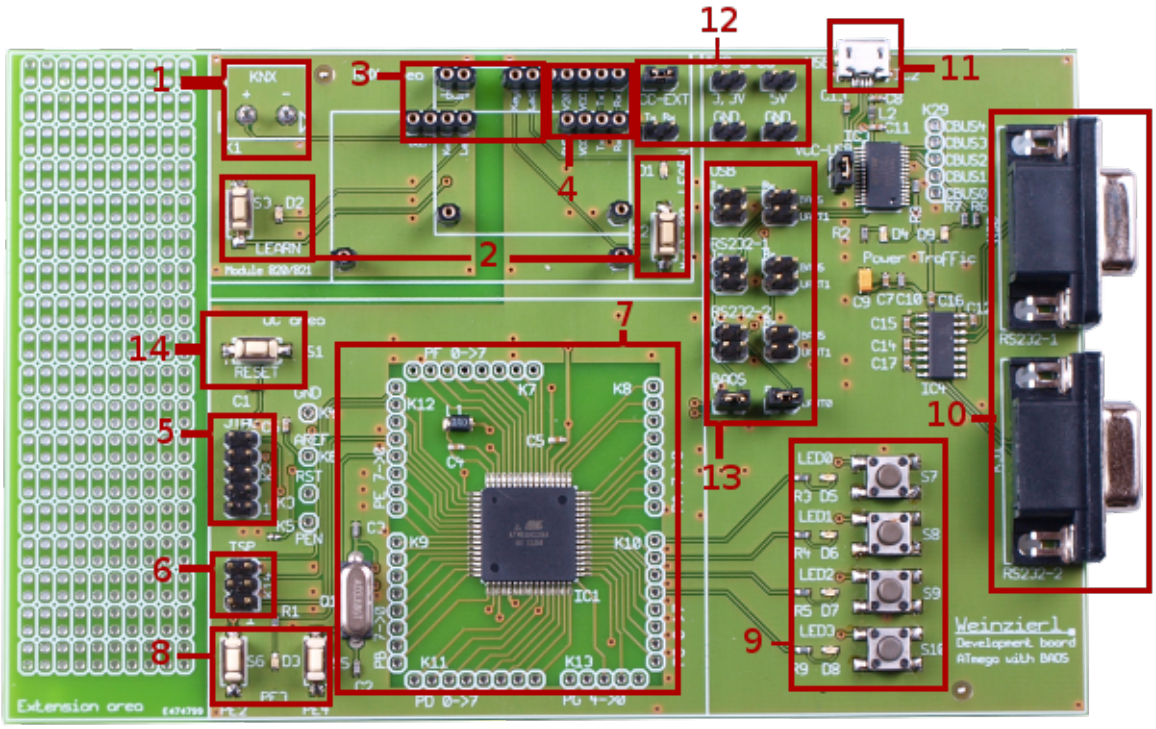


Figure 3.3. Development Board

The Development Board contains the following:

1. The connector to the KNX bus.
2. *Learning Keys* and LEDs. These keys and LEDs are not for the application. They are used to program the KNX individual address. (e. g. 1.1.32). The Development Board can host different BAOS Modules (0820 and 0822). Due to different power concepts individual LED/key pairs exist for each form factor. The 0820 has optical couplers, which isolate galvanically the BAOS Module from the Development Board. In this case the Development Board must be powered by the USB port or by other means. In case of the 0822 the Development Board is powered by the BAOS Module.
3. Connection to the BAOS Board: LED, Learning Key and to the KNX bus. (See *Section 4.1, "Pinning Of The BAOS Modules"*)
4. Serial connection including VCC/GND for data exchange between the the BAOS Module and the Development Board. (See *Section 4.1, "Pinning Of The BAOS Modules"*)
5. JTAG connector for programming and debugging the microcontroller of the Development Board.
6. ISP connector for programming the microcontroller of the Development Board. (Debugging not possible.)
7. The microcontroller and its crystal.

8. One LED and two push buttons for the application software. These can be used for an own programmed learning mode. The LED can also be driven by PWM.
9. Push buttons and LEDs for the user application.
10. Two RS232 (UART) connectors for communication with the application or the BAOS Module. Configurable by jumpers. (See *Section 3.4.2, "Jumper Usages"*)
11. USB connector. Power supply for the board and UART (FTDI) connector for communication with the application or the BAOS Module. Configurable by jumpers. (See *Section 3.4.2, "Jumper Usages"*)
12. Jumpers for development and configuration:
    - **VCC-EXT**: Enables power supply via USB port for the optical couplers of the 0820. The 0822 does not have optical couplers, so leave it open in this case.
    - **Tx Rx**: Connector to enable listening to the serial communication between the BAOS Module and the Development Board.
    - **3,3V**: Connector for 3.3 voltage power (2 pins).
    - **5V**: Connector for 5 voltage power (2 pins).
    - **GND**: Connector for ground (2 pins).
    - **VCC-USB**: Enables the power supply via the USB connector. This is necessary if the BAOS Module 0820 is mounted.

      The BAOS Module 0822 supplies power from the KNX bus to the Development Board. If the USB connector is not in use this jumper must be closed. If the USB connector is supplying power this jumper must be open.

      It is closed by default.
13. Jumpers for serial I/O:

    Connects the serial port of the BAOS Module or the UART of the local microcontroller to the USB or RS-232 connector. In this case another device can communicate to the BAOS Module or to the application. See *Section 3.4.2, "Jumper Usages"* for all possible combinations.
14. A reset button is also available which resets the local MCU of this board (not the BAOS Module).

3.4.2. Jumper Usages

Usage of BAOS Module	VCC-EXT	VCC-USB
BAOS Module 0820	**closed**	**closed**
BAOS Module 0822	open	**closed**

Usage for serial communication	USB BAOS Tx/Rx	USB UART1 Tx/Rx	RS232-1 BAOS Tx/Rx	RS232-1 UART1 Tx/Rx	RS232-2 BAOS Tx/Rx	RS232-2 UART1 Tx/Rx	BAOS Tx/Rx
BAOS - Development Board (standard)	o/o	*don't care*	o/o	*don't care*	o/o	*don't care*	**c/c**

Usage for serial communication	USB BAOS Tx/Rx	USB UART1 Tx/Rx	RS232-1 BAOS Tx/Rx	RS232-1 UART1 Tx/Rx	RS232-2 BAOS Tx/Rx	RS232-2 UART1 Tx/Rx	BAOS Tx/Rx
BAOS - PC (USB)	**c/c**	o/o	o/o	*don't care*	o/o	*don't care*	o/o
PC (USB) - Development Board (UART1)	o/o	**c/c**	*don't care*	o/o	*don't care*	o/o	*don't care*
BAOS - PC (RS232 #1)	o/o	*don't care*	**c/c**	o/o	o/o	*don't care*	o/o
PC (RS232 #1) - Development Board (UART1)	*don't care*	o/o	o/o	**c/c**	*don't care*	o/o	*don't care*
BAOS - PC (RS232 #2)	o/o	*don't care*	o/o	*don't care*	**c/c**	o/o	o/o
PC (RS232 #2) - Development Board (UART1)	*don't care*	o/o	*don't care*	o/o	o/o	**c/c**	*don't care*
Special example: Tracing BAOS - Development Board and UART1 - PC (USB)	o/o	**c/c**	o/o	o/o	o/o	o/o	**c/c**

(o = open, **c = closed**)

3.4.3. Fuses Of The ATmega

Fuse bits are a specialty of the ATmega microcontrollers. These bits control the behavior of the MCU and its internal resources like the oscillator or the watch dog. Lock bits can protect some sections in the flash memory to save important sections. Care must be taken to set the these bits correctly of the ATmega128A microcontroller. If your code fails to be executed or uploaded, check the Fuse and Lock bits. They should be set as follows:

```
M103C = [ ]
WDTON = [ ]
OCDEN = [ ]
JTAGEN = [X]
SPIEN = [X]
EESAVE = [ ]
BOOTSZ = 4096W_F000
BOOTRST = [ ]
CKOPT = [ ]
BODLEVEL = 2V7
BODEN = [X]
SUT_CKSEL = EXTHIFXTALRES_16KCK_64MS

EXTENDED = 0xFF (valid)
HIGH = 0x99 (valid)
LOW = 0xBF (valid)

LB = NO_LOCK
BLB0 = NO_LOCK
BLB1 = NO_LOCK
```

```
LOCKBIT = 0xFF (valid)
```

3.4.4. BAOS Module

A piggyback board is located on the Development Board. This is the BAOS Module which is responsible for the whole KNX communication. The BAOS Module comes with a certified KNX stack and can be configured with ETS. It handles the KNX communicaton and is connected via a serial port (UART) to the Development Board. It is powered by the bus but only the 0822 supplies the Development Board, too. The 0820 has optical couplers which galvanically isolate the Development Board. So it must be powered by other means.

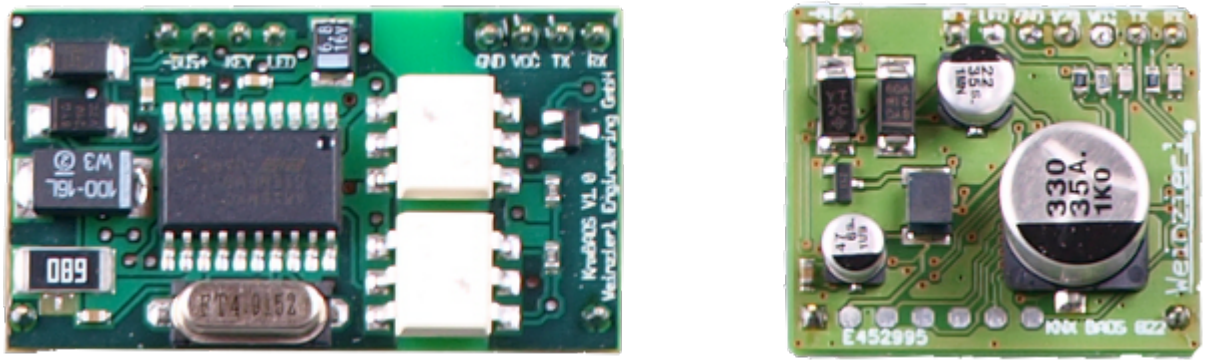
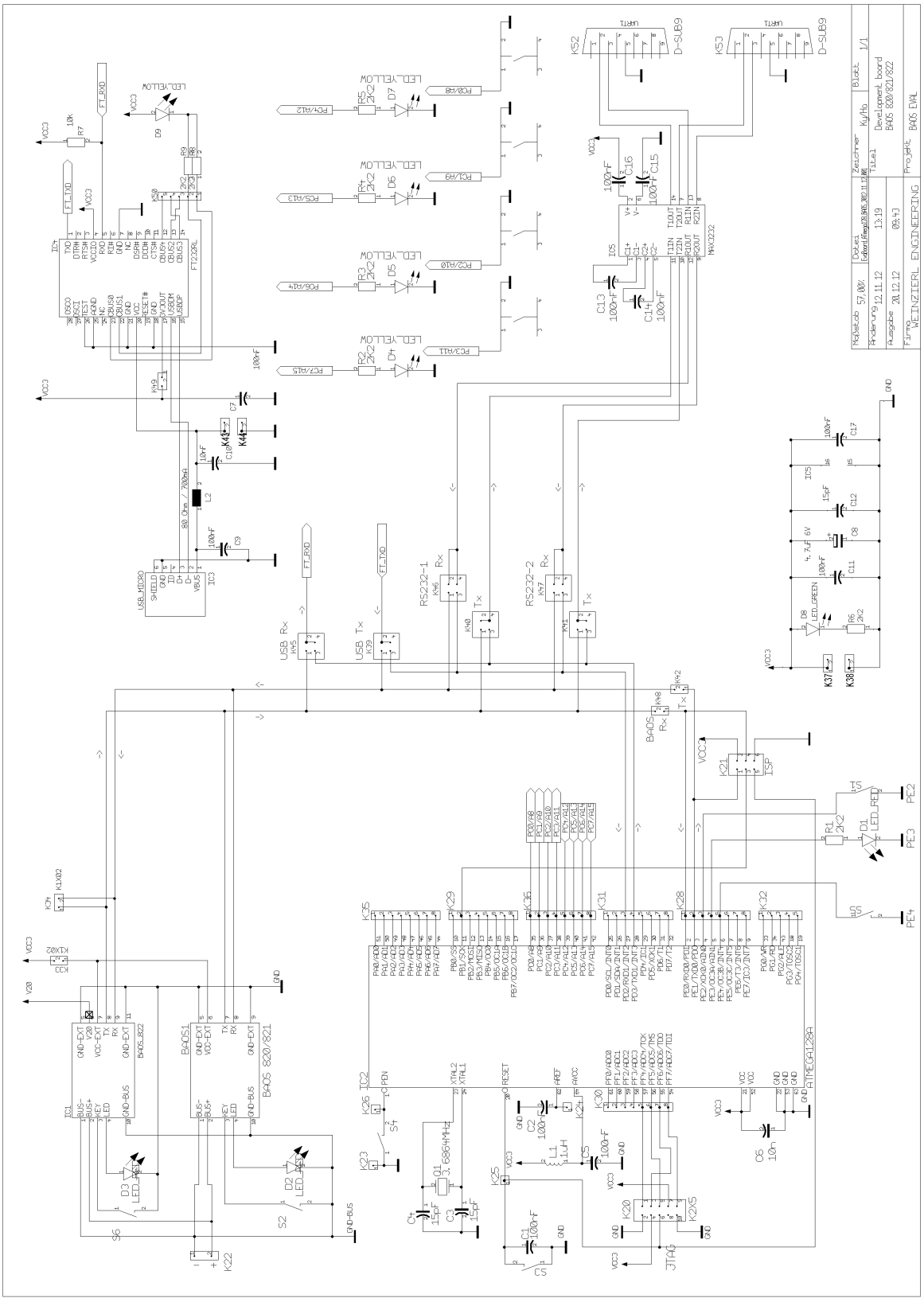


Figure 3.4. BAOS Modules (left: 0820, right: 0822)

For more information see *Chapter 4, BAOS Modules*.

3.5. Schematics Of The Development Board



BAOS Modules

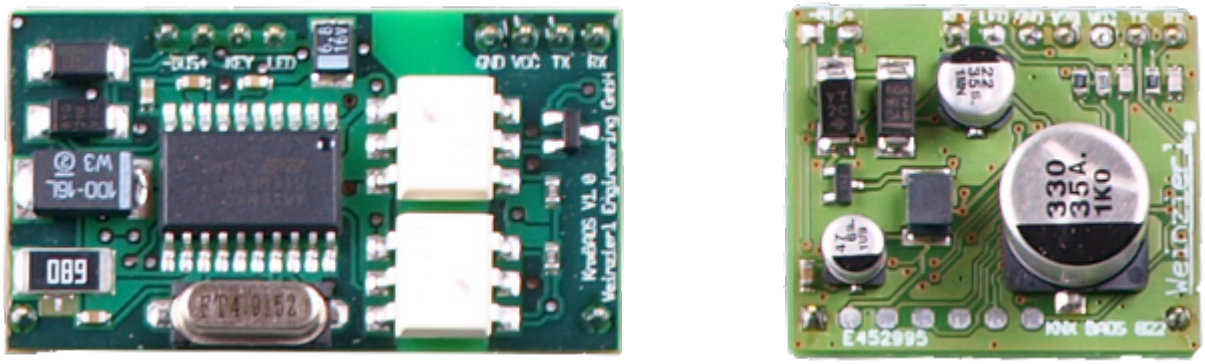


Figure 4.1. BAOS Modules (left: 0820, right: 0822)

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

- A KNX transceiver to send and receive telegrams via the KNX bus (see *Appendix A, About KNX*). This is also the power source for the module.
- A serial port (UART) for communication to the device (Base Board or other hardware). This serial port is used by an *FT1.2* protocol (see *Chapter 8, FT1.2 Protocol*) which contains either the *BAOS Protocol* or the *EMI2* protocol.

  The BAOS Protocol (see *Chapter 9, BAOS Protocol*) is mainly used for reading and writing data point values, being notified of data point value changes, reading parameters and device parameter settings.

  The EMI2 protocol (see *Chapter 10, Message Protocol EMI*) is for advanced usage. It offers the possibility to craft own KNX telegrams for different *OSI*[1] communication layers.
- Optionally a programming (learning) mode key and LED.

[1] http://en.wikipedia.org/wiki/Iso_osi

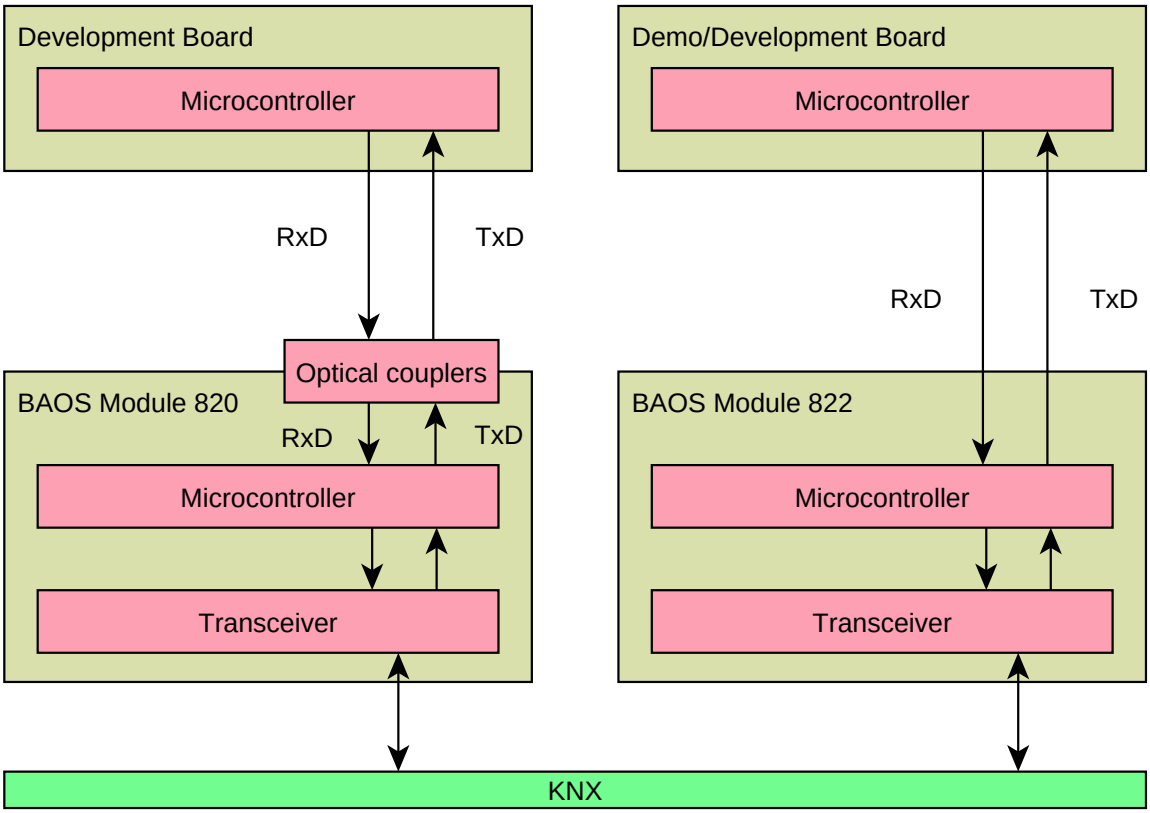


Figure 4.2. Modular overview of the hardware

The main difference between the 0820 and 0822 are the optical couplers. These couplers isolate the BAOS Module from the main device (Base Board) for electrical security reasons.

4.1. Pinning Of The BAOS Modules

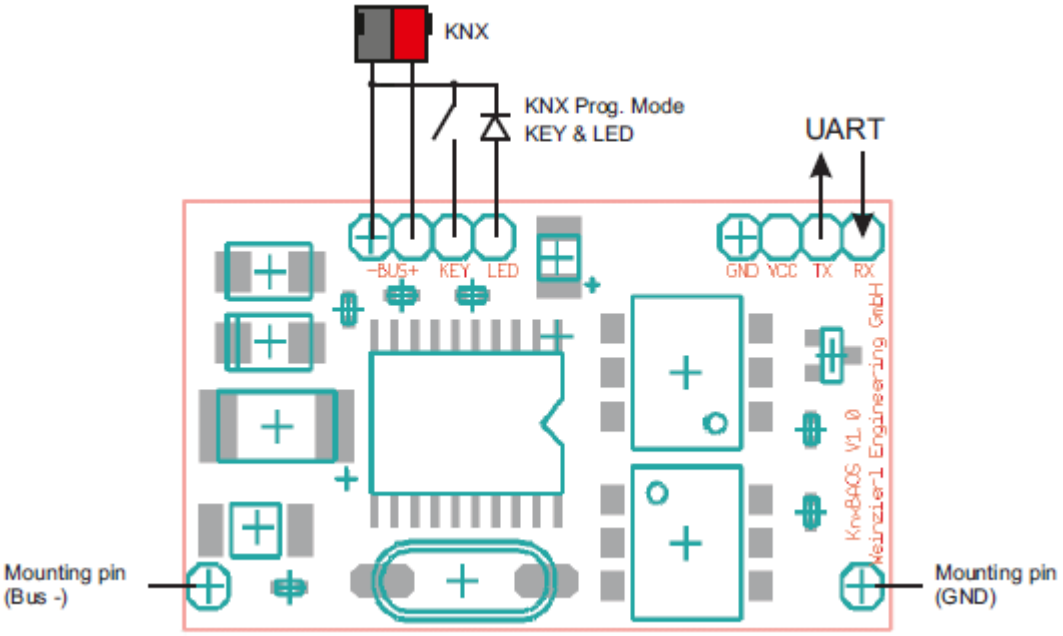


Figure 4.3. Pinning of the BAOS Module 0820

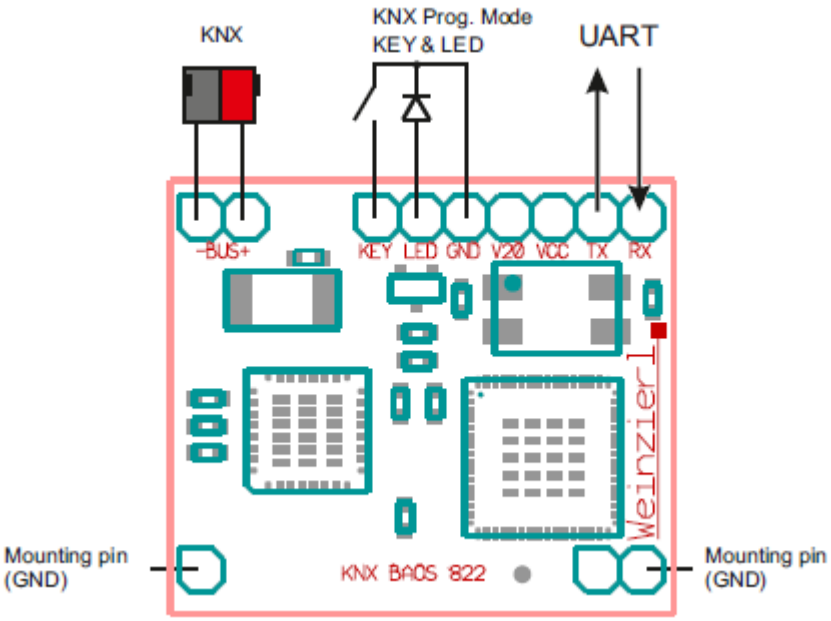


Figure 4.4. Pinning of the BAOS Module 0822

The pinning of the BAOS Modules shows the interfaces (KNX bus, UART) and the optional programming key & LED. The programming key is optional because the learning mode (programming mode for the individual address) can also be activated via the BAOS Protocol.

4.2. Hardware Requirements

The BAOS Module offers two interfaces and a learning key plus LED. To use the module the following requirements are recommended:

- Connection to the KNX bus: The BAOS Modules support a twisted pair (TP) KNX bus interface. This interface must be connected to the bus. Its nominal voltage is 29 volts in an operating range between 21 and 29 volts. Care must be taken about the polarity. The typical KNX plug is grey and red.

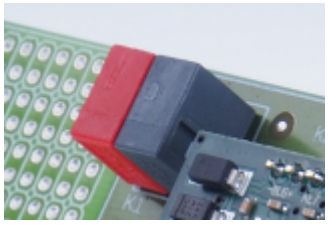


Figure 4.5. KNX plug

- Connection to the application hardware: The UART connects the application hardware with the BAOS Module via the ObjectServer/FT1.2. It is a serial port using 0V to 3-5 volts for the signals. The baud rate is 19200 per default and can be changed to 115200. Data bits are 8, even parity and 1 stop bit: 8e1.

Warning

Don't connect a RS-232 serial port directly to the pins of the 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.

- Optional: Learning key. A button should be connected to set the BAOS Module into programming mode for downloading an individual address. Alternatively this can be done by a BAOS Protocol service.
- Optional: LED for the learning key. The programming mode should be indicated by a red LED.

The BAOS Module 0820 has optical couplers which galvanically isolate the application hardware from the KNX bus. This is recommended for every device which is connected to other objects. This application hardware must supply VCC and GND to the BAOS Module, too. It is required for the application side of the optical couplers.

The BAOS Module 0822 supports the application hardware with VCC and GND. It will be powered by the KNX bus. This is only recommended for devices which have no other electrical connections (including ground).

4.3. Modular Overview Of The Firmware

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

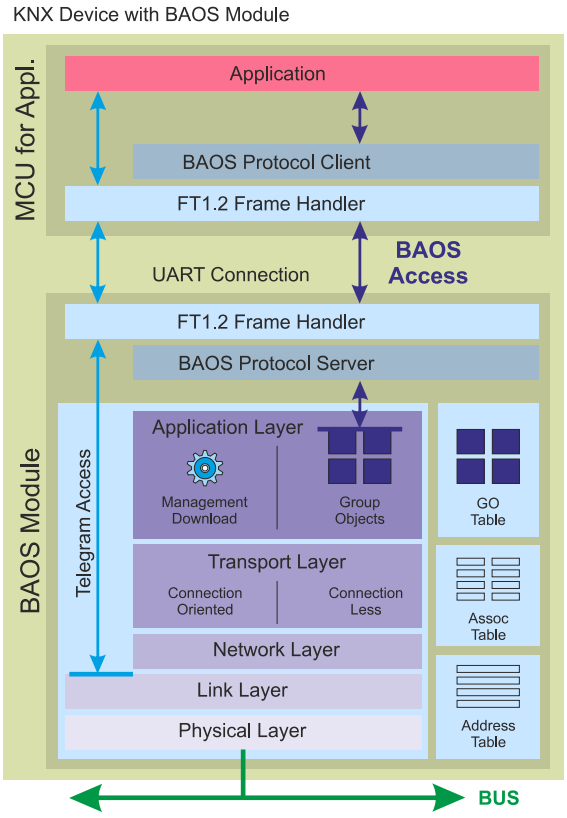


Figure 4.6. Modular overview of the firmware

The firmware contains a fully implemented, certified KNX communication stack, which conforms 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 250 data points and up to 250 parameters. They can be modified by the application and by KNX bus events. 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 microcontroller of the BAOS Module. You might render your device to be permanently unusable.

4.4. Other BAOS Devices

Other BAOS devices with different interfaces are also available.

Figure 4.7. BAOS Device 0870

The KNX Serial Interface BAOS 0870 is much like the BAOS Module except the serial interface does not go to another board. It is available at a D-sub connector. Using a level converter it is able to connect to an RS-232 PC interface. The communication protocol is the same: BAOS Protocol or EMI2 protocol, each embedded into an FT1.2 frame. For a quick start, a demonstration tool is available. More info is available at this *web page*[2].

Figure 4.8. BAOS Device 0772

The KNX IP BAOS 0772 offers an ethernet interface 10Base-T (LAN RJ-45) for connecting a PC or another device. It supports 1000 communication objects and thus implements the second version of BAOS Protocol to address these many objects. Up to 10 BAOS connections can be used simultaneously. More info is available at this *web page*[3].

[2] http://www.weinzierl.de/index.php/en/all-knx/knx-devices-en/knx-serial-baos-870-en
[3] http://www.weinzierl.de/index.php/en/all-knx/knx-devices-en/knx-ip-baos-772-en

Generic ETS Database

A generic database for ETS is included in this package. It can be used for developing an own application while the use of data points is not defined. The database containing the product KNX BAOS is stored in **`KnxBAOS_Demo/KnxBAOS_Demo_ETS_Projects/KNX_BAOS_82x_87x.knxprod`**. Use **Import Products** to load the database and use it in a project. All data points can be configured individually so the developer of the application can freely use the data points.

Per default all data points are disabled:

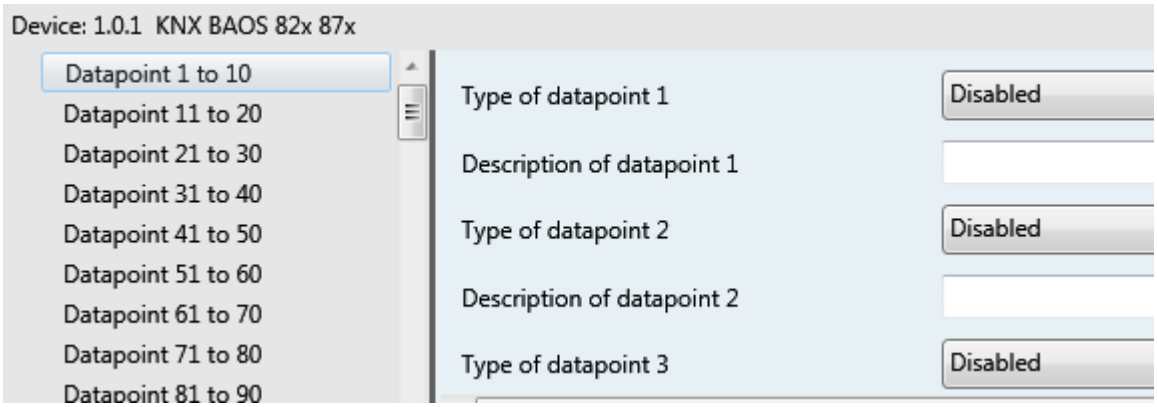


Figure 5.1. Generic ETS database

According to your programming enable and connect some data points. See *Section A.5, "Data Point Types"* for which data point types are available. The description of the data point is just a text field which is not downloaded into the KNX device. It stays in the ETS project just for information.

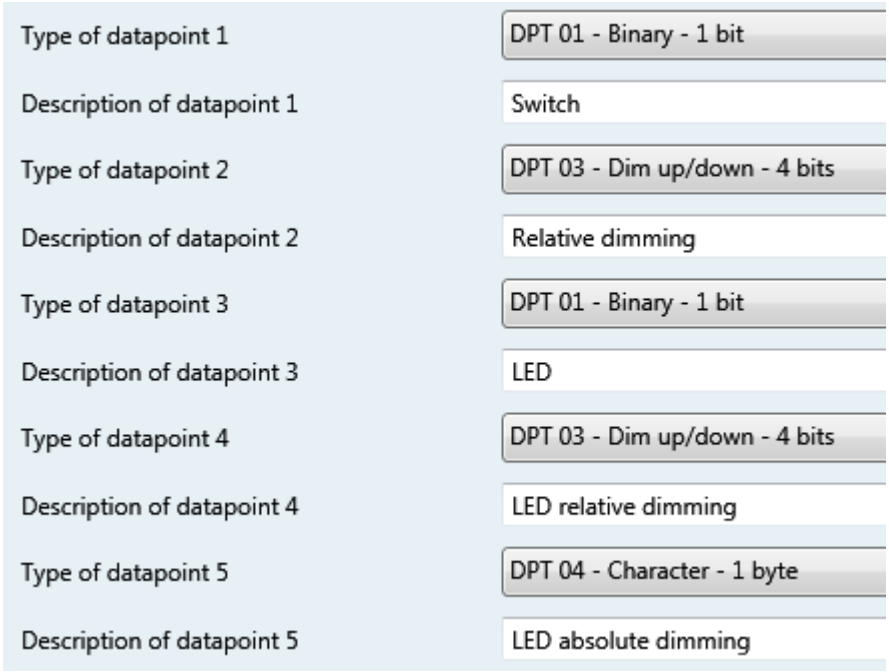


Figure 5.2. Some data points enabled

The application must know about the usage of the data points, so changing two parameters is necessary. To access the parameters from BAOS Module use the **GetParameterByte.Req** command. See **`KnxBAOS_Protocol_v1.pdf`** for more info.

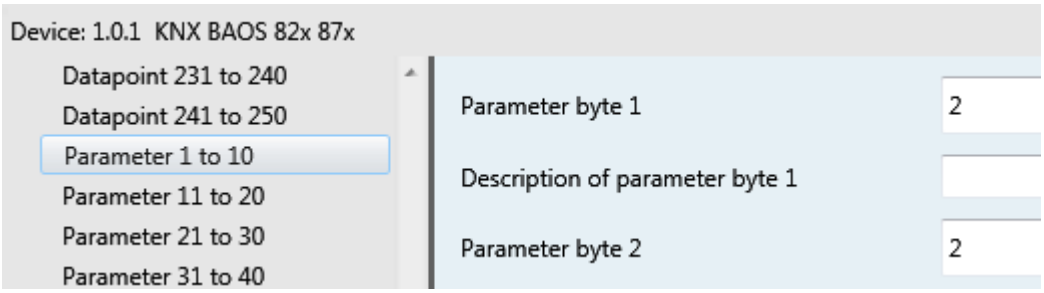


Figure 5.3. Some parameters configured

The data points can be linked as usual in ETS. Finally make a download.

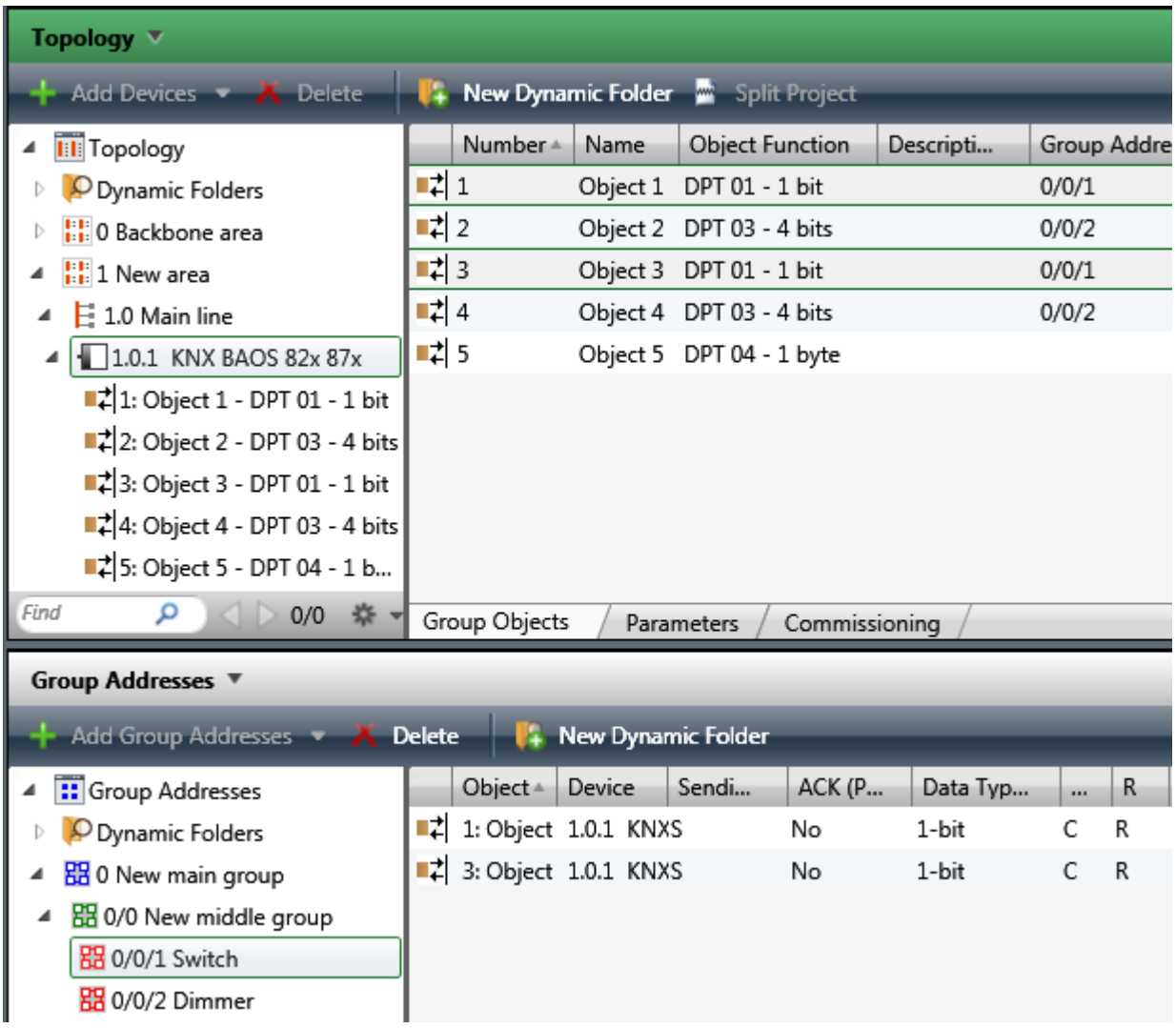


Figure 5.4. The data points linked to group addresses

The generic database has the advantage that all data points can be configured as any type. This is the usual way while development. For the common commissioning it is most inconvenient since the installer has to define the correct type for every data point. So it is necessary to create an individual database after finishing the development. 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 important.

To to this, it is necessary to register as member of the KNX group and obtain the *ETS Manufacturer Tool*[1] to create such a database. This database can be registered and a product is derived which can be used by the installer.

[1] http://www.knx.org/knx-tools/manufacturer-tool/description

The Application Framework

The anatomy of the demonstration application included in this package is a typical embedded software. The sources contain the following topics:

1. **`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 files. Some posix typedefs are defined there, but commented out since most compilers come up with these types. If not, just uncomment them:

```
typedef unsigned char bool_t;     // normally not part of posix
//typedef unsigned char uint8_t;
//typedef unsigned int uint16_t;
//typedef unsigned long uint32_t;
```

2. **`Main.c`** is the main entry. It initializes all components, like system and application, and enters a never ending main loop. This main loop contains all applications, each called there. Every application call should return as soon as possible so other applications (if there are more) are not blocked. A so called *state machine* for every application is a valid choice for a simple embedded software design.

3. **`App.c`** contains the main application. It handles the logic of the actuator, key events and the data points.

```
nEvent = AppKey_GetKeyEvent(keynumber);
```

This function returns the current key event. The key number is mapped in **`AppKey.h`**:

```
#define APP_KEY_COUNT       6           // Count of application keys
#define IS_KEY_PRESSED_0    GET_KEY_S5  // Port pin for channel 0
#define IS_KEY_PRESSED_1    GET_KEY_S6  // Port pin for channel 1
#define IS_KEY_PRESSED_2    GET_KEY_S7  // Port pin for channel 2
#define IS_KEY_PRESSED_3    GET_KEY_S8  // Port pin for channel 3
#define IS_KEY_PRESSED_4    GET_KEY_S9  // Port pin for channel 4
#define IS_KEY_PRESSED_5    GET_KEY_S10 // Port pin for channel 5
```

In this case the key S5 is mapped to key #0, S6 to key #1, and so on. The mapping can be changed freely as long as the number of used keys.

```
switch(nEvent)
{
    case KEY_EV_LONG:
        /* Do something */
        break;

    case KEY_EV_SHORT:
        /* Do something */
        break;

    case KEY_EV_RELEASE:
        /* Do something */
        break;

    case KEY_EV_NONE:
        break;
```

```
}
```

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. The time for a long press is defined in **`AppKey.h`** (unit milliseconds):

```
#define KEY_TIME_LONG 300
```

Every case contains one or more statements which check whether a certain data point is in use or not. This is controlled by the parameter bytes, which must be retrieved at the start of the application and at reset indication from the BAOS Module. The ETS sets the parameter bytes.

```
/// Parameter byte assignment
///
/// Parameter bytes used by this application:
///
/// PB#1: Configuration for ...
/// PB#2: Configuration for ...

enum PbUsage_tag
{
    PB_UNUSED     =  0,                 // PB#0 is never used
    PB_FIRST      =  1,                 // First used parameter byte
    PB_XXX1_TYPE  =  1,                 // PB#1
    PB_XXX2_TYPE  =  2,                 // PB#2
    PB_MAX                              // All remaining parameter bytes are not used
};


/// Retrieve parameters bytes we need.

void App_RetrieveParameterBytes(void)
{
    KnxBaos_GetParameterByte(PB_FIRST, PB_MAX - 1);
}


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

void App_HandleResetIndication(void)
{
    App_RetrieveParameterBytes();       // Request parameter bytes
}
```

The BAOS Module will resond to this call using **`App_HandleGetParameterByteRes()`**. This function stores the parameters we need:

```
/// 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] nStartByte Start byte from telegram
/// @param[in] nByte Current nByte from telegram
/// @param[in] nIndex Current number of byte

void App_HandleGetParameterByteRes(
```

```
    uint8_t nStartByte, uint8_t nByte, uint8_t nIndex)
{

    switch(nStartByte + nIndex)                 // Parameter byte number
    {

        case 1:                                 // Get configuration 1
            m_nConfig1 = nByte;
            break;

        case 2:                                 // Get configuration 2
            m_nConfig2 = nByte;
            break;

        default:                                // Ignore all other parameter bytes
            break;
    }

}
```

Every data point which is connected can now be used. All others are skipped.

```
if((m_nConfig1 == 1)                            // Data point configured as switch?
    || (m_nConfig1 == 2))                       // or as dimmer?
{
    /* act on data point #1 as switch */
}

if(m_nConfig1 == 2)                             // Data point configured as dimmer?
{
    /* act on data point #2 as relative dimmer */
}
```

In this case data points #1 and #2 are used as dimmer (switching and relative dimming) if **`m_nConfig1`** is 2. If it is 1 data point #2 is not used, but data point #1 as switch only. If the data point is configured by ETS a KNX message will be sent:

```
nValue = LED_OFF;
KnxBaos_SendValue(DP_SWITCH_O, DP_CMD_SET_SEND_VAL, 1, &nValue);
```

Furthermore plenty callback functions are defined in **`App.c`**. For Example:

```
void App_HandleDatapointValueInd(
    uint8_t nStartDatapoint, uint8_t nDpId, uint8_t nDpState,
    uint8_t nDpLength, uint8_t* pData)
{
    // Implement this
}
```

All relevant parameters from a received KNX message are extracted and provided in this function. Some function are already implemented.

Note

For more information about the parameters and the object server protocol, see **KnxBAOS_Protocol_v1.pdf** which is also included in this distribution.

4. **AppKey.c** is the driver for handling the push buttons on the Development/Demo Board. It also handles its debouncing, long/short presses and releases.

5. **AppBoard.c** initializes the ports of our microcontroller at the Development/Demo Board. See **AppBoard.h** which port is connected to the components on the board.

6. **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 (100%) 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
```

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

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

```
uint16_t KnxTm_GetTimeMs(void);         /* 0x0000 - 0xffff */
```

Note

If a timer reaches its upper value, it resets to 0 and counts again. So be aware you might get negative results calculating two time stamps if an overflow occurred.

Warning

The timer does not start immediately after a reset or power up. The application initializes the timer while booting up.

To perform a delay of 200 milliseconds, do the following:

```
while(TRUE) /* This is the main loop */
{
    static uint16_t nTimeStamp;
    static enum eState_t nState = IDLE;

    switch(nState)
    {
        case IDLE:
            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;
    }

}
```

The following is a busy delay. It can be done for brief periods while initialization. It is not recommended to use this in the main loop since it consumes CPU time. Though if we have only one application it doesn't matter.

```
KnxTm_SleepMs(5);
```

9. **`KnxProgMode.c`** handles the programming mode of the BAOS Module. It is a polling service, which is called in certain periodes, like every second. It calls the BAOS function to get the state of the programming mode and visualizes its state at the red LED of the Development Board.

   This service is not available at the Demo Board.

10. **`KnxBaos.c/KnxBaosHandler.c`** contains the implementation of the object server protocol, which is the core protocol of the BAOS. It mainly contains routines for sending and receiving (**`KnxBaosHandler.c`**) telegrams.

11. **`KnxBuf.c`** handles a send and receive buffer for FT1.2.

12. **`KnxFt12.c`** handles the FT1.2 (PEI10) protocol which is used by the communication with the BAOS Module.

13. **`KnxSer.c`** is the low level serial driver used by the communication with the BAOS Module.

14. **config.h** can contain some defines and macros for various purposes. Currently only debugging configuration is implemented:

```
#define _DEBUG 1
```

    Normally this line is disabled, since Atmel Studio can maintain this via its own generated **Makefile**. It can be overridden here if necessary. Other defines can also be inserted here.

For a deeper exploration of the demo applications see **Documents/BAOS_DevBoardSoftware_API.chm** also included in this package.

The main application (including all timer and BAOS communication routines) is frequently interrupted by 2 interrupts: timer and communication. The communication interrupt occurs for every sent or received character at the UART port (normally BAOS). The timer interrupts the Timer handler every millisecond which manages a global counter and the timeouts for the FT1.2 handler.

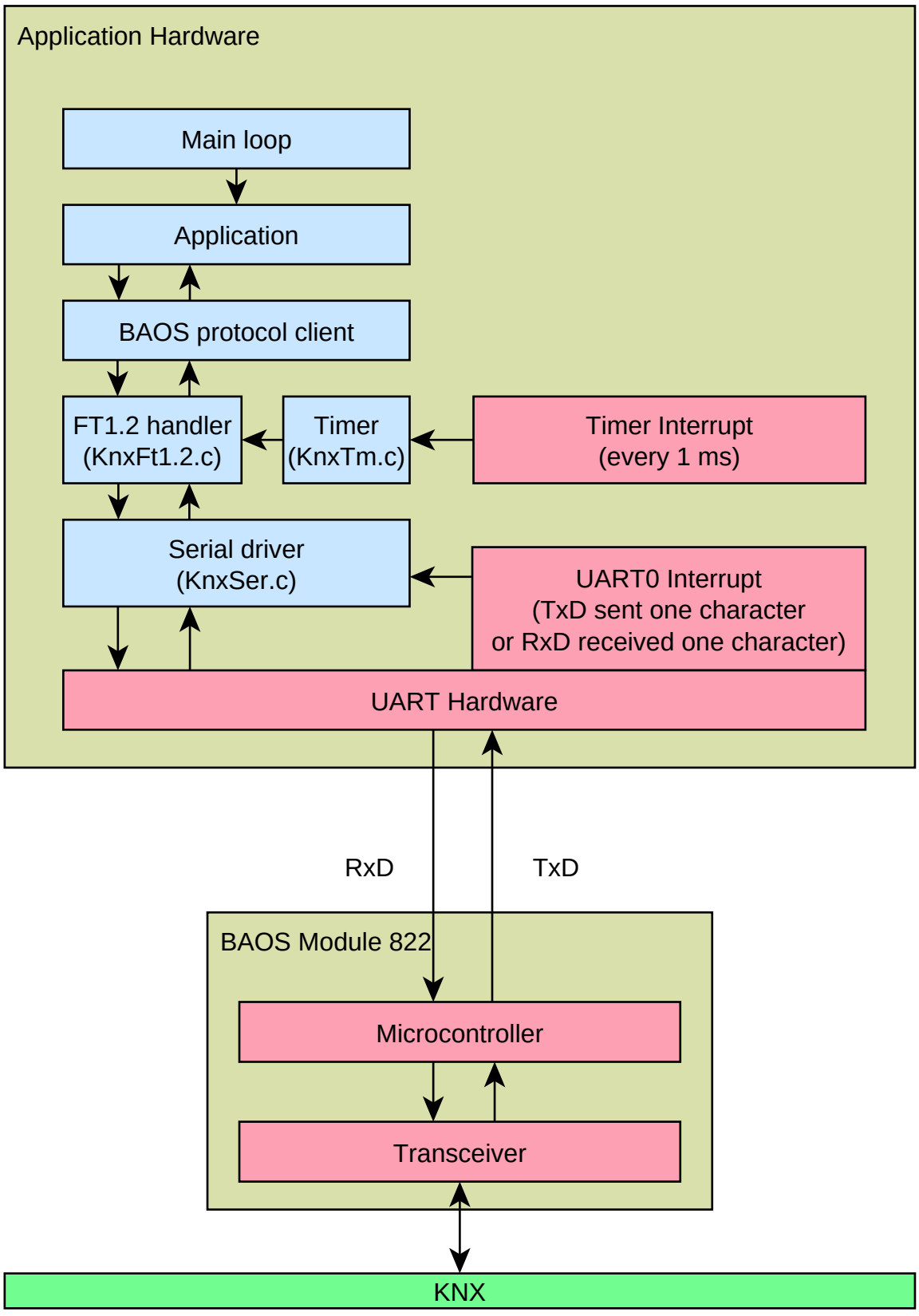


Figure 6.1. Overview of interrupts

6.1. Creating Own Applications

In order to make your own applications for the Development/Demo Board, the file **App.c** is the central place. The function **App_Init()** does the initialization. The function **App_Main()** is frequently called by **main()**. Here, in the function **App_Main()** has to be called. 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_LED1** from **AppBoard.h** to control the LED.

With the function **KnxBaos_SendValue(uint16_t nDataPoint_Id, uint8_t nDataPoint_Command, uint8_t nDataPoint_Length, uint8_t* pDataPoint_Value)** the data points can be accessed. The data points which were received by BAOS are handled in functions like **App_HandleXXX()**. They will be called automatically after data reception and analysis.

6.2. Common Cases

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

6.2.1. 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 to data point #2, which has one byte size (1). This value change is sent to the KNX bus.

Sending a value of more bytes might require a conversion of endianess since network communication is always defined in big endian whereas some microcontrollers use little endian. A 4 byte value 0x11223344 is stored in big endian like this: 0x11223344. The same value is stored in little endian like this: 0x44332211. So we need a conversion routine for values larger than 1 byte.

```
// Send a 32 bit value with correct endianess:
float fValue = 1.2f;
KnxBaos_SendValue(3, DP_CMD_SET_SEND_VAL, sizeof(fValue), KnxBaos_Host2Net32(&fValue));
```

The routines **KnxBaos_Host2Net32()** and **KnxBaos_Host2Net16()** convert 32 bit (4 byte), respectively 16 bit (2 byte) values to big endianess. These values can be send to the KNX network.

Received values can be converted back with **KnxBaos_Net2Host16()** and **KnxBaos_Net2Host32()**.

If the host microcontroller is operating in big endian mode a conversion will not be done by these routines since it is not necessary.

6.2.2. Get Data Point Value

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

```
```

```
void App_HandleDatapointValueInd(
    uint8_t nStartDatapoint, uint8_t nDpId, uint8_t nDpState,
    uint8_t nDpLength, uint8_t* pData)
{

    switch(nDpId)
    {
        case DP_LED_SWITCH_I:               // LED object selected

            if(*pData == LED_ON)            // Switch on
            {
                AppDim_OnSwitch(TRUE);
            }
            else if(*pData == LED_OFF)      // Switch off
            {
                AppDim_OnSwitch(FALSE);
            }

            break;
    }
}
```

If the data point ID matches (**DP_LED_SWITCH_I**) the new value pointed by **pData** is analyzed (**LED_ON** or **LED_OFF**) and we handle accordingly: switch on or off an LED via **AppDim_OnSwitch()**.

Note

The terms **data point** and **data point ID** mean the same. There is no difference.

If the value is more than 1 byte the endianess problem must also be handled:

```
void App_HandleDatapointValueInd(
    uint8_t nStartDatapoint, uint8_t nDpId, uint8_t nDpState,
    uint8_t nDpLength, uint8_t* pData)
{

    switch(nDpId)
    {
        case DP_TEMPERATURE_I:              // Temperature object selected

            // Receiving a 32 bit float value
            float fValue = *(float *)KnxBaos_Net2Host32(pData);
            App_ShowTemperature(fValue);
            break;
    }
}
```

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

```
KnxBaos_GetDpValue(2, 1);
```

This request the current value of data point #2. The value 1 is the number of data points. In this case we want to know only data point #2. After sending this request we will be informed by **App_HandleGetDatapointValueRes()**:

```
void App_HandleGetDatapointValueRes(
    uint8_t nStartDatapoint, uint8_t nDpId, uint8_t nDpState,
    uint8_t nDpLength, uint8_t* pData)
{

    if(*pData == LIGHT_ON)
    {
        AppLight_SwitchOn();
    }
    else
    {
        AppLight_SwitchOff();
    }

}
```

The parameters of this routine are the same as in **`App_HandleDatapointValueInd()`**.

6.2.3. Get Parameter Byte

The ETS can set some parameters while download. These parameters can be used to configure some values. E. g. set a time out, a lighting value, 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, starting at byte #1. After sending this request we will be informed by **`KnxBaos_OnGetParameterByteRes()`**:

```
void App_HandleGetParameterByteRes(
    uint8_t nByteNo, uint8_t nByte)
{

    if(nByteNo == MAX_TEMP_PARAMETER_INDEX)
    {
        App_SetMaxTemperature(nByte);
    }

}
```

This routine is called for every byte we requested. The value is delivered in **`nByte`**. **`nByteNo`** is the current byte number of the request.

6.3. Special Cases

Some special cases are 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.

6.3.1. Write Value To A Group Object

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 receiver(s). It is also possible to write to such a group object without any data point involved. In this case we have to construct an EMI2 message (see *Chapter 10, Message Protocol EMI* for more information).

As an example if a group object **2/5/5** is configured as absolute dimming value (1 byte) to a dimmer input, we can write a value 0-255 to change the light.

```
uint8_t aBuffer[] =
{
    0x0A,                   // Length of this array
    0x11,                   // EMI2 service code: L_Data.req
    0xBC,                   // EMI2 KNX control field
    0x13, 0x03,             // EMI2 source address 1.3.3
    0x15, 0x05,             // EMI2 group address 2/5/5
    0xE2,                   // EMI2 code + length 2 bytes (including APCI)
    0x00,                   // EMI2 TPCI + APCI
    0x80,                   // EMI2 APCI
    0x78                    // EMI2 data byte (= 120)
};

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

This writes the value 120 to the group object **2/5/5**.

6.3.2. Read Value From A Group Object

Reading a value from a group object is more complicated than just writing it. The BAOS stack must be in a special mode which does not filter messages from the KNX bus. To do a read the following must be done:

1. Switch the BAOS stack to a special mode: route all telegrams from the link layer directly to the application.

```
uint8_t aBuffer[] =
{
    0x07,                          // Length of this array
    0xA9,                          // EMI2 service code: PEI_Switch.req
    0x00,                          // EMI2 system status
    0x18, 0x34, 0x56, 0x78, 0x9A // EMI2 switch to link layer
};

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

2. Even with the re-routing of all telegrams from the link layer to the application the address table still filters all telegrams which are not for the BAOS Module. Only telegrams configured by the ETS for the BAOS Module pass through. To disable this the address table must be switched off by writing a length of 0. But first save the old length value to write it back later.

```
uint8_t aBuffer[] =
{
    0x04,                          // Length of this array
    0xAC,                          // EMI2 service code: PC_Get_Value.req
    0x01,                          // EMI2 data length (1 byte)
    0x40, 0x00                     // EMI2 address 4000
};

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

This requests one byte from address 0x4000 which is the current length of the address table. 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

        [...]

        if(bAccept == TRUE)             // Do we accept the telegram?
        {
            [...]
        }
        else
        {
            // Here we can handle all received telegrams in
            // m_pBaosRcvBuf[]. m_pBaosRcvBuf[0] stores the length
            // the following bytes are EMI2 telegrams.
        }

    }

    [...]
}
```

The buffer **`m_pBaosRcvBuf[]`** will contain this response (the length byte at index 0 is omitted):

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

Address 0x4000 contains 0x15. This value must be saved for later.

3. Disable the address table by writing length = 0:

```
uint8_t aBuffer[] =
{
    0x05,                       // Length of this array
    0xA6,                       // EMI2 service code: PC_Set_Value.req
    0x01,                       // EMI2 data length
    0x40, 0x00,                 // EMI2 address 4000
    0x00                        // EMI2 data value 0
};

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

This writes 0 to address 0x4000, which disables the address table.

4. Everything is prepared, now read the group object **`2/5/5`**. Our own address is **`1.3.3`** and we request to read one byte:

```
uint8_t aBuffer[] =
{
    0x09,               // Length of this array
    0x11,               // EMI2 service code: L_Data.req
    0xBC,               // EMI2 KNX control field
    0x13, 0x03,         // EMI2 source address 1.3.3
    0x15, 0x05,         // EMI2 group address 2/5/5
    0xE1,               // EMI2 code + length 2 bytes (including APCI)
    0x00,               // EMI2 TPCI + APCI
```

```
    0x00                    // EMI2 APCI
};

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

Again, the responses (this time there will be two) can be handled in **KnxBaos_Process()**, File: **KnxBaos.c**. **KnxBaos_Process()** will be called for every response. The buffer **m_pBaosRcvBuf[]** contains these values (the length byte is omitted):

```
2E BC 13 03 15 05 E1 00 00                                  L_Data.con
29 BC 12 03 15 05 F2 00 40 30                               L_Data.ind
```

The first response is a confirmation. I. e. a copy of our request. The second response (GrpValResp) contains the value 0x30 from the group object **2/5/5**.

Note

More information about EMI2 telegrams can be found in *Chapter 10, Message Protocol EMI* and in **KNX System Specifications/03_06_03 EMI_IMI** available at the *KNX Specifications page*[1].

5. To restore the BAOS stack to normal mode (BAOS enabled), the address table must be restored and the PEI switch set to default.

```
uint8_t aBuffer[] =
{
    0x05,                       // Length of this array
    0xA6,                       // EMI2 service code: PC_Set_Value.req
    0x01,                       // EMI2 data length
    0x40, 0x00,                 // EMI2 address 4000
    0x15                        // EMI2 data value 0x15
};

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

The data value 0x15 is, of course, the recently saved length of the address table.

6. Finally switch off the routing from the link layer to the application (PEI switch).

```
uint8_t aBuffer[] =
{
    0x07,                       // Length of this array
    0xA9,                       // EMI2 service code: PEI_Switch.req
    0x00,                       // EMI2 system status
    0x12, 0x34, 0x56, 0x78, 0x9A // EMI2 switch back to default mode
};

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

[1] http://www.knx.org/knx-en/knx/technology/specifications/index.php

Connecting PC Via BAOS Interface

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. This PC must be connected to the BAOS Module the way the embedded micro controller is. To do this connect the RS-232 interface of the PC to the Development Board and set the jumpers accordingly. Setting the necessary jumpers at the Development Board enables the direct connection of the BAOS Module to the PC. This enables the BAOS communication directly to the PC. A demonstration client is included for this purpose in this package. This *Demo Client* is located in **`Tool_BAOS_DemoProgram`** and must be installed to use it. Just run the installer file and follow the instructions.

Connect the serial port of the Development Board to the PC and set the appropriate jumpers (See *Section 3.4.2, "Jumper Usages"*).

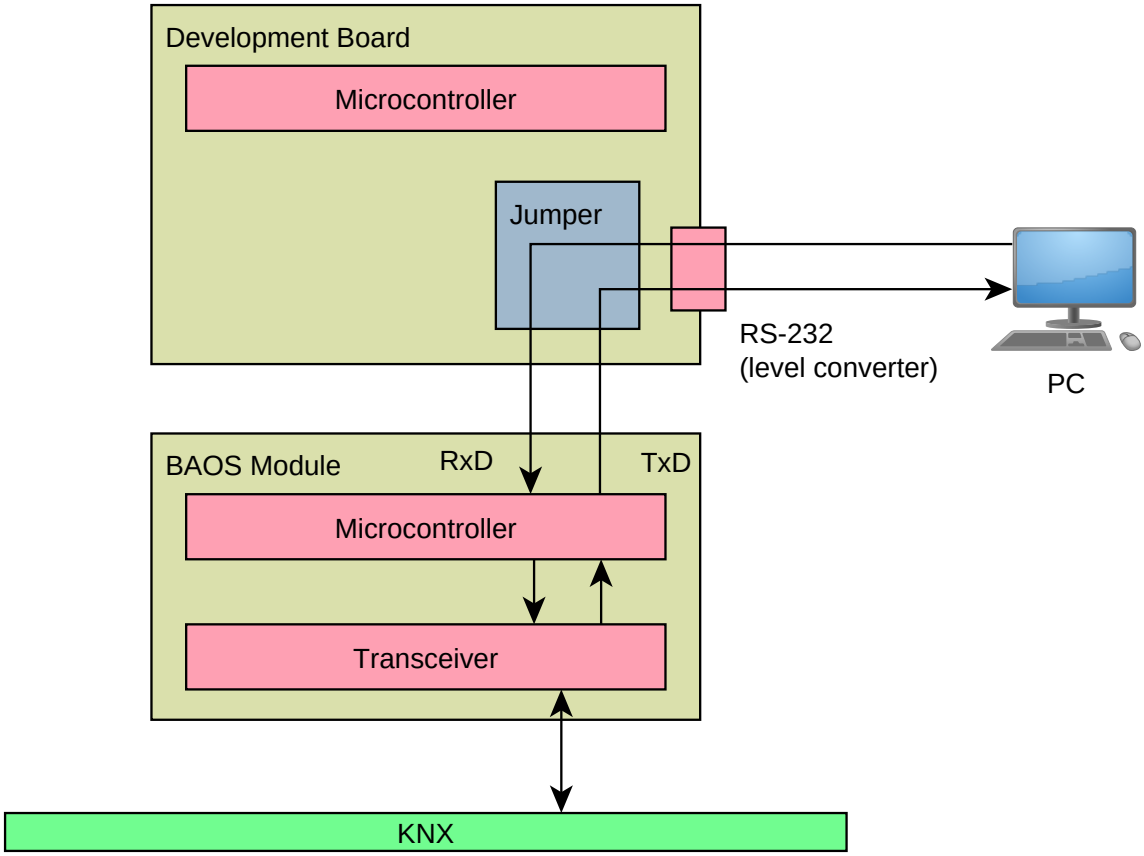


Figure 7.1. PC And BAOS

To use RS-232-1 disconnect both **`BAOS Tx`** and **`Rx`** jumpers and connect them to **`RS-232-1 BAOS Tx`** and **`Rx`**.

Start the Demo Client, normally installed in **`C:/Program Files (x86)/Weinzierl Engineering GmbH/KnxBAOS DemoClient for protocol 1.x`**, use **RS232**, appropriate **COM** port and **19200** baud. If it successfully **Open**s the port, choose a telegram and **Send** it. A response will be displayed in the Telegram Tree.

Don't forget to reset the jumpers after using the Demo Client.

FT1.2 Protocol

The communication between the Base Board and the BAOS Module uses the ObjectServer or the EMI2 protocol. This protocol is encapsulated in the FT1.2 protocol for reliability reasons. The typical device architecture looks like this:

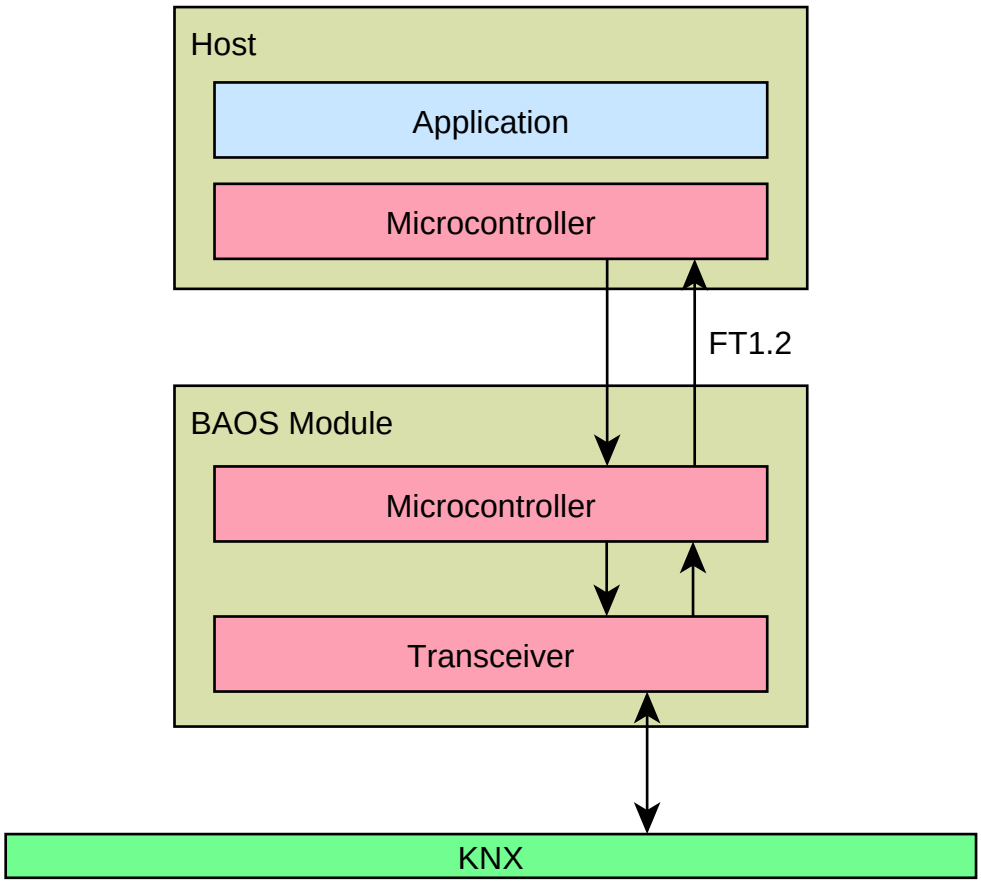


Figure 8.1. Typical device architecture

8.1. General

The FT1.2 protocol is based on the international standard IEC 0870-5-1 and 0870-5-2. An asymmetric transmission procedure is used. I. e. the host initiates a message transfer and the BAOS Module sends a response. The protocol is restricted to point-to-point (no address field) communication.

8.2. Physical

8.2.1. Interface

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

- **RxD**: Received data
- **TxD**: Transmit data
- **GND**: 0 V Ground

- additional for 0820: **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.

8.2.2. Timings

The timing of the FT 1.2 communication between the application and the BAOS Module are shown in the next figure. The application sends an FT 1.2 frame to the BAOS Module which acknowledges it. After a while the module sends a response frame to the application which also acknowledges it.

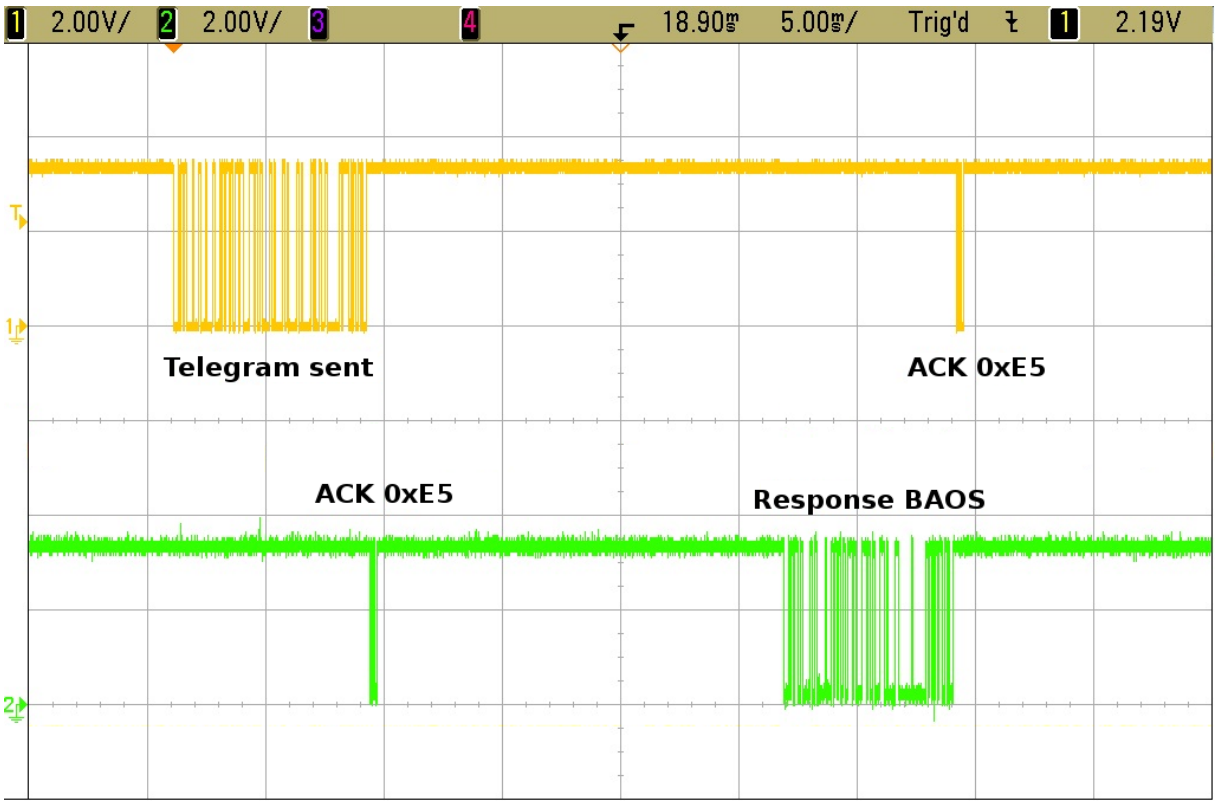


Figure 8.2. BAOS communication timing

8.3. FT 1.2 Frame Format

The FT 1.2 protocol ensures data integrity by using a defined header, a checksum and a stopping character. There are two frame variants: a fixed frame for reset requests and a variable frame length for containing data.

An FT 1.2 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

Figure 8.3. FT 1.2 reset frame

To request a reset of internal counters and states, send an FT 1.2 reset request:

```
10 40 40 16                                                    RESET_REQ
```

Note

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

An FT 1.2 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
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							

Figure 8.4. FT 1.2 data frame

The **FT 1.2 control field** has the following information:

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

  1 = Host (application) to BAOS Module

  0 = BAOS Module to Host (application)

BAOS Protocol

9.1. BAOS Frame

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

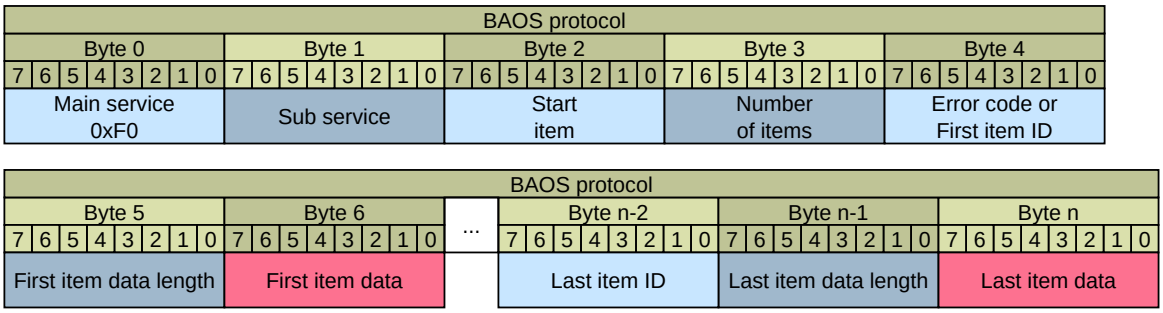


Figure 9.1. General BAOS format

Some important services and their responses:

- **`GetServerItem.Req`**

Offset	Field	Size	Value	Description
+0	MainService	1	**`0xF0`**	Main service code
+1	SubService	1	**`0x01`**	Subservice code
+2	StartItem	1		ID of first item
+3	NumberOfItems	1		Maximal number of items to return

`GetServerItem.Res`

Offset	Field	Size	Value	Description
+0	MainService	1	**`0xF0`**	Main service code
+1	SubService	1	**`0x81`**	Subservice code
+2	StartItem	1		As in request
+3	NumberOfItems	1		Number of items in this response
+4	First item ID	1		ID of first item
+5	First item data length	1		Data length of first item
+6	First item data	1 - 255		Data of first item
...	...		**`...`**	...
+n-2	Last item ID	1		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 08 01                                 GetServerItem.Req
BAOS:         F0 81 08 01 08 06 11 22 33 44 55 66         GetServerItem.Res
```

- **`GetDatapointValue.Req`**

Offset	Field	Size	Value	Description
+0	MainService	1	**`0xF0`**	Main service code
+1	SubService	1	**`0x05`**	Subservice code
+2	StartDatapoint	1		ID of first data point
+3	NumberOfDatapoints	1		Maximal number of data points to return

`GetDatapointValue.Res`

Offset	Field	Size	Value	Description
+0	MainService	1	**`0xF0`**	Main service code
+1	SubService	1	**`0x85`**	Subservice code
+2	StartDatapoint	1		As in request
+3	NumberOfDatapoints	1		Number of data points in this response
+4	First DP ID	1		ID of first data point
+5	First DP state/length	1		State/length byte of first data point
+6	First DP value	1 - 14		Value of first data point
...	...		**`...`**	...
+n-2	Last DP ID	1		ID of last data point
+n-1	Last DP state/length	1		State/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).

```
APPLICATION:  F0 05 01 02                                 GetDatapointValue.Req
BAOS:         F0 85 01 02 01 01 00 02 02 88 99            GetDatapointValue.Res
```

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

Offset	Field	Size	Value	Description
+0	MainService	1	**`0xF0`**	Main service code

Offset	Field	Size	Value	Description
+1	SubService	1	**0xC1**	Subservice code
+2	StartDatapoint	1		ID of first data point
+3	NumberOfDatapoints	1		Number of data points in this indication
+4	First DP ID	1		ID of first data point
+5	First DP state/length	1		State/length byte of first data point
+6	First DP value	1 - 14		Value of first data point
...	...		**...**	...
+n-2	Last DP ID	1		ID of last data point
+n-1	Last DP state/length	1		State/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 an one byte value (= 0x55).

```
BAOS:         F0 C1 03 01 03 81 55                        DatapointValue.Ind
```

- **SetDatapointValue.Req**

Offset	Field	Size	Value	Description
+0	MainService	1	**0xF0**	Main service code
+1	SubService	1	**0x06**	Subservice code
+2	StartDatapoint	1		Lowest ID of data points to set
+3	NumberOfDatapoints	1		Number of data points to set
+4	First DP ID	1		ID of first data point
+5	First DP cmd/length	1		Command/length byte of first data point
+6	First DP value	1 - 14		Value of first data point
...	...		**...**	...
+n-2	Last DP ID	1		ID of last data point
+n-1	Last DP cmd/length	1		Command/length byte of last data point
+n	Last DP value	1 - 14		Value of last data point

SetDatapointValue.Res

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

Offset	Field	Size	Value	Description
+3	NumberOfDatapoints	1	**0x00**	
+4	ErrorCode	1	**0x00**	

Example: Set value of data point #5

Data point #1 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 05 01 05 31 01                    SetDatapointValue.Req
BAOS:         F0 86 05 00 00                          SetDatapointValue.Res
```

- **GetParameterByte.Req**

Offset	Field	Size	Value	Description
+0	MainService	1	**0xF0**	Main service code
+1	SubService	1	**0x07**	Subservice code
+2	StartByte	1		Index of first byte
+3	NumberOfBytes	1		Maximal number of bytes to return

GetParameterByte.Res

Offset	Field	Size	Value	Description
+0	MainService	1	**0xF0**	Main service code
+1	SubService	1	**0x87**	Subservice code
+2	StartByte	1		As in request
+3	NumberOfBytes	1		Number of bytes in this response
+4	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 08 08                             GetParameterByte.Req
BAOS:         F0 87 08 08 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_v1.pdf** which is also included in this package.

9.2. BAOS Frame Embedded In An FT 1.2 Frame

The BAOS Protocol is encapsulated into the FT 1.2 frame for data integrity.

An FT 1.2 frame using the BAOS Protocol looks as follows:

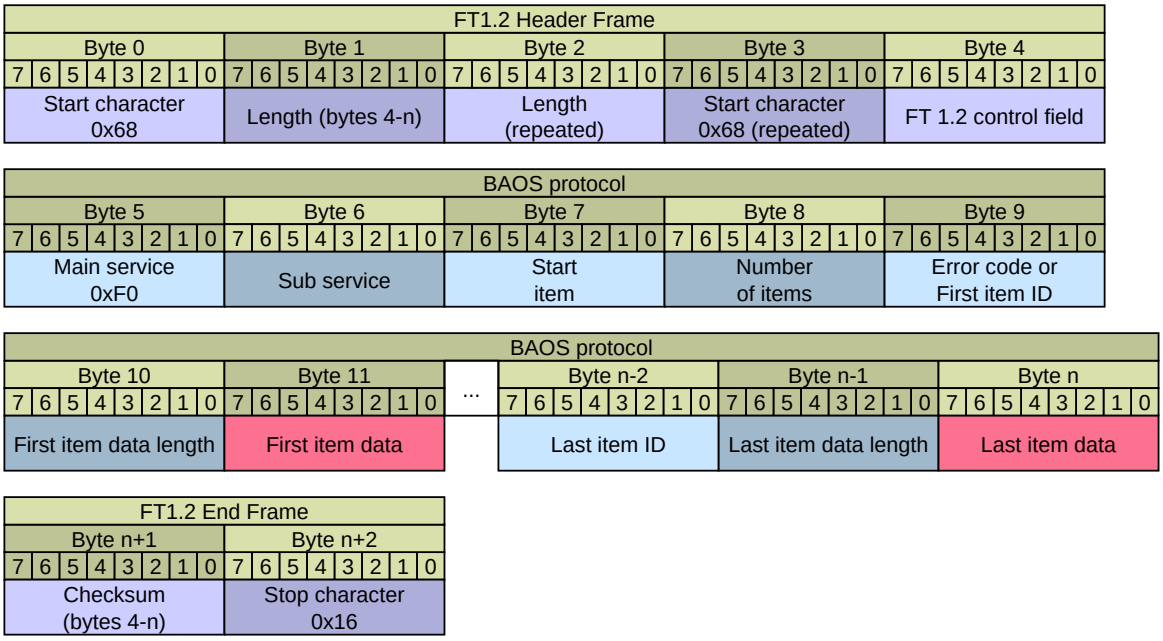


Figure 9.2. BAOS frame embedded in an FT 1.2 frame

The FT 1.2 control field bits, see *Section 8.3, "FT 1.2 Frame Format" [49]*.

Message Protocol EMI

10.1. EMI2 Protocol

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:

1. **`Telegram access via the network layer (NL).`** This access leaves the group address filtering active. All telegrams are treated in the normal way like in the BAOS Protocol access. The services for this functionality are part of the KNX standard (*EMI2* - External Message Interface version 2).

2. **`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.

The EMI2 telegram format contains the KNX telegram as follows:

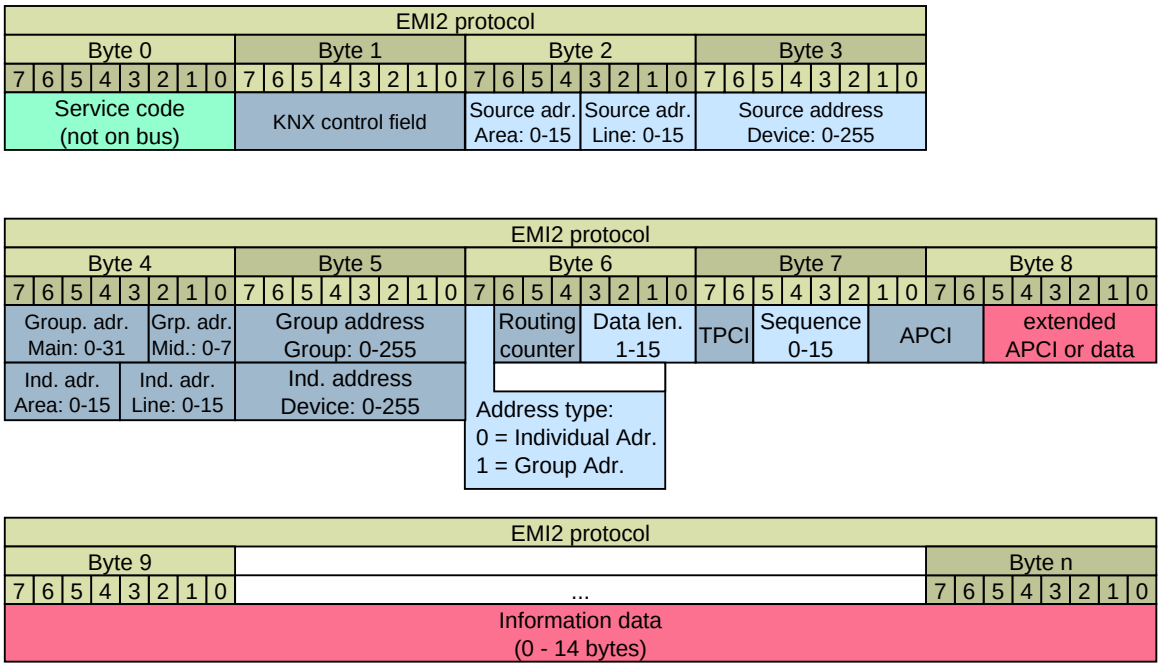


Figure 10.1. EMI2 telegram format

The KNX control field bits, see *Section A.2, "KNX Twisted Pair Telegrams" [71]*.

10.2. EMI2 Frame Embedded In An FT 1.2 Frame

The EMI2 protocol is used to generate own KNX telegrams and exchange them with the corresponding stack layer. It is also encapsulated into the FT 1.2 frame for data integrity.

An FT 1.2 frame using the EMI2 protocol looks as follows:

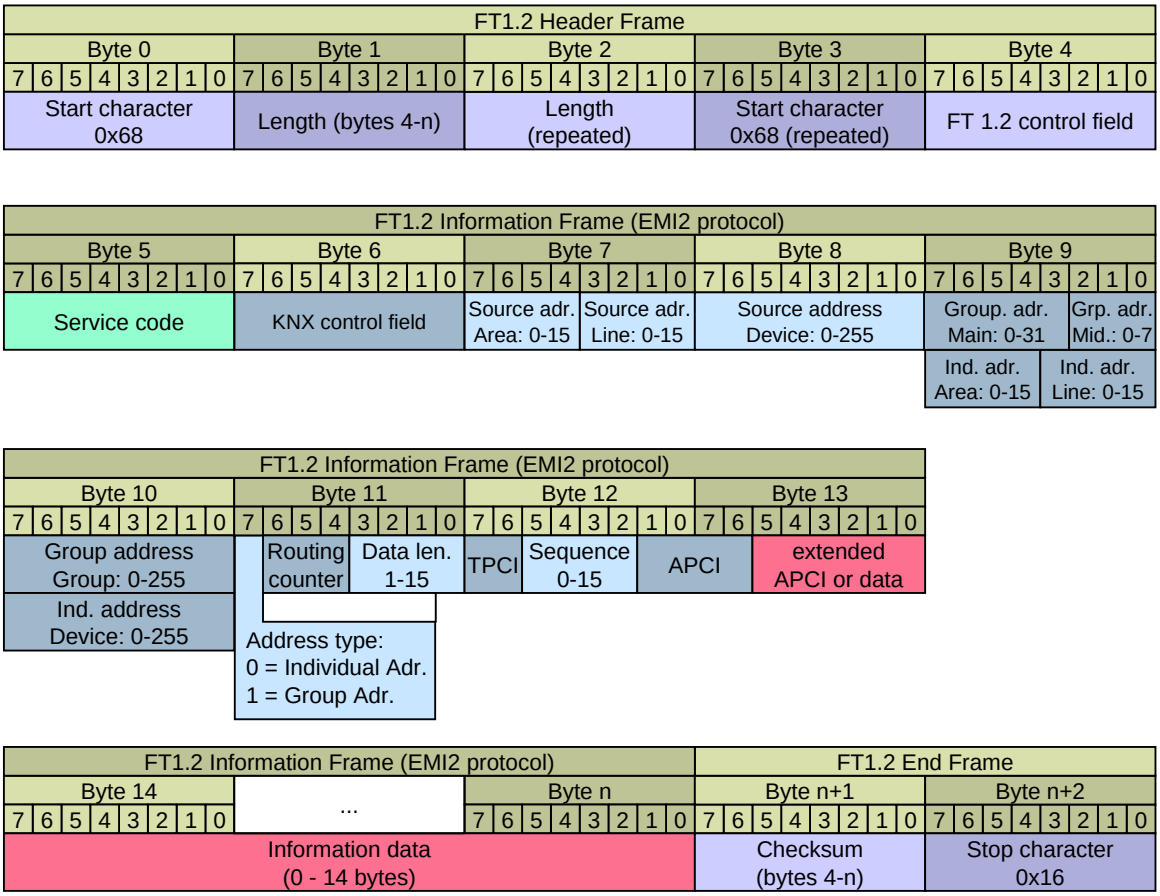


Figure 10.2. EMI2 frame embedded in an FT 1.2 frame

The FT 1.2 control field bits, see *Section 8.3, "FT 1.2 Frame Format" [49]*.

The KNX control field bits, see *Section A.2, "KNX Twisted Pair Telegrams" [71]*.

For more information about the FT1.2 protocol see **KnxBAOS_Protocol_v1.pdf**, Appendix D which is also included in this package.

10.3. Access To The Network Or Link Layer

To access the network or link layer a certain message must be sent to activate the access.

- **Network layer**:

```
PC:   A9 00 12 84 56 78 9A                                  PEI_SWITCH_req
```

- **Link layer**:

```
PC:   A9 00 18 34 56 78 9A                                  PEI_SWITCH_req
```

- **Disable access**:

```
PC:   A9 00 12 34 56 78 9A                                    PEI_SWITCH_req
```

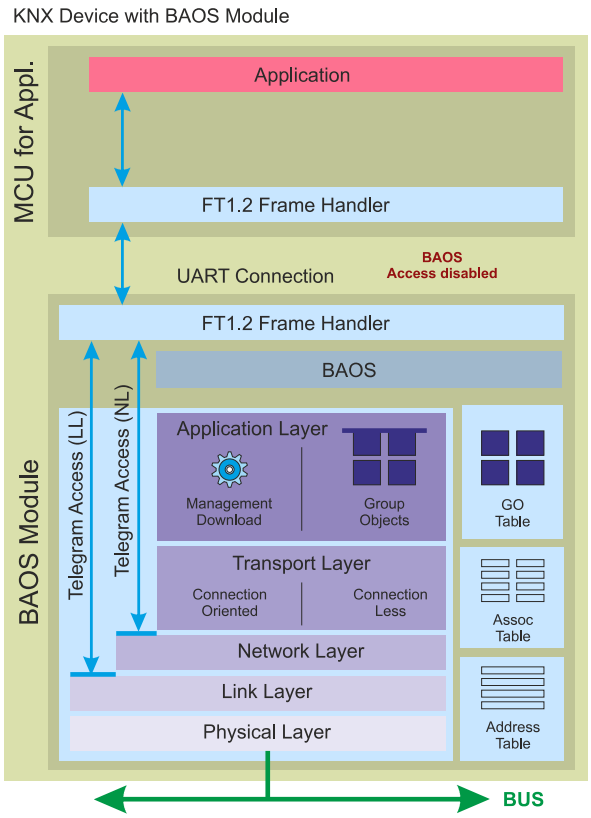


Figure 10.3. Telegram access to the network or link layer

10.4. Group Telegram Communication On Network Layer

The intended usage of BAOS is an interface to KNX twisted pair on data point level. Nevertheless it is possible to send and to receive *group telegrams* directly without the use of the object server.

First we have to switch on the correct internal message routing via PEI message (PC_SET_VAL_req). Then we can send/receive some telegrams (N_GROUP_DATA_req/N_GROUP_DATA_con/ N_GROUP_DATA_ind). At last we disable the PEI routing (PC_SET_VAL_req).

To receive any group messages send the following EMI2 messages:

Disable group address filter of module

```
PC:   A6 01 40 00 00                                          PC_SET_VAL_req
```

This means to write 0x00 at location 0x4000 which holds the address table length. This is a non volatile value, so the data is stored also on power down.

Switch to Network Layer Group

```
PC:   A9 00 12 84 56 78 9A                                    PEI_SWITCH_req
```

With this setting the module sends all group telegrams to UART. Management telegrams will be routed internally. Therefore the module is still programmable by ETS. This setting is lost after a power down.

Receive Group Telegrams

```
BAOS: 3A BC 11 01 00 01 E1 00 80                                N_GROUP_DATA_ind
BAOS: 3A BC 11 01 00 01 E1 00 81                                N_GROUP_DATA_ind
```

Send Group Telegrams

```
PC:   22 0C 00 00 00 01 E1 00 80                                N_GROUP_DATA_req
BAOS: 3E BC FF FF 00 01 E1 00 80                                N_GROUP_DATA_con

PC:   22 0C 00 00 00 01 E1 00 81                                N_GROUP_DATA_req
BAOS: 3E BC FF FF 00 01 E1 00 81                                N_GROUP_DATA_con
```

After sending a **N_GROUP_DATA_req** the application must wait for a **N_GROUP_DATA_con** before it sends the next **N_GROUP_DATA_req**.

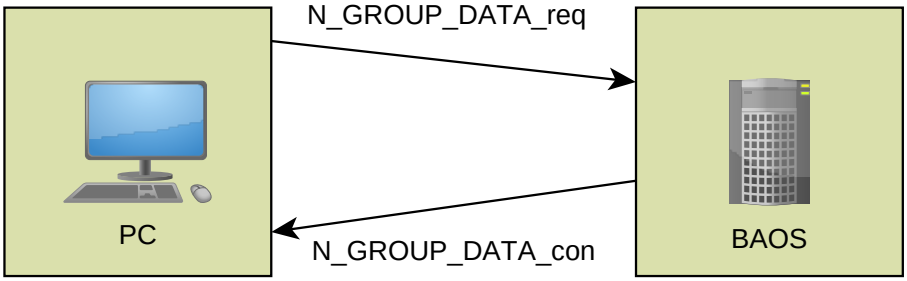


Figure 10.4. EMI2 telegram flow control

The services have to be packed in FT1.2 frames. Here are some examples:

```
                A6 01 40 00 00                          PC_SET_VAL_req
```

As FT1.2:

```
68 06 06 68 73 | A6 01 40 00 00 | 5A 16
```

or

```
68 06 06 68 53 | A6 01 40 00 00 | 3A 16
```

Example:

```
                A9 00 12 84 56 78 9A                    PEI_SWITCH_req
```

As FT1.2:

```
68 08 08 68 73 | A9 00 12 84 56 78 9A | 1A 16
```

or

```
68 08 08 68 53 | A9 00 12 84 56 78 9A | FA 16
```

Some more example for EMI2 uses:

Reset from Bus:

```
C0                                                  FT12_RESET_IND
```

Send Reset to the interface:

```
A9 C0 12 34 56 78 9A                                PEI_SWITCH_REQ
```

Redirect NL group telegrams and user date to PEI

```
A9 00 12 84 56 78 8A                                PEI_SWITCH_REQ
```

Receiving of group telegrams (example)

```
3A BC 11 01 00 01 E1 00 81                          EMI2_N_GROUP_DATA_IND
```

Sending of group telegrams (example)

```
22 0C 00 00 00 01 E1 00 81                          EMI2_N_GROUP_DATA_REQ
```

Confirm frame

```
3E BC FF FF 00 01 E1 00 81                          EMI2_N_GROUP_DATA_CON
```

10.5. Group Telegram Communication On Link Layer

It is also possible to go deeper into the communication stack and use telegrams on the Link Layer. The same steps are necessary as for using the Network Layer.

First we have to switch on the correct internal message routing via PEI message (PC_SET_VAL_req). Then we can send/receive some telegrams (L_DATA_req/L_DATA_con/L_DATA_ind). At last we disable the PEI routing (PC_SET_VAL_req).

To receive any messages send the following EMI2 messages:

Disable group address filter of module

```
PC:   A6 01 40 00 00                                    PC_SET_VAL_req
```

This means to write 0x00 at location 0x4000 which holds the address table length. This is a non volatile value, so the data is stored also on power down.

Switch to Link Layer Group

```
PC:   A9 00 18 34 56 78 9A                                  PEI_SWITCH_req
```

With this setting the module sends all telegrams to the own application. This setting is lost after a power down.

Receive Telegrams

```
BAOS: 29 BC 11 01 00 01 E1 00 80                            L_DATA_ind
BAOS: 29 BC 11 01 00 01 E1 00 81                            L_DATA_ind
```

Send Telegrams

```
PC:   11 0C 00 00 00 01 E1 00 80                            L_DATA_req
BAOS: 4E BC FF FF 00 01 E1 00 80                            L_DATA_con

PC:   11 0C 00 00 00 01 E1 00 81                            L_DATA_req
BAOS: 4E BC FF FF 00 01 E1 00 81                            L_DATA_con
```

After sending a **L_DATA_req** the application must wait for a **L_DATA_con** before it sends the next **L_DATA_req**. See *Figure 10.4, "EMI2 telegram flow control"*.

The services have to be packed in FT1.2 frames. Here is an example:

```
              A9 00 18 34 56 78 9A                  PEI_SWITCH_req
```

As FT1.2:

```
68 08 08 68 73 | A9 00 18 34 56 78 9A | 1A 16
```

or

```
68 08 08 68 53 | A9 00 18 34 56 78 9A | FA 16
```

Some more example for EMI2 uses:

Redirect LL group telegrams and user date to PEI

```
A9 00 18 34 56 78 8A                                  PEI_SWITCH_REQ
```

Receiving of group telegrams (example)

```
29 BC 11 01 00 01 E1 00 81                            EMI2_L_DATA_IND
```

Sending of group telegrams (example)

```
11 0C 00 00 00 01 E1 00 81                            EMI2_L_DATA_REQ
```

Confirm frame

```
2E BC FF FF 00 01 E1 00 81                              EMI2_L_DATA_CON
```

Programming The Base Board

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

The source is written in C suitable for the free available *WINAVR (GNU) compiler*. It will support you in creating own applications based on the ATmega microcontroller. For information about this controller, see

Atmel 8-bit Microcontroller with 128KBytes In-System Programmable Flash ATmega128A [1]

or

Atmel 8-bit Microcontroller with 4/8/16/32KBytes In-System Programmable Flash ATmega328 [2].

11.1. Additional Hardware

To program the Base 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 6.1* and ETS 4

11.2. Installation Of IDE And Compiler

To work with the ATmega you need an interface between PC and the Base Board. We recommend the *AVR JTAGICE3*, which is available at the *Atmel web site* [3]. Download the following free programs, which are not included here due to license restrictions:

Atmel Studio is the IDE (integrated development environment) based on *Microsoft Visual Studio*. It is available for free at the *Atmel web site*[4]. It uses the compiler set WINAVR (GNU). If you already installed Atmel Studio or AVRStudio make sure you have at least version 6.1.7601 - Service Pack 1.

11.3. First Debugging Steps

Connect the JTAG interface to the Base Board and the hardware debugger (e. g. JTAGICE3). The debugger is connected to the PC. Furthermore connect the BAOS Module to the KNX bus. The Development Board needs its power at the USB interface. Connect it to any PC.

The following figure shows the hardware build-up.

[1] http://www.atmel.com/Images/doc8151.pdf
[2] http://www.atmel.com/Images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet.pdf
[3] http://www.atmel.com/tools/jtagice3.aspx
[4] http://www.atmel.com

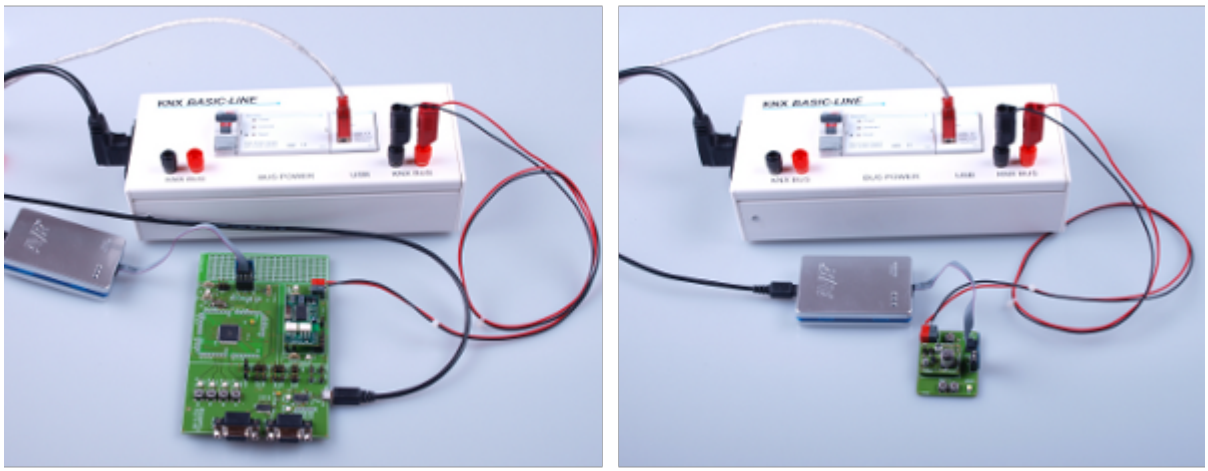


Figure 11.1. Hardware build-up

In this figure, the KNX bus is provided by the KNX BASIC-LINE. It consists of a power supply and a *KNX USB Interface*.

Now start Atmel Studio and open the project **`DevBoardSoftware.atsln`**. Select menu **Build** # **Configuration Manager** and check the correct project for build:

Project	Configuration	Platform	Build
DemoBoardSoftware	Debug	AVR	check this in case of the DemoBoard
DevBoardSoftware	Debug	AVR	check this in case of the DevelopmentBoard

Important

Check only **one** Project for build.

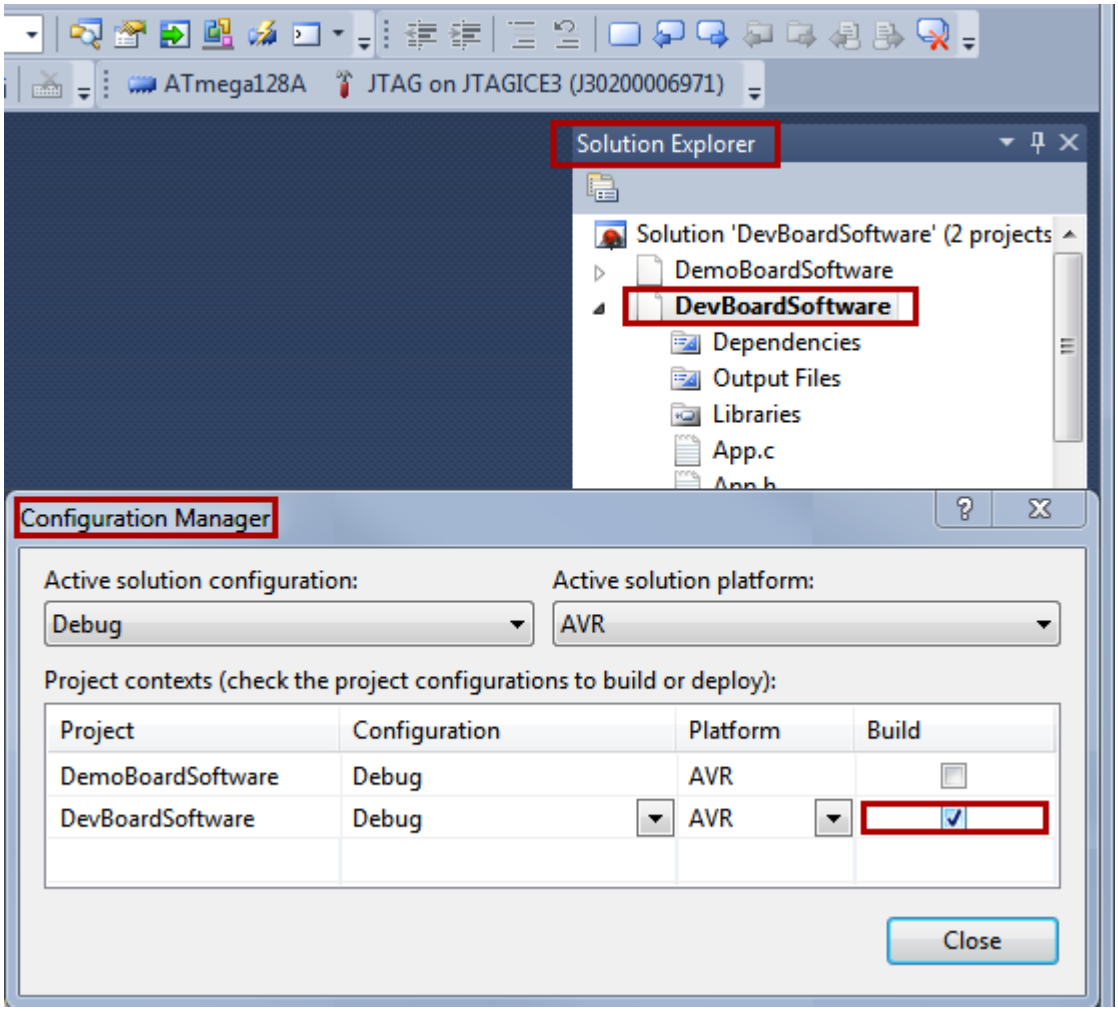


Figure 11.2. Atmel Studio configuration

Exit dialog by clicking **Close** and select the correct project in the **Solution Explorer** (located at the right position). Select **Project** # **Set as StartUp Project** and **Project** # **Properties**. Check whether everything is configured correctly:

Tab	Parameter	Development Board	Demo Board
Device	Device Name	ATmega128A	ATmega328
Tool	Selected debugger/ programmer	JTAGICE3 (or similar)	JTAGICE3 (or similar)
	Interface	JTAG	debugWIRE
	Programming settings	Erase only program area or Erase entire chip	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**.

Important

At this point it might be possible a dialog window pops up telling **Failed to launch debug session with debugWIRE.** [...] **Do you want to use SPI to enable the DWEN fuse?** In this case press the **Yes** button and follow the instructions.

A few seconds later the application starts running at the Base Board 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** again.

Note

If you have problems programming the device via ISP, it's likely the FUSE bits are incorrect. Resetting them might also fail. In this case it is necessary to start a debug session with DebugWire. Enter an AtmelStudio project and select menu **Project** # **Properties...**. There select **Tool**, **JTAGICE3**, Interface **debugWire** and start debugging (**F5**). While running in a debug session select menu **Debug** # **Disable debugWire and Close**. This closes the current debug session and sets the FUSE bit for enabling the ISP interface. Now enter the **Tools** # **Device Programming** and use ISP interface to manipulate the FUSE bits, program the device, etc.

11.4. The Application Framework

To see the anatomy of the demonstration application see *Chapter 6, The Application Framework*.

11.5. Creating Own Applications

To make your own application see *Section 6.1, “Creating Own Applications”*.

Appendix A. About KNX

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 which conforms to the *OSI model*[1]. Every single device connected to this bus has its own microcontroller 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 no further effects 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 to use a sensor which is also an actor. So it's possible to send and receive data.

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 an area line.
- **`Backbone coupler (BC)`** to connect more area lines.

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

[1] http://en.wikipedia.org/wiki/Iso_osi

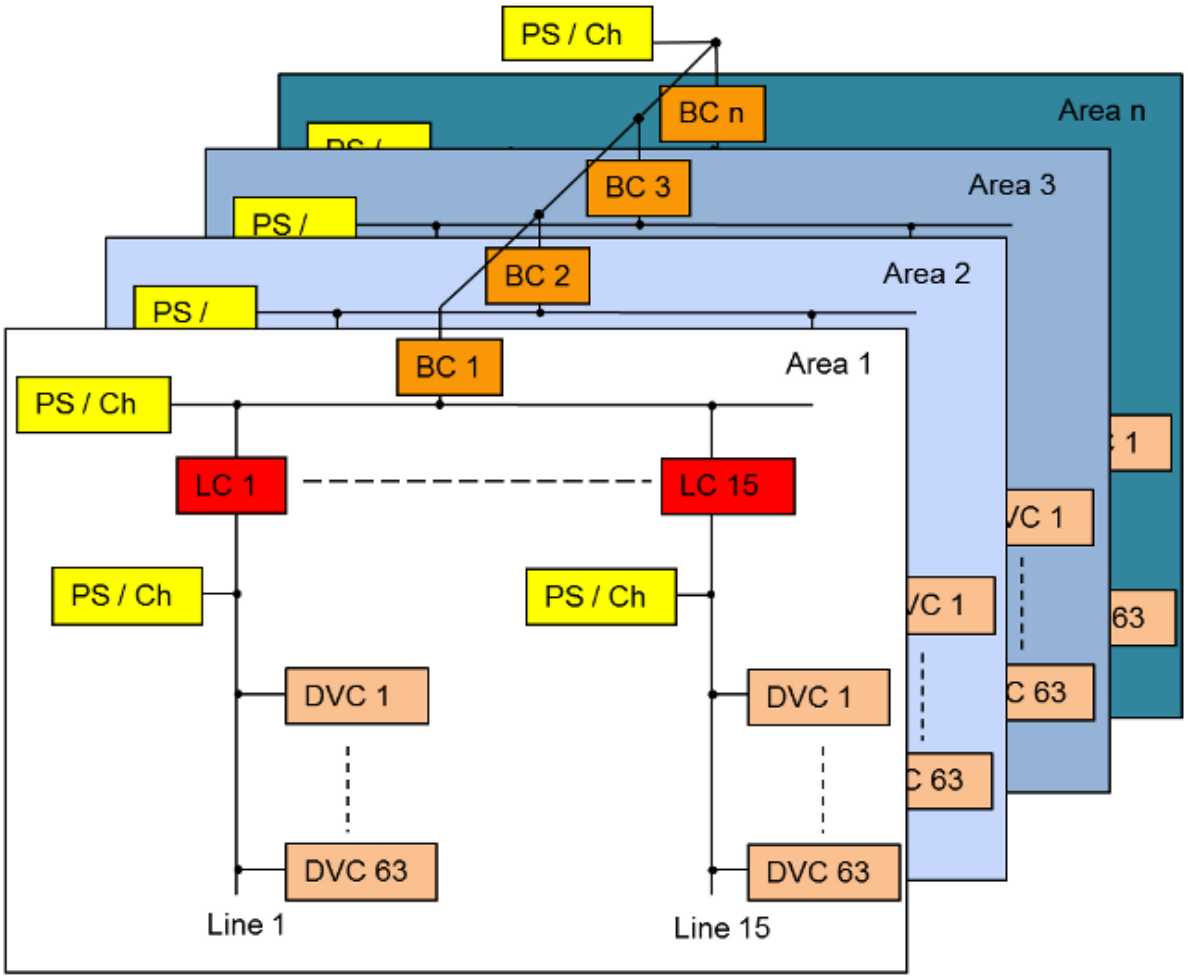


Figure A.1. KNX basic topology

A.1. KNX Twisted Pair Bus System

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 volts. The operating range of the bus devices are located between 21 and 29 volts.

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

A.2. 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 actors). Connection oriented telegram are generally used for downloads by the ETS. Standard telegram have a data length up to 15 bytes. For more data an extended frame is used.

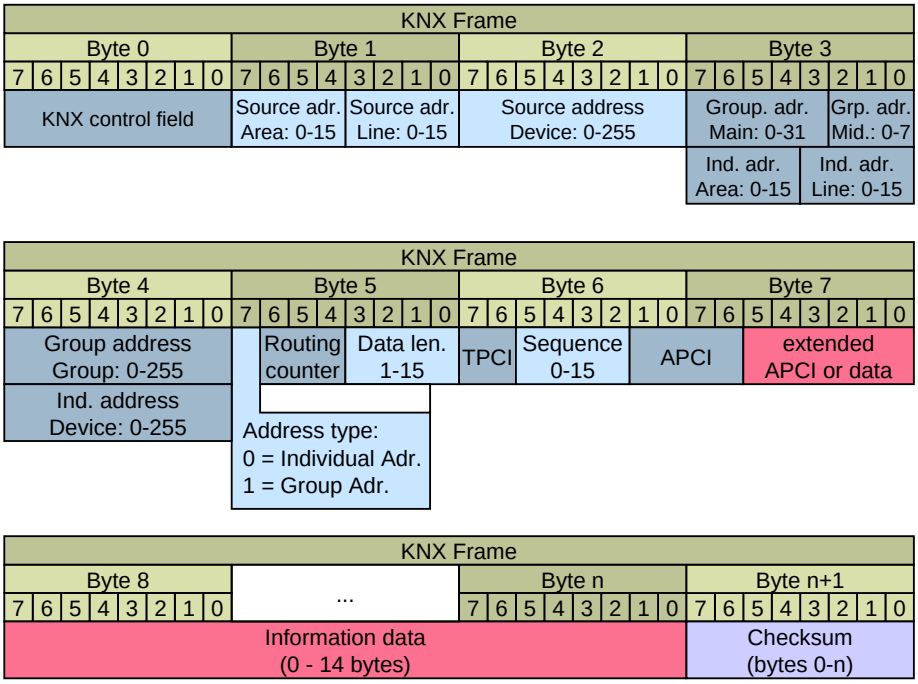


Figure A.2. KNX standard connection less telegram frame

The fields of this telegram are:

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

- **`Bit #0:`** EMI2 network layer access:

  0 = Positive confirmation

  1 = Negative confirmation

- **`Bit #1:`** EMI2 link layer access:

  0 = Don't care

  1 = no L2 acknowledge requested

- **`Bit #2-3:`** Priority:

  00 = System priority (System functions)

  01 = Urgent priority (Alarm functions)

  10 = Normal priority (High prioritized functions)

  11 = Low priority (Low prioritized functions)

- **`Bit #4:`** unused, must be 1

- **`Bit #5:`** Repeat flag:

  0 = repeated L_Data_Standard frame

  1 = not repeated L_Data_Standard frame

  0 = in case of an Acknowledge frame

1 = in case of L_Poll_Data frame

- **Bit #6:** Poll data/data:

  0 = L_Data frame

  1 = L_Poll_Data frame

- **Bit #7:** Frame type:

  1 = L_Data_Standard frame

  0 = L_Data_Extended frame

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 A.4, "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 **transport protocol control information (TPCI)** is a code from the transport layer.

The **sequence** number is part of the TPCI. It is used for connection oriented frames (e. g. ETS download).

The **application protocol control information (APCI)** is a code from the application layer.

The **extended application protocol control information (APCI)** is a code from the 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. In case of zero bytes the extended APCI can hold the data.

The **checksum** validates the frame. I 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*[2].

A.3. 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

[2] http://www.knx.org/knx-en/knx/technology/specifications/index.php

transmission is complete, the telegram must be acknowledged (ACK) after time t2. All addressed bus devices acknowledge the receipt of the telegram simultaneously.

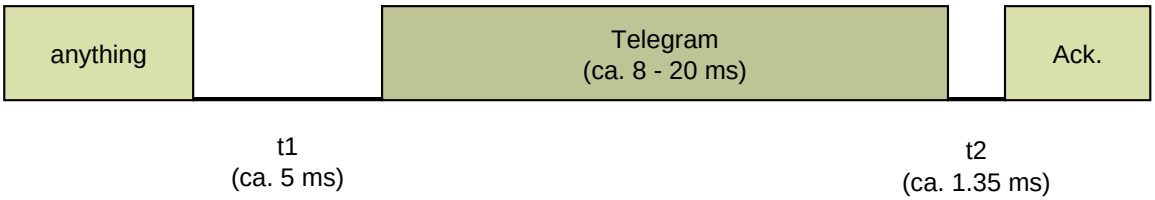


Figure A.3. KNX Telegram Timing

A.4. 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.

Byte 0								Byte 1							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Area: 0-15				Line: 0-15				Device: 0-255							

Figure A.4. Individual Address Frame

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

  The default in ETS4 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							

Figure A.5. 3-Level Group Address

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

A.5. Data Point Types

The data point types describe the size, the value range and its representation of information data. The most common types are:

Type	ID	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

Type	ID	Name	Encoding (Representation)	Value range	Size (bits)
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
uint32	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*[3].

[3] http://www.knx.org/knx-en/knx/technology/specifications/index.php

Appendix B. Commissioning With ETS

ETS (Engineering Tool Software) is a manufacturer independent configuration tool software to design and configure intelligent home and building control installations with the KNX system. It is also necessary for configuring the BAOS Module. ETS runs on computers using the *Microsoft Windows* operating system.

B.1. Install ETS

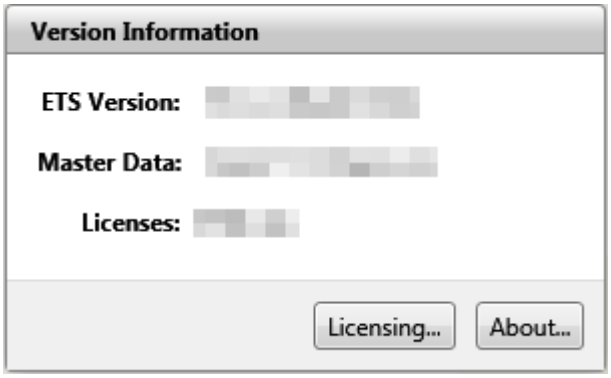
It can be downloaded from the *KNX download page*[1]. Version 4 suits fine for working which the BAOS Module.

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

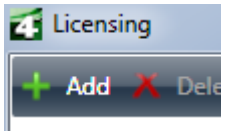
B.2. Install ETS License

A license is also necessary to run ETS. The free demo license allows 3 devices per project. If more are needed, we recommend to do the *eCampus*[2] training to get a *Lite* license which allows up to 20 devices. The full license is available at the KNX Online Shop.

The demo license is installed automatically. If you have another license, like the Lite one, install it as follows:

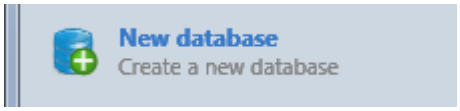
Copy the license file (.lic) to any location on your local disk.	
Start ETS.	
The first screen shows an overview with **Version Information**. There is a button named **Licensing...**. Push this button and a dialog window opens showing the installed licenses.	

[1] http://www.knx.org/knx-tools/ets4/download
[2] http://wbt4.knx.org

Use the **Add** menu to load more licenses. A file browser will appear. Use it to choose your new license.	

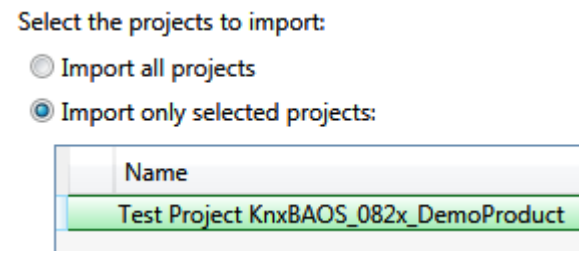
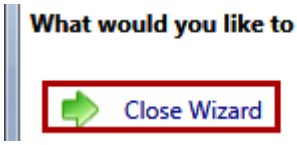
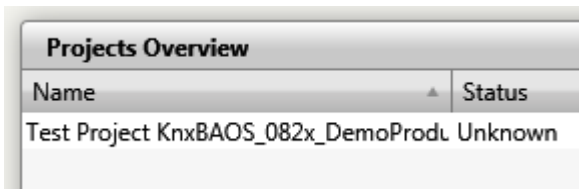
B.3. Create A Database

ETS needs a database to store all data like projects and products. So it is necessary to create a database. This must be done only once.

Click on the button **New database** at the left side to create a new database on your local disk.	

B.4. Import A Project

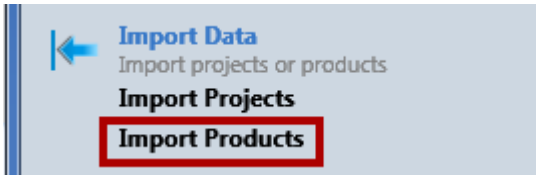
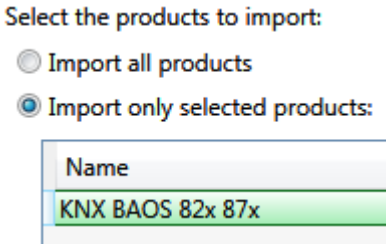
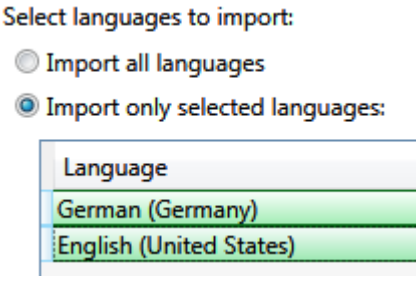
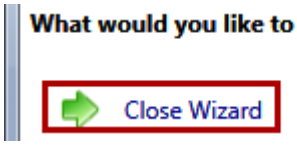
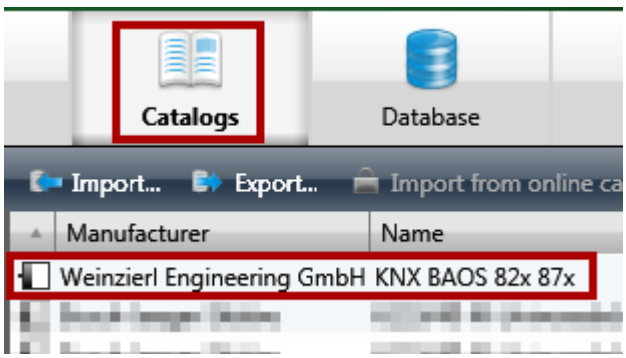
To do the first steps in ETS using the Development/Demo Board connect it as described above to the KNX bus. Don't forget to connect the PC to a KNX Interface, too. After powering up the board, start ETS and import the test project included in this package:

Import Projects	
Next (to skip the information page). Select the file to import: *KnxBAOS_Demo/ KnxBAOS_Demo_ETS_MT/ Test Project KnxBAOS_082x_Demo Product.pr5*. **Next**; **Next** (skip conversion); **Next** (import Test Project KnxBAOS_082x_DemoProduct).	
Import; After a while **Close Wizard**.	
Now open the new Project by double-clicking it.	

B.5. Import A Product

Importing an already prepared project usually contains some products, i. e. KNX devices. Nevertheless it is necessary to import some more products into the catalog. If there is a product in in the catalog missing, contact the manufacturer for the data to import into the catalog. Sometimes the products can be downloaded from their sites.

This package also contains the product file for the 0820/0822 BAOS Modules. To import it, do the following:

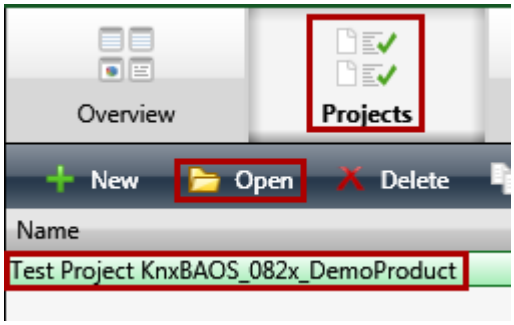
Import Products	
Next (to skip the information page). Select the file to import: *KnxBAOS_Demo/KnxBAOS_Demo_ETS_Projects/KNX_BAOS_82x_87x.knxprod*. **Next**; **Next** (skip conversion); **Next** (import KNX BAOS 82x 87x).	
Next (to skip the information page). Select the languages to import. **Next**.	
Import; After a while **Close Wizard**.	
Now open the **Catalogs** by clicking it and verify if the product has been inserted.	

Note

A product in ETS is a certain device from a certain manufacturer. It is digitally signed, so it cannot be manipulated. To test an own product which is not certified by the ETS and thus the KNX organization, the following must be done. Create a product with the *Manufacturing Tool* and save it in a test project. A project normally contains the complete information about a fully commissioned building with all its KNX devices and their connections. The test project will contain only one uncertified device which can be used for testing purposes. This device can be copied from this test project to other projects but it cannot be installed into the product catalog.

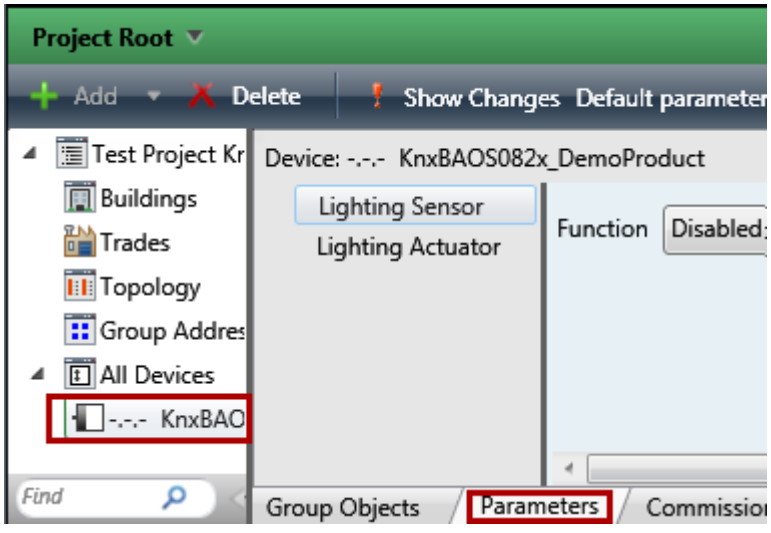
B.6. Open A Project

To open an existing project, do the following:

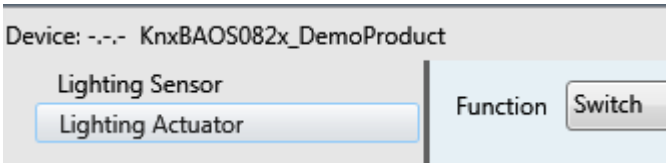
Click **Projects** and select the project you want to open. Click **Open** to open it.	

B.7. Commissioning A Project

After importing the *`Test Project KnxBAOS_082x_DemoProduct.pr5`* and opening it (or creating a new project) we can commission the project and configure the KNX devices.

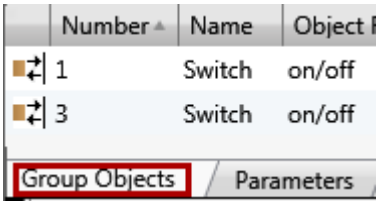
We have an empty project now which only contains the *KnxBAOS_082x_DemoProduct*. It is located in the tree at **`All Devices`**. Go there, click this device and select **`Parameters`** in the neighboring window:	

Select **Lighting Sensor** and change its function to **Switch**. Do the same with **Lighting Actuator**.



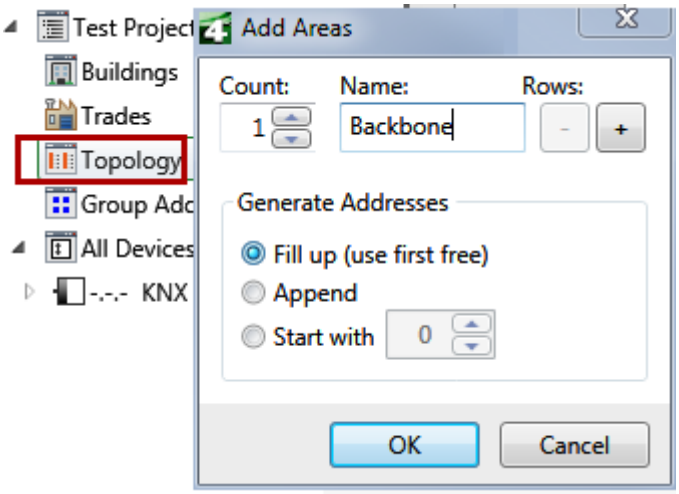
Select **Group Objects**. We see two **`Group Objects`**:

```
1   Switch   on/off
3   Switch   on/off
```

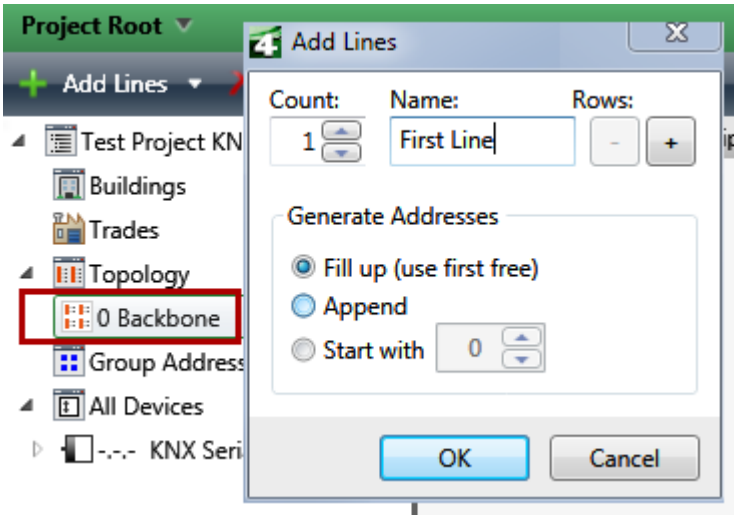


To actually use the device we must create a valid **`Topology`**.

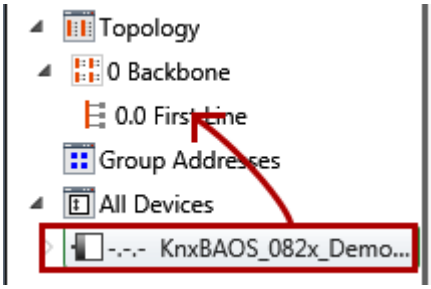
Select **`Topology`** in the tree and **`click right mouse button`**. Choose **Add: Areas** and enter a name in the dialog (e. g. **`Backbone`**).

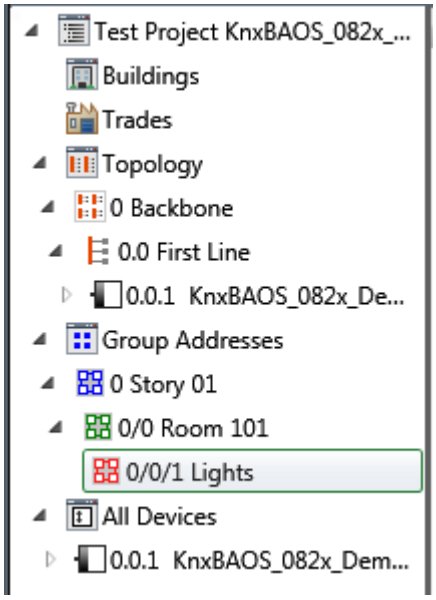
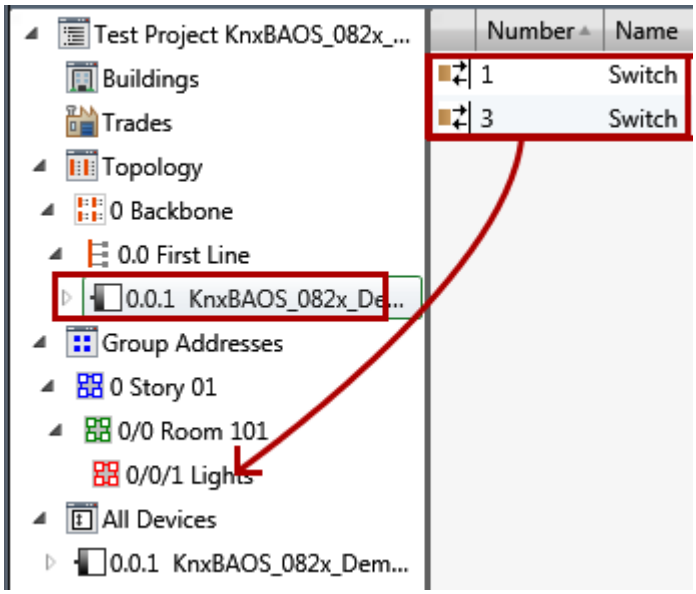


Choose the newly created area with **`right mouse button`** and **Add: Lines**. Enter a name (e. g. **`First Line`**).



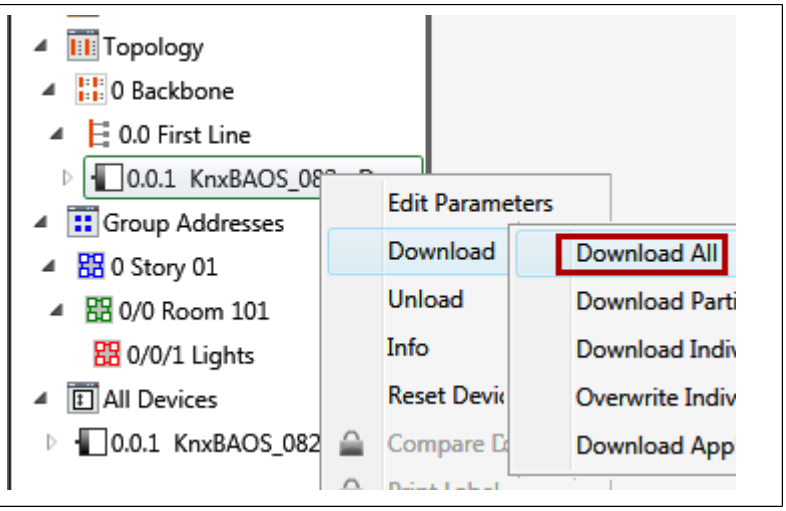
Drag and drop the **`KnxBAOS_082x_DemoProduct`** to the newly created line.



To configure the device select **`Group Addresses`** in the tree and **`click right mouse button`**. Choose **Add: Main Groups** and enter a name in the dialog (e. g. **`Story 01`**). Choose the newly created group with **`right mouse button`** and **Add: Middle Groups**. Enter a name (e. g. **`Room 101`**). Again choose the newly created Middle Group with **`right mouse button`** and **Add: Group Addresses**. Enter a name (e. g. **`Lights`**).	
To connect the *Group Objects* click **KnxBAOS_082x_DemoProduct** and drag and drop its *Group Objects* into *Group Address* **`Lights`**.	

B.8. Download A Configuration

To finalize the configuration of some KNX devices these devices must be selected and the configuration downloaded:

Select the device **`KnxBAOS_082x_DemoProduct`** with **`right mouse button`** and **Download** # **Download All**. Press the *Learn button* on the Board (red LED must light up briefly).	

Appendix C. Using Net'n Node

The package includes the program *Net'n Node*. With this program you can communicate with the BAOS Module without ETS. Install it and connect the PC to the KNX bus via bus interface, e. g. KNX USB Interface 0311. (Available at *Weinzierl web site*[1])

Net'n Node can be found in the directory **`NetnNode`**. It contains an MSI installation archive for *MS Windows*. Simply double click it and follow the installation instructions.

Important

To run Net'n Node a license file is required. It can be obtained from `support@weinzierl.de` .

Note

Net'n Node uses a local network connection. If you get a *Windows Security Warning* (firewall), click on **Allow access**. No data will be sent over the Internet.

Now choose your bus interface by hitting the button **Add Port**. A pop up menu appears with several options. Depending on your connection to the KNX bus choose the applicable one. Commonly it is **KNX via USB**.

Activate the bus interface by clicking the button **ON** and configure the interface by clicking the **Setup** button in the tool bar. This opens a dialog window containing some parameters. Check and alter **Address Table Length**. It must contain **00 hex** and click **Close**. The Address Table acts as filter for all telegrams. Only telegrams matching the Address Table are displayed. Setting the length to 0 disables the filtering.
Make sure to select **Link Layer** to see messages in the monitor.

Now to display the telegrams in the monitor, which are sent by the BAOS Module, choose the menu item **File** # **New Telegram List** # **Standard**. A new telegram window appears containing the current telegrams which are sent via *KNX* bus.

If the buttons are pressed on the board, Net'n Node shows the telegrams sent by the device. At the right hand side of the table the values can be read. The current address of our BAOS Module is shown in the source address (**Src-Addr**) column of the table.

Net'n Node is also capable of creating and sending own telegrams. To send one to the device proceed as follows. Select the menu **Send KNX** # **KNX Interworking Datapoint Types** # **DPT 01 - Binary - 1**

[1] http://www.weinzierl.de

bit. This opens a dialog window where we can select the content of our telegram. Set the address type to **Dec 3L** and enter the correct **Destination Group Address** which is per default **`0/0/1`**. Select **True/On** in the **Data** area and click the button **Send**. The LED should light up. Simultaneously the telegrams of the KNX bus are shown in the telegram view.

Now close the dialog and exit Net'n Node to free the resources for other tools like ETS.

Note

It is also possible to use the **Busmon** as Layer. In this case all telegrams on the KNX bus are visible in the Telegram List as they are on the bus. In this mode no routing and filtering is done since these things are done in the upper layers.

Selecting **Link Layer** shows all telegrams from the link layer and enables to edit and send a self made telegram.

Appendix D. Individual ETS Entries

To create own individual ETS entries the following is required:

1. The *KNX Manufacturer Tool (MT4)*[1] and a valid license.
2. The *KNX Engineering Tool Software (ETS4)*[2] and a valid license.
3. This Weinzierl KNX BAOS Development Kit.
4. KNX BAOS client software (included in this development kit).

The MT4 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:

1. KNX membership
2. ISO 9001 certification
3. Certification of the product

`Facts good to know about the BAOS Module for working with MT4:`

Profile Class	System 7
Interface	Serial asynchronous
Protocol	FT 1.2 based
Address Table	starting at address 0x4000
Association Table	starting at address 0x4200
ComObjects Table	starting at address 0x4400, 250 pre-defined
Parameters	starting at address 0x4900, 250 parameters each 1 byte

GO number	DPT	Length
0	not available	not available
1 ... 32	01, 02, ... 18	1 bit ... 14 bytes
33 ... 250	01, 02, ... 15, 17, 18	1 bit ... 4 bytes

[1] http://www.knx.org/knx-tools/manufacturer-tool/description
[2] http://www.knx.org/knx-tools/ets4/description

Note: Group Object vs. Communication Object

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.

D.1. Example For Creating An Individual ETS Database

To create a database, start MT4 and do the following:

D.1.1. Project

- Create a new project by selecting menu **File**+**New** # **Project...**, select **KNX MT Project**, browse for a location to store the project, enter a name (e. g. **`KnxBAOS_082x_DemoProduct`**) and hit **OK**.
- In the dialog window it is recommended to select **ETS 3.0f** since a lot of users still use ETS3. For manufacturer select either your own name or **M-00FA KNX Association** and hit **OK**.

D.1.2. Create New Application

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

  Click the project name in the Solution Explorer (**`KnxBAOS_082x_DemoProduct`**) and use the menu **Project** # **Add New Item...**, select **Application Program**, edit its name (e. g. **`KnxBAOS_082x_DemoApp.mtxml`**) and hit **Add**. Enter values in the dialog window:

```
Application Number:  0001h
Application Version: 10h
Name:                KnxBAOS_082x_DemoApp
Mask Version:        [0701h] 7.1
```

  and hit **OK**.

D.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 (**`KnxBAOS_082x_DemoProduct`**) and use the menu **Project** # **Add New Item...**, select **Hardware**, edit its name (e. g. **`KnxBAOS_082x_DemoHw.mtxml`**) and hit **Add**. Enter values in the dialog window:

```
Serial Number:  SN2014.01.07.001
Version Number: 0100h
Hardware Name:  KnxBAOS_082x_DemoHw
```

  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] KnxBAOS_082x_DemoApp`**. Choose the correct medium type (e. g. **`TP`**) and hit **OK**.

D.1.4. Binary Import

- Edit the application by selecting the application tab and then its name (e. g. **`KnxBAOS_082x_DemoApp`**). Use right mouse button menu **Import binary data**, browse to the s19-File which is also provided in this package (e. g. **`KnxBAOS_Module_2010_09_21.s19`**) and open it. Select **Add ComObjects** and hit **OK**.

  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 **Code Segments** entry. Click it and see the table. It contains memory regions for the following purposes (see other entries at same tree level):

```
Address Table:     4000h, size 01FFh
Association Table: 4200h, size 01FFh
ComObjects:        4400h, size 0408h
    Object   0... 32: 14 bytes size
    Object  33...250:  4 bytes size
    Object 251...254:  1 bit size
ComObjectsRefs:
    Object   0... 32: 14 bytes size
    Object  33...250:  4 bytes size
    Object 251...254:  1 bit size
```

  Load Procedures/Complete:

  1. Connect
  2. Unload lsm=#01h (Address Table)
  3. Unload lsm=#02h (Association Table)
  4. Unload lsm=#03h (Application Program)
  5. Load lsm=#01h (Address Table)
  6. AbsSegment lsm=#01h, type=00h, addr=4000h, ... (Address Table)
  7. TaskSegment lsm=#01h, addr=4000h (Address Table)
  8. LoadCompeted lsm=#01h (Address Table)
  9. Load lsm=#02h (Association Table)
  10. AbsSegment lsm=#02h, type=00h, addr=4200h, ... (Association Table)
  11. TaskSegment lsm=#02h, addr=4200h (Association Table)
  12. LoadCompeted lsm=#02h (Association Table)
  13. Load lsm=#03h (Application Program)
  14. AbsSegment lsm=#03h, type=00h, addr=0700h, ... (Application Program)

15. AbsSegment lsm=#03h, type=00h, addr=4400h, ... (Application Program)
16. AbsSegment lsm=#03h, type=00h, addr=4810h, ... (Application Program)
17. AbsSegment lsm=#03h, type=00h, addr=4900h, ... (Application Program)
18. AbsSegment lsm=#03h, type=00h, addr=4E00h, ... (Application Program)
19. TaskSegment lsm=#03h, addr=4800h (Application Program)
20. TaskCtrl1 lsm=#03h, addr=0000h, count=00h (Application Program)
21. LoadCompeted lsm=#03h (Application Program)
22. Restart
23. Disconnect

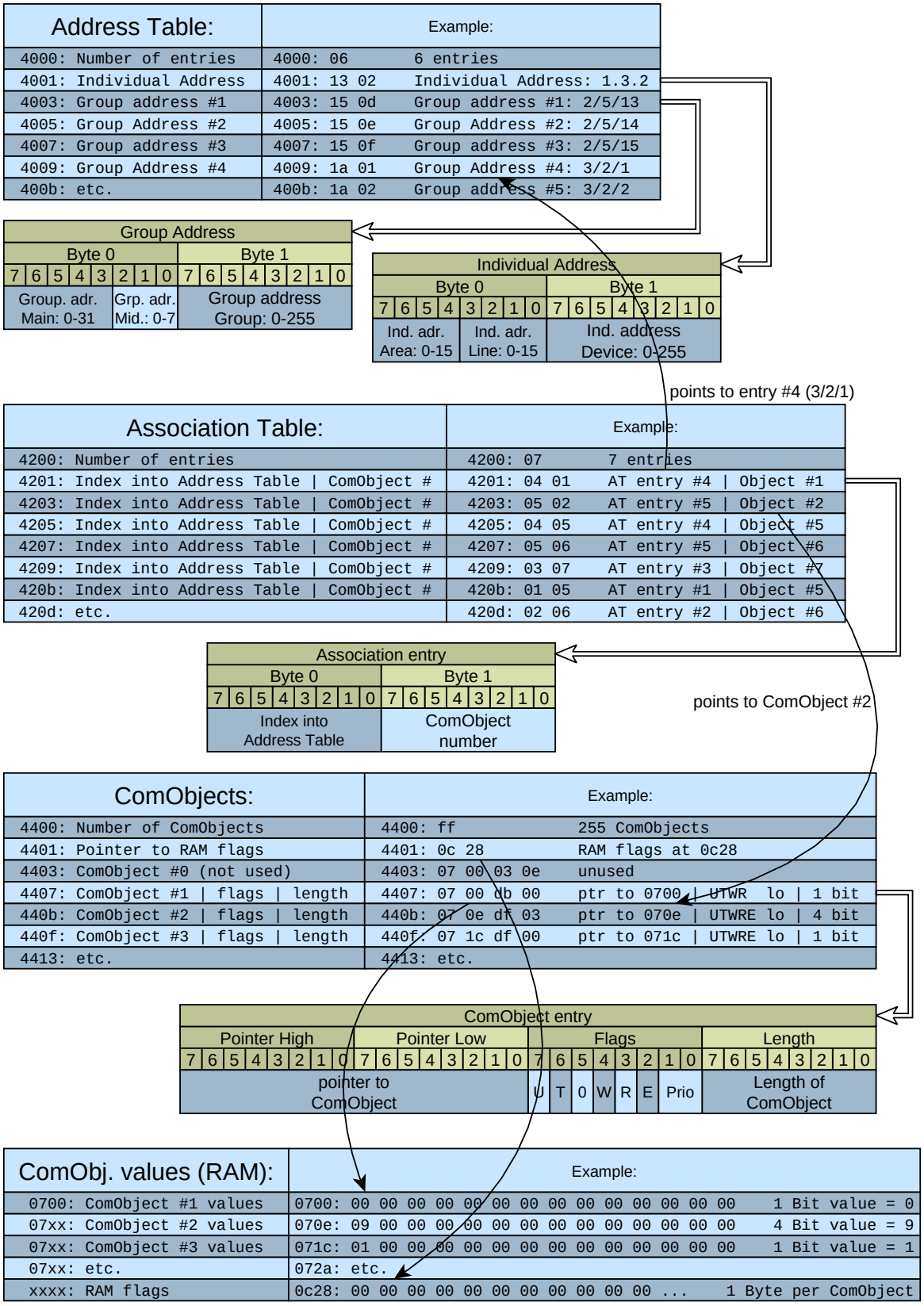


Figure D.1. System 7 Virtual Memory

The **`ComObject flags`** have the following information:

- **`Bit #0-1:`** Priority:

  00 = System priority (System values)

  01 = Urgent priority (Alarm values)

  10 = Normal priority (High prioritized values)

  11 = Low priority (Low prioritized values)

- **`Bit #2:`** ComObject (Data Point) enabled:

  0 = disabled

  1 = enabled (i.e. linked by ETS to group address)

- **`Bit #3:`** Read 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)

- **`Bit #4:`** Write via bus:

  0 = Value is not writable

  1 = Value is writable from bus

- **`Bit #5:`** unused, must be 0

- **`Bit #6:`** Transmit (send):

  0 = Changed value will not be sent to the bus

  1 = Changed value will be sent to the bus

- **`Bit #7:`** Update (from response frame, not used in current devices):

  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

The **`ComObject length`** has the following information:

Value	Length	Value	Length
0	1 bit	7	1 byte
1	2 bits	8	2 bytes
2	3 bits	9	3 bytes
3	4 bits	10	4 bytes
4	5 bits	11	6 bytes
5	6 bits	12	8 bytes
6	7 bits	13	10 bytes
		14	14 bytes
		15	variable length

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

Parameters:	Example:
4900: Parameter #1	4900: 00
4901: Parameter #2	4901: 00
4902: Parameter #3	4902: 00
4903: etc.	4903: etc.

ComObject Data Types:	Example:
4e00: DataPointType #0 (not used)	4e00: 00
4e01: DataPointType #1	4e01: 01
4e02: DataPointType #2	4e02: 03
4e03: etc.	4e03: etc.

Figure D.2. Virtual Memory: Parameters and ComObject Data Types

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 BAOS Module the following parameters are used:

Address	Usage
4900	Not used
4901	Switch: 0 = disabled, 1 = switch, 2 = dimmer, 3 = shutter
4902	Light: 0 = disabled, 1 = switch, 2 = dimmer
4903 - 49ff	Not used

The **`ComObject Data Types`** determine the data point types as it is done by the ETS. These values are used by the BAOS ObjectServer. For each ComObject the data point type is determined by one byte. The values are as follows:

Value	Data Type	Value	Data Type
0	Disabled	10	Time - 3 bytes
1	Binary - 1 bit	11	Date - 3 bytes
2	Binary controlled - 2 bits	12	Unsigned value - 4 bytes
3	Dim up/down - 4 bits	13	Signed value - 4 bytes
4	Character - 1 byte	14	Float value - 4 bytes
5	Scaling - 1 byte	15	Access data - 4 bytes
6	Signed value - 1 byte	16	Character string - 14 bytes
7	Unsigned value - 2 bytes	17	Scene - 1 byte
8	Signed value - 2 bytes	18	Scene controlled - 1 byte
9	Float value - 2 bytes	255	Unknown

D.1.5. Create Visible Data Points

The BAOS Module has 255 ComObjects. Due to the available memory, the first 33 Objects can use up to 14 bytes each. The next 218 can use up to 4 bytes and the remaining 5 Objects only 1 bit.

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.

  Select the following ComObjects in **ComObjectRefs** table and edit them (The flags can only be edited in the **ComObjects** table):

ComObject	Text	Function Text	Object Size	Data Point Type	Flags
[0001h] Obj1-	Switch	on/off	1 Bit	[1.1] DPT_Switch	--CT--
[0002h] Obj2-	Dim	up/down	4 Bit	[3.7] DPT_Control_Dimming	--CT--
[0003h] Obj3-	Switch	on/off	1 Bit	[1.1] DPT_Switch	RWC---
[0004h] Obj4-	Dim	up/down	4 Bit	[3.7] DPT_Control_Dimming	RWC---
[0005h] Obj5-	Dim	value	1 Byte	[5.10] DPT_Value_1_Ucount	RWC---

  The meaning of the Flags are described in *Section D.1.4, “Binary Import” [90]*.

D.1.5.1. 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:        [4900]
Internal Name:       Button
Offset:              0000h
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:        [4E00h]
Internal Name:       DPT-Value_GO1
Offset:              0001h
Parameter Type:      ButtonFunction_t
Text:                DPT-Value_GO1
Create ParameterRef: True
```

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

```
Access:              None
Bit Offset:          00h
Code Segment:        [4E00h]
Internal Name:       DPT-Value_GO2
Offset:              0002h
Parameter Type:      ButtonFunction_t
Text:                DPT-Value_GO2
Create ParameterRef: True
```

  and hit **OK**.

- Unfold the dynamic part. **Channel 1** should already be existing. 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: Lighting Sensor
```

and hit **OK**.

- Add a parameter to the ParameterBlock. 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 ParameterBlock. Select **PageButton** and choose right mouse button menu **Add new** # **Choose**. Enter these values in the dialog window

```
Parameter:           [0002h] DPT-Value_GO1
Create Default when: False
```

and hit **OK**.

Select the newly created **DPT-Value_GO1** 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 0002h] Switch`**
Add new # **ParameterRefRef**	**`[0003h] DPT-Value_GO1`**

Go to **Static/ComObjectRefs** and locate **Obj1** (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.

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

Menu	Item to select
Add new # **ComObjectRefRef**	**`[0001h 0100h] Switch-on/off`**
Add new # **ParameterRefRef**	**`[0004h] DPT-Value_GO1`**
Add new # **ComObjectRefRef**	**`[0002h 0003h] Dim-up/down`**
Add new # **ParameterRefRef**	**`[0003h] DPT-Value_GO2`**

D.1.5.2. 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:        [4900]
Internal Name:       Light
Offset:              0001h
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:        [4E00h]
Internal Name:       DPT-Value_GO3
Offset:              0003h
Parameter Type:      LightFunction_t
Text:                DPT-Value_GO3
Create ParameterRef: True
```

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

```
Access:              None
Bit Offset:          00h
Code Segment:        [4E00h]
Internal Name:       DPT-Value_GO4
Offset:              0004h
Parameter Type:      LightFunction_t
Text:                DPT-Value_GO4
Create ParameterRef: True
```

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

```
Access:              None
Bit Offset:          00h
Code Segment:        [4E00h]
Internal Name:       DPT-Value_GO5
Offset:              0005h
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: Lighting Actuator
  ```

  and hit **OK**.

- Add a parameter to the ParameterBlock. 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 ParameterBlock. Select **PageLight** and choose right mouse button menu **Add new** # **Choose**. Enter these values in the dialog window

  ```
  Parameter:           [0007h] Light
  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 0004h] Switch-on/off**
Add new # **ParameterRefRef**	**[0008h] 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 0101h] Switch-on/off**
Add new # **ParameterRefRef**	**[000Ah] DPT-Value_GO3**
Add new # **ComObjectRefRef**	**[0004h 0005h] Dim-up/down**
Add new # **ParameterRefRef**	**[0009h] DPT-Value_GO4**
Add new # **ComObjectRefRef**	**[0005h 0006h] Dim-value**
Add new # **ParameterRefRef**	**[000Dh] DPT-Value_GO5**

D.1.6. Hide Unwanted Data Points

- To hide all unwanted data points 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 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 **`[000Ch] 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 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, #6 to #254). Copy (right mouse button at first (empty) column) and paste them to **(Hidden)**.

D.1.7. 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 MT4.

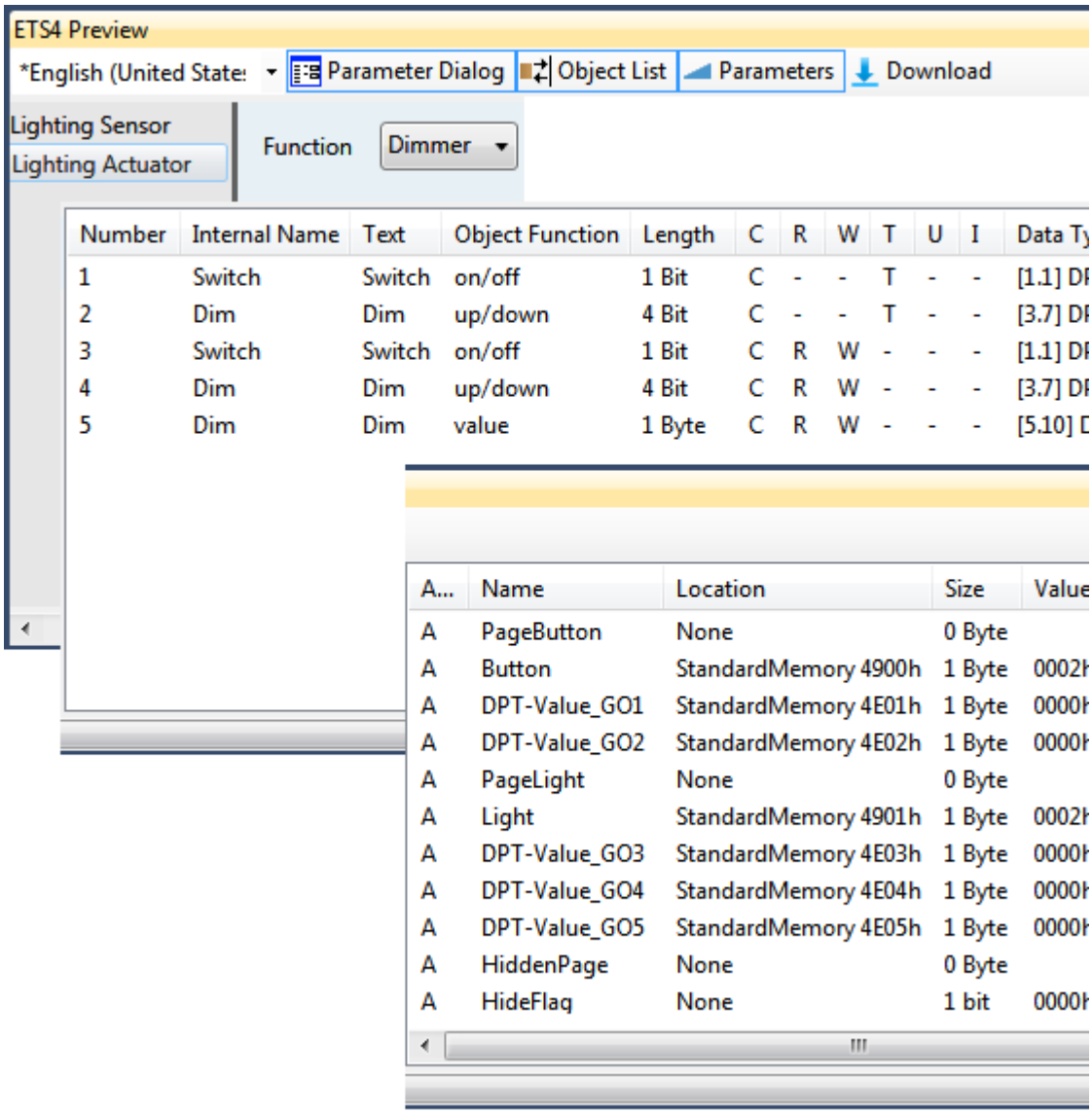


Figure D.3. ETS Preview

Close the preview window and continue.

This example project is also included in this package at **`KnxBAOS_Demo/KnxBAOS_Demo_ETS_MT`**.

D.1.8. Create New Product

- Create a product name by selecting the hardware name again (the line containing the serial number). Use right mouse button menu **Add new Product** and enter an order number, the product's text and hit **OK**.

- Create a catalog by selecting the project name in the Solution Explorer (**`KnxBAOS_082x_DemoProduct`**) and use the menu **Project** # **Add New Item...**, select **Catalog**, edit its name 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 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**.

D.1.9. Export The Project

- Export project for testing

  Select name of the application (e. g. **KnxBaos_82x_Product**) 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 pr5-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 ETS4.

D.2. Test The Individual ETS Database in ETS

To test the database, start ETS4 and do the following:

- Import the test project created by MT4 into ETS4. 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 Man line** containing the device at 1.1.32.
- Create two Group Addresses in the **Group Addresses** view: **New main group/New middle group/ On-Off** and **New main group/New middle group/Dimming**.
- 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: Switch** and **3: Switch** to the group address **On-Off**.

  Drag the data points **2: Dim** and **4: Dim** to the group address **Dimming**.

- Finally press the learning key on the Base Board and select **Download** # **Download All**.

Switching and dimming of the LED is possible now.

Appendix E. 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*[1] -> 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*[2].
- 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*[3].
- 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 0820/0822, the device *inherits* the certified status of it. Therefore only the application specific tests (interworking/functionality) are required.

For further details please contact *info@weinzierl.de* .

[1] http://www.knx.org
[2] http://www.knx.org/knx-en/knx/technology/specifications/index.php
[3] http://www.knx.org/knx-en/knx/technology/specifications/index.php

Appendix F. Revision History

Revision 1.4 **2013-12-03** **Johannes Geiss**

Document written.

Index

R

S

T

U

W