

# Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-  
ICT

## **Masterthesis**

### ***Integrating KNX into the Smart Home Controller***

#### **Siebe Nijis**

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

#### **PROMOTOR :**

dr. Nikolaos TSIOGKAS

#### **PROMOTOR :**

ing. Robin MOONS

Gezamenlijke opleiding UHasselt en KU Leuven



Universiteit Hasselt | Campus Diepenbeek | Faculteit Industriële Ingenieurswetenschappen | Agoralaan Gebouw H - Gebouw B | BE 3590 Diepenbeek

Universiteit Hasselt | Campus Diepenbeek | Agoralaan Gebouw D | BE 3590 Diepenbeek  
Universiteit Hasselt | Campus Hasselt | Martelarenlaan 42 | BE 3500 Hasselt



**2022**  
**2023**

# Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-  
ICT

## ***Masterthesis***

### ***Integrating KNX into the Smart Home Controller***

#### **Siebe Nijs**

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

#### **PROMOTOR :**

dr. Nikolaos TSIOGKAS

#### **PROMOTOR :**

ing. Robin MOONS



**KU LEUVEN**



# Preface

During my master's thesis at Bits & Bytes, I was given the opportunity to not only conduct research but also gain practical experience in various aspects of the industry. This included troubleshooting and diagnosing problems at customer sites, connecting electrical wiring, and engaging with external partners. These experiences have greatly enhanced my understanding of the field and provided me with invaluable skills that I can apply in my future career.

I would like to thank Ing. Robin Moons for providing unwavering support during my internship. His encouragement and guidance has been instrumental in my professional growth. I would also like to extend my heartfelt appreciation to Dr. Nikolaos Tsiogkas for fostering a friendly and approachable communication environment. His valuable guidance and constructive feedback has been invaluable assets in my learning journey.



# Contents

<b>Preface</b>	<b>1</b>
<b>List of Tables</b>	<b>5</b>
<b>List of Figures</b>	<b>7</b>
<b>Glossary</b>	<b>9</b>
<b>Abstract</b>	<b>11</b>
<b>Abstract in English</b>	<b>13</b>
<b>1 Introduction</b>	<b>15</b>
<b>2 KNX and the Smart Home Controller</b>	<b>17</b>
2.1 Introduction to the Smart Home Controller . . . . .	17
2.1.1 SHC features in KNX . . . . .	17
2.1.2 Expandability of the SHC . . . . .	18
2.2 Introduction to KNX . . . . .	18
2.2.1 The KNX bus system . . . . .	18
2.2.2 KNX Twisted Pair Telegrams . . . . .	19
2.2.3 The KNX ETS Software . . . . .	20
2.2.4 Datapoints and parameters . . . . .	22
2.2.5 Example of KNX communication . . . . .	22
2.3 KNX Certification . . . . .	22
2.3.1 Membership . . . . .	22
2.4 KNX development approach . . . . .	23
2.4.1 Fundamental components . . . . .	23
2.4.2 Types of development approach . . . . .	24
<b>3 The KNX Serial Interface</b>	<b>25</b>
3.1 Partial development . . . . .	25
3.1.1 The KNX Certified Digital Transceivers . . . . .	25
3.1.2 KNX stack and it's microcontroller . . . . .	26
3.2 OEM Serial Interface . . . . .	27
3.3 Final Decision for the KNX Serial Interface . . . . .	27
3.4 KNX BAOS Module 832 . . . . .	27
3.5 Communication with the KNX BAOS module 832 . . . . .	28

3.5.1	The data message frame . . . . .	28
3.5.2	BAOS services . . . . .	29
<b>4</b>	<b>MbedOS</b>	<b>33</b>
4.1	Mbed OS 5 . . . . .	33
4.2	Mbed OS 6 . . . . .	33
<b>5</b>	<b>Software</b>	<b>35</b>
5.1	Circular buffers . . . . .	35
5.2	Mapping datapoints to SHC types . . . . .	36
5.3	KNX temperature, dimming and blinds . . . . .	37
5.3.1	Temperature . . . . .	37
5.3.2	Dimming . . . . .	38
5.3.3	Blinds . . . . .	38
5.4	Timing, threads and interrupts . . . . .	39
5.5	Flow of the data . . . . .	40
5.5.1	Outgoing data . . . . .	40
5.5.2	Incoming data . . . . .	40
5.5.3	Configuration settings from KNX . . . . .	41
5.6	Data stored in the SHC . . . . .	41
5.7	Code documentation . . . . .	42
<b>6</b>	<b>Product database entry</b>	<b>43</b>
6.1	Creating a product database entry . . . . .	43
6.2	Parts of the application program . . . . .	43
6.3	Maximum amount of parameters . . . . .	44
6.4	Testing in ETS . . . . .	44
<b>7</b>	<b>Conclusion</b>	<b>47</b>
	<b>Bibliography</b>	<b>50</b>
<b>A</b>	<b>Doxygen</b>	<b>51</b>

# List of Tables

2.1	SHC inputs and outputs [1]. . . . .	18
3.1	Differences between brands of KNX certified digital transceivers [2, 3, 4]. . . . .	26
3.2	BAOS module 832 communication services [5]. . . . .	29
3.3	Details of a data message frame [5]. . . . .	30
3.4	Byte allocation for datapoint services [5]. . . . .	30
3.5	Byte allocation for the parameter byte request service [5]. . . . .	30
3.6	Byte allocation for the parameter byte request service [5]. . . . .	31





# List of Figures

2.1	SHC overview [6]. . . . .	17
2.2	KNX TP signal shape [7]. . . . .	19
2.3	KNX TP telegram structure [7]. . . . .	20
2.4	KNX TP collision avoidance [7] . . . . .	21
2.5	Screenshot of a group address in the group addresses panel in ETS 6 . . . . .	21
2.6	Screenshot of the configuration panel of a push button switch in ETS 6 . . . . .	22
2.7	Notable differences between tiers of KNX membership [8] . . . . .	23
3.1	TP/E981.23 (a), NCN5130 (b) and TP-UART2+ (c) block schemes [2, 3, 4]. . . . .	26
3.2	KNX Baos Module 832 (a), its dimensions (b) and its pin assignment (c) [9]. . . . .	28
3.3	Structure of BAOS module 832 data message [5]. . . . .	28
3.4	Structure of BAOS module 832 reset messages [5]. . . . .	28
5.1	Screenshot of part of the datapointToSHCMap array. . . . .	36
5.2	KNX 2-Octet Float Value bit assignment and details [10] . . . . .	37
5.3	KNX four bit assignment and details of dimming type. [10] . . . . .	38
5.4	Dataflow of the incoming and outgoing KNX serial data . . . . .	40
6.1	Screenshot of the SHC configuration panel in ETS . . . . .	45



# Glossary

Symmetrical wire communication	In symmetrical wire communication, data is transmitted using two wires, with each wire carrying a signal that is the mirror image of the other. This means that the voltage on one wire is equal and opposite to the voltage on the other wire, which helps to cancel out any interference or noise that may be present in the communication channel.
--------------------------------	---



# Abstract

This master's thesis explores the integration of the KNX bus system, a widely used home automation protocol, into the Smart Home Controller (SHC) developed by Bits & Bytes. The SHC allows for the automation of various home features via a smartphone app, switches or other means. By integrating the KNX system, which connects a variety of devices with a single cable, the SHC can expand its capabilities to control and be controlled by KNX devices, potentially increasing its market share.

The research methodology consists of three main steps: exploring options to develop or acquire a transceiver that converts 30 V KNX telegrams to 3.3 V serial data, expanding the SHC code to handle this data while minimizing its impact on existing functionality, and creating a product database entry for KNX to enable programming and configuration within the KNX programming software ETS.

This study reveals that the Weinzierl Baos 832 module is the most suitable transceiver option. To achieve integration, the Weinzierl BAOS module is used along with an expanded SHC code and a newly created product database entry. As a result, the SHC's inputs and outputs can be programmed within ETS, with additional configuration options for the output relays and temperature sensors. These additional configuration options allow the SHC module to be fully configured in ETS, eliminating the need for external tools.



# Abstract in het Nederlands

Deze masterproef onderzoekt de integratie van het KNX-bussysteem, een veelgebruikt domotica-protocol, in de Smart Home Controller (SHC) ontwikkeld door Bits & Bytes. De SHC maakt het mogelijk om verschillende huisfuncties te automatiseren via een smart phoneapp, schakelaars of andere manieren. Door het KNX-systeem, dat verschillende apparaten met één kabel verbindt, te integreren, kan de SHC zijn mogelijkheden uitbreiden om zowel KNX-apparaten aan te sturen, als door KNX-apparaten aangestuurd te worden, wat het marktaandeel van de module kan vergroten.

De onderzoeksmethodologie bestaat uit drie belangrijke stappen: het onderzoeken van opties om een transceiver te ontwikkelen of aan te schaffen die 30 V KNX-telegrammen omzet in 3,3 V seriële data, het uitbreiden van de SHC code om deze data te verwerken terwijl de impact op de bestaande functionaliteit geminimaliseerd wordt, en het creëren van een product database entry voor KNX om programmering en configuratie binnen de KNX-programmeersoftware ETS mogelijk te maken.

Deze studie toont aan dat de Weinzierl Baos 832-module de meest geschikte transceiveroptie is. Voor de integratie wordt de Weinzierl Baos-module gebruikt samen met een aangevulde SHC code en een nieuw gecreëerde product database entry. Hierdoor kunnen de in- en uitgangen van de SHC binnen ETS worden geprogrammeerd met extra configuratiemogelijkheden voor de uitgangsrelais en temperatuursensoren. Dankzij deze extra configuratieopties kan de SHC-module volledig in ETS worden geconfigureerd zodat er geen externe tools meer nodig zijn.





# Chapter 1

## Introduction

Bits & Bytes (B&B) is an IT company that specializes in developing both hardware and software for home automation. Their primary module for managing home devices is called the Smart Home Controller (SHC). It enables control of a variety of devices, including blinds, lights, switches, outlets, electrical appliances and climate control. It can function either as a standalone module or as part of the B&B home system.

The standalone SHC module is designed for small houses and apartments, but can be expanded with up to 16 SHCs to accommodate more devices. For high-end homes, the B&B home system expands upon the SHCs with the B&B Server, offering additional features like touchscreen panels and integrated security devices such as alarms and cameras. The touchscreen panel provides an easy-to-use interface for managing connected devices, simplifying the process of home automation. The B&B Home System is built to centralize control over various aspects of home automation, integrating different devices and systems like lighting, temperature control, and security into a single platform.

Bits & Bytes targets newly built and renovated homes for their smart home system, as it requires specific wiring for each device to connect to the SHCs. Electricians experienced in smart homes typically prefer the KNX system, which uses a bus to connect all devices with just one wire, resulting in greater flexibility and reduced labor and wiring costs. This standardised system is also recommended by architects. To meet this requirement, the B&B Server can be coupled to the KNX bus. The KNX addresses can then be manually linked to the B&B Server, which allows KNX output modules to be controlled by the B&B Home System.

The current setup poses a challenge for programmers, as it requires them to individually program the B&B Home System, set the linked address, and program the KNX device for each automation, resulting in a tedious and inefficient process. Furthermore, because standalone SHCs currently do not support KNX, and third-party KNX modules must be purchased to integrate it in the B&B Home System, Bits & Bytes has decided to integrate KNX into the SHC.

The integration of KNX functionality into the SHC provided by Bits & Bytes has the potential to increase the device's market potential as home automation system by including native KNX support. To accomplish this objective, this study seeks to identify an appropriate KNX Serial Interface for the SHC, with a particular emphasis on minimizing the impact on the SHC's already-existing program during software development to ensure effective and efficient performance.

While the primary objective of this study is to integrate KNX specifically into the SHC, it is important to note that the integration of the Serial Interface and the development of a low-impact software design can serve as a foundation for the development of future devices. The knowledge gained from this study can be applied to other home automation systems and contribute to the advancement of the field, ultimately leading to a more comprehensive and user-friendly home automation experience.

This study is structured into several chapters, each focusing on specific aspects of integrating KNX into the SHC. Chapter 2 provides an overview of the SHC's functions and the desired functions accessible through KNX. It also explores the workings of the KNX protocol, the KNX programming software ETS, and the certification and development requirements.

Chapter 3 focuses on the KNX Serial Interface, a crucial component for reliable communication between the KNX bus and the SHC's processor. It explains the steps involved in developing and acquiring OEM equipment for the interface, including the decision-making process and the operation of the selected KNX Serial Interface.

Chapter 4 describes the step-by-step approach taken to integrate KNX functionality into the SHC. It discusses running the KNX program on a development board using Mbed Studio, covering the software setup process and the integration of the KNX library into the SHC.

Chapter 5 explores key aspects of the expanded code for the SHC. It addresses the implementation of circular buffers, mapping data points to SHC types, and handling KNX commands related to temperature, dimming, and blinds. Additionally, it discusses resolving timing issues through the use of threads and interrupts, and describes the flow of data and code documentation.

Chapter 6 focuses on the development of a product database entry for the SHC. This entry serves two purposes: enabling the SHC to be imported as a KNX device and allowing its I/O to be connected to other KNX devices, and providing convenient access to various configuration settings through ETS.

Chapter 7 concludes this study by summarizing the key findings and insights from the previous chapters and providing recommendations for future research and development in the field of home automation.

# Chapter 2

## KNX and the Smart Home Controller

This chapter explores SHC and the KNX protocol. It will provide a brief summary all the functions of the SHC and desired functions accessible through KNX. Additionally, the chapter will delve into the workings of the KNX protocol, the KNX programming software ETS and KNX certification and development requirements.

### 2.1 Introduction to the Smart Home Controller

The SHC is a DIN rail module for automating home devices. It can be used in a Bits & Bytes Home system or as a standalone device controlled by a smartphone. This section will explore all functions of the smart home controller and the desired functions accessible through KNX. Figure 2.1 shows the SHC with an overview of its features.

The SHC uses the LPC 4088 microprocessor, which is based on the Arm Cortex M4.

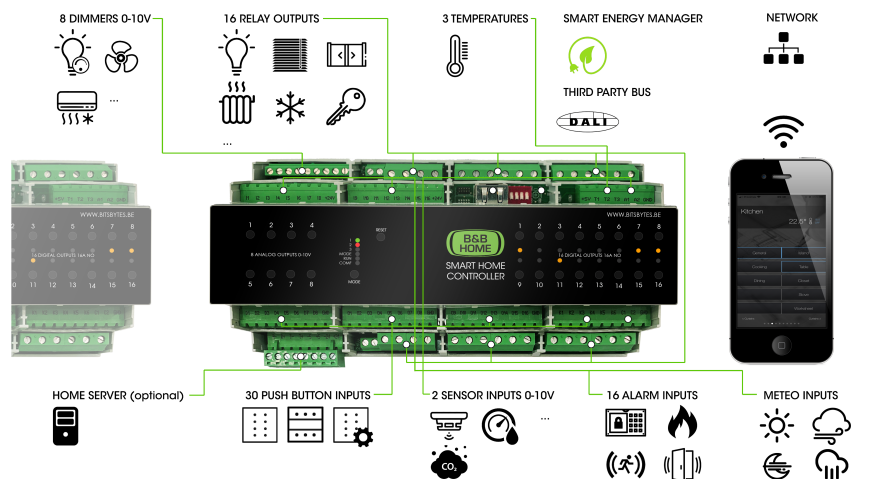


Figure 2.1: SHC overview [6].

#### 2.1.1 SHC features in KNX

Table 2.1 shows the inputs and outputs (I/O) of the SHC. Upon completion of this study, it is expected that all outputs will also be controllable through KNX, as well as showing the status of inputs and outputs in KNX [1].

Table 2.1: SHC inputs and outputs [1].

I/O amount and name	Type of I/O
16 digital outputs	16 A 230V relay outputs
8 dimmer outputs	0-10 V outputs
6 comfort keys	Push button inputs
16 digital inputs	Push button inputs
8 dimmer inputs	Push button inputs
3 temperature inputs	Sensor inputs
2 analog inputs	0-10 V inputs
2 counter inputs	Counter inputs
2 indicator outputs	Open collector outputs

In addition to the I/O options mentioned, Bits & Bytes has requested a feature that enables receiving temperature values from KNX. These values are intended to replace the readings obtained from the temperature inputs. This functionality allows for the utilization of KNX temperature sensors with a B&B Home System, as the SHC transmits its temperature readings to the server.

Programming the SHC is possible with the SHC Configuration Tool. This tool is explained in more detail in chapter 6. Another goal of this study is to make both the feature that enables receiving temperature values from KNX and the relay configuration options programmable from within the KNX programming software ETS.

### 2.1.2 Expandability of the SHC

To ensure future expansion possibilities for the SHC, its development was executed with a forward-looking approach. As part of this approach, the printed circuit board (PCB) was designed to include an expansion space with serial connections, a KNX connector, a button and a led. The expansion slot layout was deliberately selected to fit the Weinzierl Baos module that was available at the time. This approach allowed for integration of future KNX expansions while maintaining the SHC's functionality and compatibility with other serial devices.

## 2.2 Introduction to KNX

The KNX protocol is an open standard, overseen by the KNX Association, that governs the development, promotion, and maintenance of KNX technology. KNX communication protocols and standards are open and accessible to any manufacturer or developer who wishes to create compatible products. This open approach has enabled the production of a diverse array of KNX-certified devices by numerous manufacturers, affording users extensive flexibility and a wide range of options when seeking building automation solutions. This section will delve more deeply into the details of KNX device communication and functionality.

### 2.2.1 The KNX bus system

The KNX system is a building control bus system that uses a standardized communication protocol to enable interoperability among its devices. The devices in a KNX system are equipped with microprocessors that enable the exchange of data packets, also known as telegrams, via a common bus network. The telegrams can contain diverse types of information, such as status

updates, control commands, and configuration data. The communication of data between devices within a KNX system can be facilitated through a range of communication media, each with distinct transmission methods. These media include KNX Twisted Pair (TP), which utilizes a TP data cable, known as a bus cable. Additionally, KNX Powerline is another option that employs the existing 230 V mains network. KNX Radio Frequency represents another communication medium, where radio signals are used for data exchange. Finally, KNX IP is a communication medium that relies on Ethernet for data transmission [7].

As [1] states TP is the most common communication medium for KNX installations, this study will focus on twisted pair communication between KNX and the SHC. It's worth noting that, thanks to the bus system, any devices that require a different type of communication medium could simply be linked to the TP bus.

### 2.2.2 KNX Twisted Pair Telegrams

In a KNX TP system, the bus cable serves as both the communication medium and the power source for all connected devices. The nominal voltage of the bus is 24 V, but power supplies typically provide a higher voltage of 30 V to account for voltage drops and contact resistance in the cable. To ensure reliable operation, KNX devices are designed to function within a voltage range of 21 V to 30 V. This provides a tolerance range of 9 V to compensate for potential voltage drops and fluctuations in the cable. This enables the KNX system to maintain stable communication and power delivery to all connected devices, even in case of minor voltage drops or variations in the cable [7].

The KNX TP uses asynchronous data transfer to send data one byte at a time with a data transfer rate of 9600 bit/s. Communication is symmetrical and non-earthed, which increases stability against interference signals. The transmitter creates the AC voltage corresponding to the logical zero by lowering the voltage difference between the cables with around 5 V and then cancels this voltage drop after approximately half a bit period. The rest of the system generates a positive compensating pulse to balance the voltage drop. Figure 2.2 shows that the sequence of the drop followed by the spike is interpreted as a zero, while a flat signal is interpreted as a one [7].

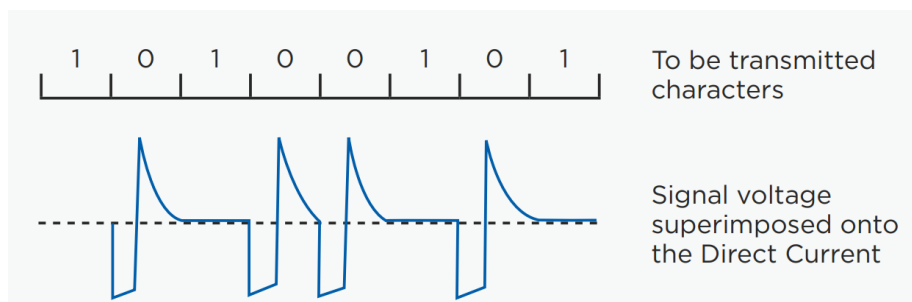


Figure 2.2: KNX TP signal shape [7].

Figure 2.3 shows the structure of a KNX TP telegram. It is build up out of 4 fields:

- The one byte Control field defines the priority of the telegram and whether or not transmission of the telegram was repeated.

- The five byte address field specifies the individual address of the sender and the destination address
- The 1-16 byte data field contains the telegram's data
- The one byte checksum field is used for parity checks

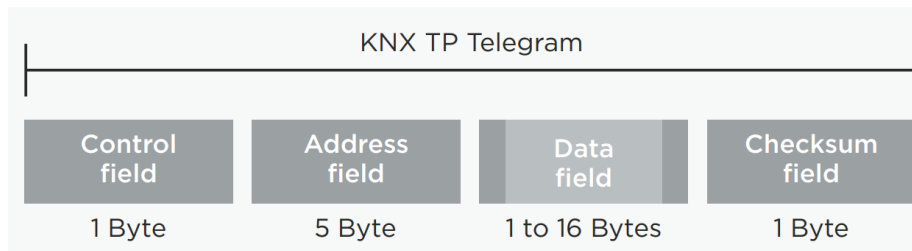


Figure 2.3: KNX TP telegram structure [7].

[8] contains more detailed information about each field.

To ensure reliable transmission, each byte of telegram in the KNX TP system is accompanied by three parity bits. For a telegram with up to six bits of data, the data field is two bytes, which results in a 9 byte telegram. Including the one byte acknowledgement signal that a device sends upon receiving the signal, the minimum theoretical transmission time can be calculated as:

$$\frac{Bytes_{total} \cdot 11 \text{ Bits}}{Baudrate} = \frac{10 \cdot 11 \text{ Bits}}{9600 \text{ Bits/s}} = 11.5 \text{ ms} \quad (2.1)$$

for a single telegram. This is within the specification, as [5] states a typical telegram in normal traffic needs around 20 ms. [7] also states a KNX TP Bus can send a maximum of 50 telegrams per second [5, 7].

In the KNX bus system, as in several other bus systems, access to the bus is random and event-driven. A telegram can only be transmitted if no other telegram is currently being transmitted. To avoid collisions during transmission, the priorities of the sending devices are regulated, as illustrated in Figure 2.4 [7].

Each transmitting device listens to every bit of data transfer on the bus. If two devices attempt to send a telegram at the same time, a collision occurs, where one sender wants to transmit a zero while the other wants to transmit a one. The device sending the one "hears" that a zero is being transmitted and detects the collision. It is then obligated to abort its own data transmission and give priority to the other transmission. After the higher priority transmission is complete, the aborted data transmission will recommence. The control field of a telegram can define its level of priority. If two telegrams have the same level of priority, their physical addresses are used to determine which telegram has priority (zero has priority over one) [7].

### 2.2.3 The KNX ETS Software

KNX devices require programming before they can perform any functions. During the programming process, a software group address is created and devices are added to this address. Once devices are added, they can be configured to suit the specific needs of the installation.

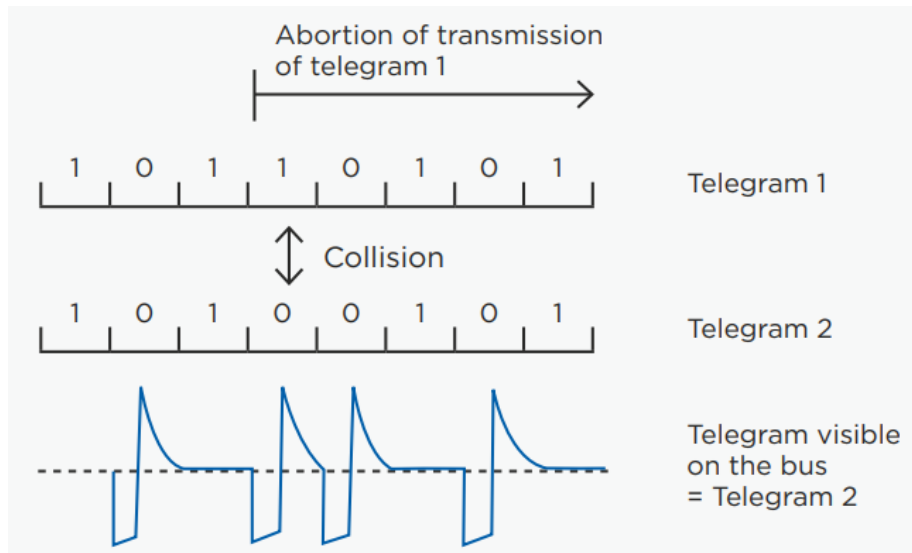


Figure 2.4: KNX TP collision avoidance [7]

The most important properties for KNX devices are the write and transmit flags. The write flag enables the value of a device to be modified via a telegram, while the transmit flag sends a telegram when the device's value changes due to an internal event. Figure 2.5 shows a screenshot of the group address panel in ETS 6. It shows the group address 0/3/0 with five objects: four output leds and one digital input signal (Digin 1). The write and transmit flags are abbreviated to W and T. This group address will result in the leds being controlled by the input signal. Figure 2.6 shows a screenshot of the configuration panel of a push button switch in ETS 6. It shows the configuration options for the push button *Digital Input 1*.

The integration of the SHC and its configuration options within ETS can streamline the adoption process for programmers already familiar with this widely used platform. It offers an opportunity for programmers to opt for the SHC over alternative KNX modules, potentially enhancing the popularity and market share of the SHC. Because of this, Bits & Bytes aims to provide configuration functionality for the SHC within ETS, comparable to the configuration panel shown in Figure 2.6. By integrating the SHC's configuration capabilities directly within ETS, KNX programmers will be able to program the SHC within this environment, without the need for the SHC configuration tool. This has the potential to simplify the configuration process and enhance the SHC's usability within ETS. For additional information on this topic, please refer to Chapter 6.

Object	Device	Sending	Data Type	C	R	W	T	U	Product	Program	Length
24: <Output 1> Led - Off/On/Blink1.1.2 Inwall 8 Input / 4 Output...		S	switch	C	-	W	-	-	Inwall 8 Input / 4 O...	AD84A02KNX	1 bit
25: <Output 2> Led - Off/On/Blink1.1.2 Inwall 8 Input / 4 Output...		S	switch	C	-	W	-	-	Inwall 8 Input / 4 O...	AD84A02KNX	1 bit
26: <Output 3> Led - Off/On/Blink1.1.2 Inwall 8 Input / 4 Output...		S	switch	C	-	W	-	-	Inwall 8 Input / 4 O...	AD84A02KNX	1 bit
27: <Output 4> Led - Off/On/Blink1.1.2 Inwall 8 Input / 4 Output...		S	switch	C	-	W	-	-	Inwall 8 Input / 4 O...	AD84A02KNX	1 bit
41: Digin 1 - DPT 1	1.1.1 KNX BAOS 830	S	1-bit, switch C	-	W	T	-	-	KNX BAOS 830	KNX BAOS 83x	1 bit

Figure 2.5: Screenshot of a group address in the group addresses panel in ETS 6



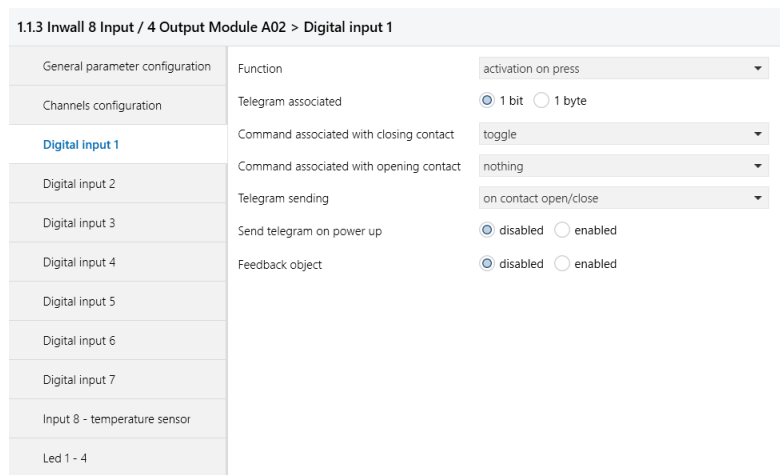


Figure 2.6: Screenshot of the configuration panel of a push button switch in ETS 6

## 2.2.4 Datapoints and parameters

In this study, the terms *datapoint* and *parameters* play a significant role. A datapoint serves as a representation of the value assigned to a group object in ETS. It possesses a unique identifier that distinguishes it from other datapoints. The number of data points thus directly determines the number of objects that a KNX device can recognize and store values for. The identifier assigned to the object is visible in Figure 2.5 in the *object* column. On the other hand, parameters refer to the amount of bytes that a device is capable of retrieving from an ETS configuration panel.

## 2.2.5 Example of KNX communication

In this example, the user configures their light switch to transmit a value of 1 when a button is pressed. This value is stored as a parameter within the light switch. The user then establishes a connection between the button input object of the light switch and the light output object of a KNX dimmer by putting them together in a group address. To implement this functionality, the user downloads the configuration to the device using ETS .

When the button is pressed, the light switch sends a telegram that includes the group address and a data value of 1. The KNX dimmer receives this telegram, recognizes the group address, and updates the corresponding data point value associated with its light output object to 1. What the dimmer does when its datapoint value is changed depends on the dimmer's configuration and programming.

## 2.3 KNX Certification

KNX certification and developer tools are limited to KNX Association members, but KNX products can still be used and integrated within a KNX system without certification or developer tools.

### 2.3.1 Membership

The KNX Association offers three tiers of membership:

- Interested party

- Licensee
- Shareholder

Figure 2.7 shows some of the notable differences between the tiers of membership. The different tiers come with different pricing. The interested party membership only offers access to developer tools and documentation, while licensee and shareholder provide access the developer tools, documentation and device registration and certification. More details on the memberships and pricing is available at [8].

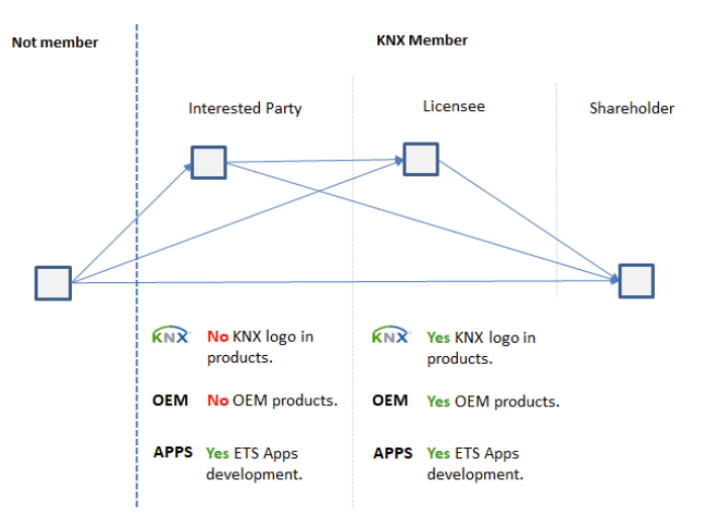


Figure 2.7: Notable differences between tiers of KNX membership [8]

During this study, Bits & Bytes chose to apply for the *interested party* membership for the first year, as it provided them with the ability to develop ETS applications and access the KNX development tools. They did not see the need to obtain certification for products quickly and wanted to explore the KNX tools and development steps before committing to a longer and more costly membership.

## 2.4 KNX development approach

A KNX device is composed of a set of fundamental components that serve as its building blocks. These essential components, including the transceiver, microcontroller, stack, PCB, and product database entry, form the backbone of the device’s functionality. However, what sets each KNX device apart is the ability to expand upon these components by adding additional features and functionality. This flexibility allows manufacturers to meet specific requirements and differentiate them from others in the market.

### 2.4.1 Fundamental components

The microcontroller is responsible for executing the device’s embedded software and controlling its operations. It handles tasks such as processing data, managing inputs and outputs, and interacting with other components of the device.

The KNX stack refers to the microcontroller software that implements the KNX communication protocol. It enables the device to send and receive messages according to the KNX standard, ensuring compatibility and interoperability within the KNX network.

The PCB serves as the physical layer of the KNX device. It provides a platform for integrating and connecting various electronic components, including the transceiver, microcontroller, and other functionalities specific to the device.

The product database entry allows manufacturers to create and configure ETS functionality for their device, such as configurable parameters and datapoints. This entry includes information about the device's functionalities, parameters, and communication behavior. Once a device is certified and its product database entry is created, any changes to it requires re-certification as a new product.

KNX devices can incorporate additional features such as buttons, sensors, actuators, and other peripherals. These features are integrated into the physical layer, usually through the PCB design, to enable specific functionalities in the device.

An ETS app is not a part of a KNX device, but refers to a software application that provides access to the ETS data and product database entries. It is designed to be downloaded and integrated into the ETS software, expanding its functionality and allowing users to interact with and manipulate the product database entries. The ETS app is not to be confused with the application program, which is a part of the product database entry.

It should be noted that using certified KNX components or devices may require contacting the manufacturer of the device.

## **2.4.2 Types of development approach**

Three different approaches can be taken to develop a KNX device: Full development, partial development and OEM acquisition.

In OEM acquisition, only minimal changes are made to an existing KNX device. These changes involve aspects such as the device's exterior design, manufacturer branding, and potentially reducing certain software parameters. The OEM approach only requires registration and no certification.

The partial development approach involves utilizing existing KNX components, such as the KNX stack or transceiver while potentially developing other parts from scratch. This approach allows the reuse of certified components. Only the final device, along with any components developed from scratch, need to undergo certification. When certifying a device, an accompanying product database entry is necessary, which links to the device, and only certain descriptions can be altered without requiring re-certification.

In the full development approach, all the components of the KNX device, including the transceiver, KNX stack and application program, are created from scratch. This approach requires each individual component to undergo the certification process to ensure compliance with the KNX standard and ensures that the manufacturer is fully independent.

# Chapter 3

## The KNX Serial Interface

In order to establish reliable communication between the KNX bus and 3.3 V serial , a KNX Serial Interface is a crucial component. It converts all incoming and outgoing signals of a KNX telegram into a digital format that can be read by the SHC's processor and vice versa, as well as handle all KNX TP protocols. This chapter will explore the steps needed to partially develop and acquire OEM equipment for a KNX Serial Interface. The selection of a KNX Serial Interface involves both the decision of whether to partially develop the interface or to acquire an OEM product. Furthermore, this chapter will also explain operation of the selected KNX Serial Interface.

### 3.1 Partial development

Partial development of a new KNX device requires creating a PCB with KNX certified components. For the Serial Interface, it involves the identification and selection of a KNX digital transceiver, as well as the evaluation of an appropriate microcontroller and software, commonly referred to as the KNX stack. It enables hardware customization, which is not possible when acquiring an OEM module.

#### 3.1.1 The KNX Certified Digital Transceivers

A digital transceiver is the integrated circuit (IC) that converts the data from the 30 V KNX bus to a 3.3 V digital signal. [11] offers a list of all certified KNX system components. These components include three brands of digital transceivers for TP. The transceivers with the most features for each brand are:

- The TP/E981.23 by Elmos
- The NCN5130 Series by ON Semiconductor
- The TP-UART2+ by Siemens

Figure 3.1 gives a side-by-side comparison of block diagrams for the different transceivers. This shows how each model is put together and what sets them apart from one another. Since the schemes portray blocks at different levels of detail, they cannot be compared reliably [2, 3, 4].

All three transceivers run on KNX bus power, support SPI and UART communication, and can generate external clocks while also featuring a 20 V regulator. However, they differ in their baud

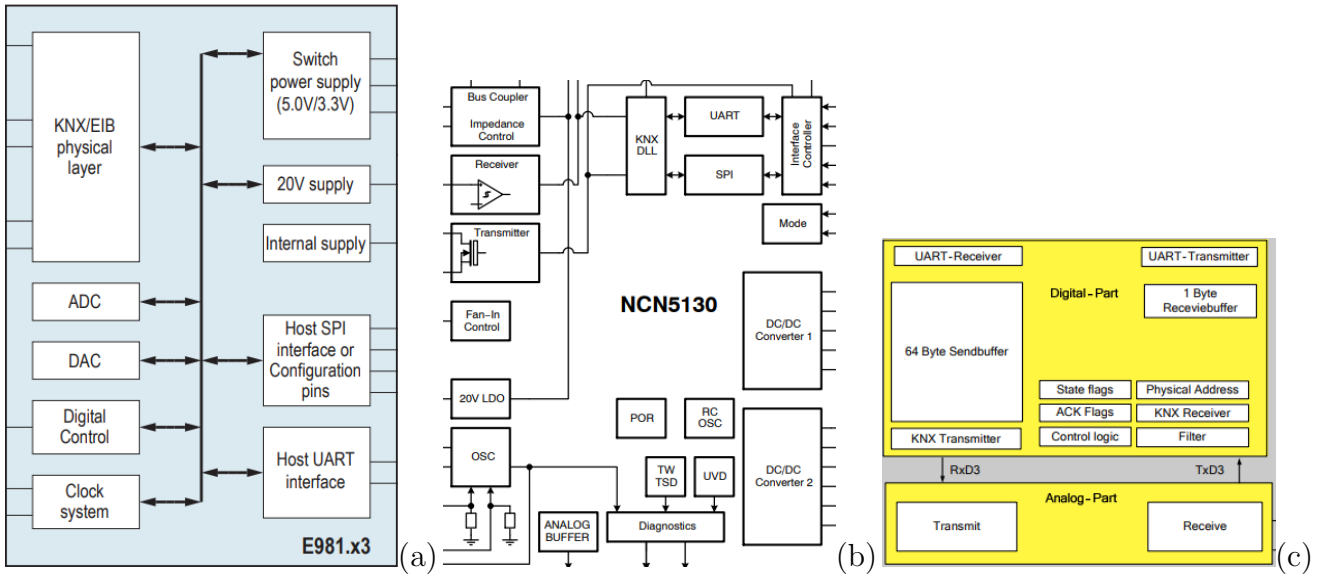


Figure 3.1: TP/E981.23 (a), NCN5130 (b) and TP-UART2+ (c) block schemes [2, 3, 4].

rates, power supply capabilities and clock frequencies. The NCN has a fixed 3.3 V buck converter and one that can be set between 1.2 V and 21 V, supports baud rates of either 38400 or 19200 and provides an external clock of 8 or 16 Mhz. The Tp-uart has a shunt converter that can toggle between 3.3 V and 5 V, and an external clock of 4.9152 Mhz. The TP/E981.23 has a buck converter that can toggle between 3.3 V and 5 V and an external clock of 8 or 7.3728 Mhz. This is summarized in Table 3.1 [2, 3, 4].

Table 3.1: Differences between brands of KNX certified digital transceivers [2, 3, 4].

	NCN5130	TP-UART2+	TP/E981.23
Serial interface	Spi/Uart	Spi/Uart	Spi/Uart
Baud rate	19200/38400	19200/115200	19200/115200
External clock	8/16 Mhz	4.9152 Mhz	8/7.3728 Mhz
Voltage source 1	3.3 V buck	3.3/5 V shunt	3.3/5 V buck
Voltage source 2	1.2-21 V buck	/	/
Voltage regulator	20 V	20 V	20 V

The datasheets for both the NCN and TP-UART transceivers contain sample PCB layouts for various digital transceiver applications. However, information regarding the TP/E981.23 was not readily available since its datasheets were only accessible through a request form. [2, 3, 4].

### 3.1.2 KNX stack and it's microcontroller

The KNX stacks refers to the software of the microcontroller connected to the digital transceiver. The microcontroller is responsible for executing the software that implements the KNX protocols necessary for the communication between KNX devices. The software includes various functions, such as addressing, device discovery, and data transmission, that ensure effective communication between devices in the KNX system. In terms of microcontroller compatibility, the ISE stack is the most appropriate choice for 8-bit microcontrollers due to its low memory and processing requirements. Meanwhile, the Weinzierl stack is optimized for 32-bit microcontrollers, which offer more powerful computing capabilities. Additionally, Tapko's modular stack provides flexibility and versatility, allowing it to work with various microcontroller types or even on 64-bit processors.

The remaining certified KNX stacks available at [11] might also be suitable, but might not be for sale [11].

## 3.2 OEM Serial Interface

Acquiring an OEM Serial Interface skips most KNX device development, requires no certification and thus has a fast time-to market.

Two OEM manufacturers offer Serial Interfaces: The KNX BAOS Module 83x by Weinzierl and the SIM-KNX250 by Tapko. The Weinzierl outperforms the Sim-Knx in terms of overall specifications. Specifically, the former offers a significantly higher number of datapoints with 1000 compared to the latter's 250. Moreover, the BAOS 83x provides basic support for an additional 250 parameters, which can be further expanded by utilizing available memory, thus enabling it to be expandable to more than 70,000 parameters. Furthermore, the Weinzierl is comparatively more cost-effective, with a price of 35 euros in contrast to 100 euros for the Sim-Knx at the time of writing. Overall, the Sim-Knx appears to be more of a demonstration or an introductory product to the Tapko KNX stack, rather than a fully-fledged KNX product [5, 12].

The KNX BAOS Module 83x refers to the 832 and 830 modules, which have different PCB layouts and dimensions, but the exact same functionality. The only difference between them is that the 832 module is galvanically isolated from the KNX bus. The dimensions of the 832 are approximately 25 x 29 x 12 mm, while the dimensions of the 830 are approximately 25 x 44 x 8 mm. The dimensions and pin layout of the 832 module are identical to the module referenced in Section 2.1.2, which was the layout used for designing the SHC expansion space. [13, 9].

## 3.3 Final Decision for the KNX Serial Interface

Partially developing an interface allows for more control over the design and customization of the interface, but also requires a significant development time. Acquiring an OEM product, on the other hand, offers a more cost-effective and time-efficient solution, but may require compromises in terms of customization and compatibility. After thorough consideration, the KNX BAOS Module 832 emerged as a compelling option for this study, given its competitive pricing and comprehensive functionality.

As such, this study will focus on the implementation of the KNX BAOS Module 832 within the SHC, which enables a fast time-to-market.

## 3.4 KNX BAOS Module 832

The KNX BAOS Module 832 is shown in Figure 3.2. From this point onward, the module will be referred to as the BAOS module. BAOS stands for Bus Access and Object Server. The module allows the access to the KNX bus on telegram level as well as on datapoint level, but only the datapoint level will be used. The connection between application and KNX BAOS Module is established via a UART connection using FT1.2 framing. The module offers 1000 configurable datapoints and 250 parameters out of the box, with the option of expanding the parameter

amount by editing the product database entry. It communicates on a baud rate of 19200 or 115000 Bit/s and the 3.3 V and 20 V power supply the module offers will not be utilized [9].

Each datapoint the module offers has its own group object in ETS, where the group object's value is saved in the datapoint.

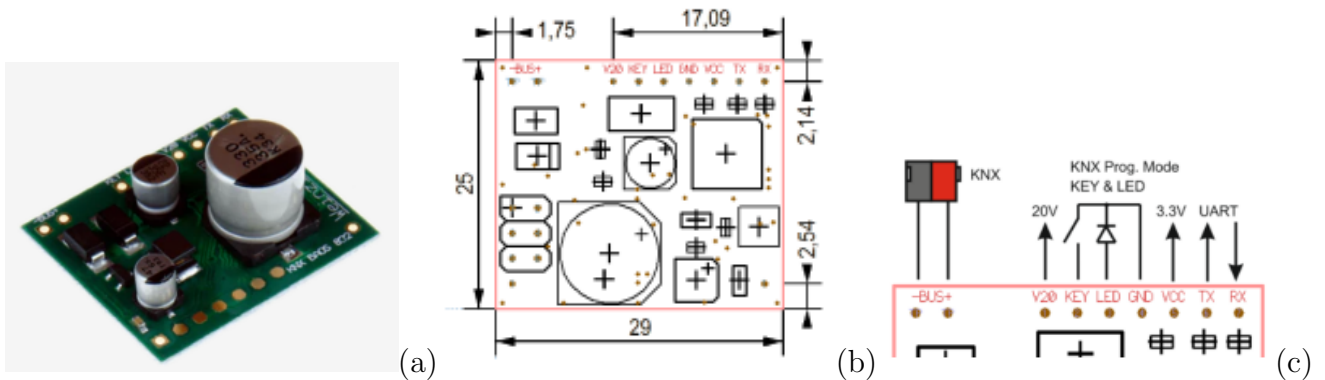


Figure 3.2: KNX Baos Module 832 (a), its dimensions (b) and its pin assignment (c) [9].

### 3.5 Communication with the KNX BAOS module 832

The FT1.2 protocol and communicates through serial using three types of frames [9]. Data messages use the frame structure shown in Figure 3.3, while the remaining two frames are utilized for reset and acknowledge messages. Reset messages, as illustrated in Figure 3.4, come in two types: a reset request, which initiates a reset when triggered, and a reset indication, which signals that a reset has occurred without being prompted by a request. Acknowledge messages consist of only the byte 0xE5.

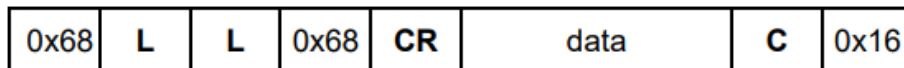


Figure 3.3: Structure of BAOS module 832 data message [5].

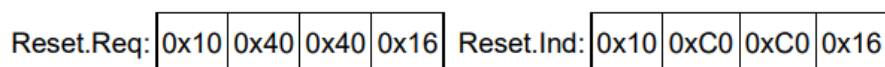


Figure 3.4: Structure of BAOS module 832 reset messages [5].

#### 3.5.1 The data message frame

This section explains how the data is structured within the frame, while section 3.5.2 explains how the data is structured for each BAOS service.

The data frame exists out of seven bytes and the data bytes to transfer. This makes a that a valid message contains at least seven bytes when no data is present. The maximum size of a message is not defined. Figure 3.3 shows the structure of the data frame and Table 3.3 shows the details of the data frame. The byte index in the table begins at zero to replicate how the bytes would be arranged in an array.

The CR byte serves as the control byte for the frame. Following a reset of the BAOS device, the CR value must be set to 0x73 for all odd frames and 0x53 for all even frames send to the BAOS device. All messages received from the device have a CR of 0xF3 for the odd frames and 0xD3 for the even frames.

The checksum byte C is the sum of all data bytes and the CR byte, and in the event of an overflow during the calculation, the value wraps around to zero and the calculation continues.

### 3.5.2 BAOS services

The BAOS module 832 features a communication protocol that provides nine distinct services. These services and their brief descriptions are outlined in Table 3.2.

Table 3.2: BAOS module 832 communication services [5].

Service	Description
GetServerItem.Req/Res	Gets settings from the module
SetServerItem.Req/Res	Sets settings on the module
ServerItem.Ind	Automatic Ind that a server item has changed
GetDatapointDescription.Req/Res	Gets datapoint parameters
GetDescriptionString.Req/Res	Gets datapoint descriptions
GetDatapointValue.Req/Res	Gets datapoint values
DatapointValue.Ind	Automatic Ind that a datapoint value has changed
SetDatapointValue.Req/Res	Sets datapoint values
GetParameterByte.Req/Res	Gets parameter bytes

Req = request, Res = response, Ind = Indication

Since the *DatapointValue* Indication automatically sends a message upon detecting a change in a datapoint value, the need for a *GetDatapointValue* Request is unnecessary. Thus, the Indication can serve as the only means of communication from the BAOS module to the SHC. To transmit values from the SHC to the BAOS module, the *SetDatapointValue* request is used. Additionally, the *GetParameterByte* service is used to retrieve the SHC configuration bytes from KNX, which enables the modification of the SHC's settings directly within KNX, eliminating the need for the SHC Configuration Tool. It should be noted that the other services were not required in this study [5].

The *DatapointValue* indication and *SetDatapointValue* request services use the same byte indexes except for the sub service byte. This is shown in table 3.4, with information about the byte values utilized for communication with the SHC . The byte indexes in the table start at index 5 to match the byte indexes of table 3.3. The command byte for a *SetDatapointValue* is always 0x03, as this corresponds to setting a new datapoint value and sending it on the bus. The state byte replaces the command byte in an *DatapointValue* indication and contains information about the state of the datapoint. Other command byte values are not used but can be found in [5].

The *SetDatapointValue* request is always expected to succeed, therefore the corresponding response indicating success or failure from the BAOS module is ignored. However, during debugging scenarios, the response could prove to be a source of information. By inspecting the error code located in the last byte of the response, the error information can be obtained. Further details regarding the error codes can be found in [5].



Table 3.3: Details of a data message frame [5].

Byte Index	Value	Description
0	0x68	Start of frame and frame header
1	L	Length of D + 1 (CR)
2	L	Length of D + 1 (CR) again
3	0x68	End of frame header
4	CR	Details in section 3.5.1
5 to 4+D	Data	Data of size D (in bytes)
6 + D	C	Checksum of all data and CR bytes
7 + D	0x16	End of frame

Table 3.4: Byte allocation for datapoint services [5].

Byte Index	Value	Description
5	0xF0	Main service code (always same value)
6	0xC1 or 0x06	Subservice code for indication (0xC1) or request (0x06)
7,8	ID <sub>1</sub>	ID of first datapoint (in a 16 bit uint)
9,10	N	Amount of datapoints in this message (in a 16 bit uint) N
11,12	ID <sub>1</sub>	ID of first datapoint (in a 16 bit uint) ID <sub>1</sub>
13	0x03	Command or state byte, details in section 3.5.2
14	L <sub>1</sub>	Length of first data value in bytes (8 bit uint) L <sub>1</sub>
15	V <sub>1</sub>	First data value V <sub>1</sub>
...	...	...
I <sub>N</sub> , I <sub>N</sub> +1	ID <sub>N</sub>	ID of N <sup>th</sup> datapoint (in a 16 bit uint) ID <sub>N</sub>
I <sub>N</sub> +3	0x03	Command byte to change datapoint value and send it on bus
I <sub>N</sub> +4	L <sub>N</sub>	Length of N <sup>th</sup> data value in bytes (8 bit uint) L <sub>N</sub>
I <sub>N</sub> +5	V <sub>N</sub>	N <sup>th</sup> data value V <sub>N</sub>

$I_N = \text{Byte index of } D_{N-1} + L_{N-1}$

Parameter bytes are bytes in the BAOS module memory that are programmable in ETS, as explained in chapter 6. The *GetParameterByte* request can be used to get these parameters bytes from the BAOS module. This is used to make the SHC configurable from within ETS. Table 3.5 shows the byte allocation for the request. The request triggers a response from the BAOS module which is shown in Table 3.6.

Table 3.5: Byte allocation for the parameter byte request service [5].

Byte Index	Value	Description
5	0xF0	Main service code (always same value)
6	0x07	Subservice code
7,8		Index of first parameter byte to return (in a 16 bit uint)
9,10		Amount of parameter bytes to return (in a 16 bit uint)

Table 3.6: Byte allocation for the parameter byte request service [5].

Byte Index	Value	Description
5	0xF0	Main service code (always same value)
6	0x87	Subservice code
7,8		Index of first parameter byte to return
9,10	N	Amount of datapoints in this message (in a 16 bit uint) N
11		First parameter byte
...	...	...
10+N	N	Last parameter byte



# Chapter 4

## MbedOS

To achieve integration of KNX functionality into the SHC, a step-by-step approach was taken. The initial step involved running the KNX program on a development board to allow faster debugging and programming. Mbed Studio, the official IDE of Mbed, was utilized for the development and building of the program. The software setup process was then divided into three parts: running a program that blinks a led (blinky) on the dev board, running the KNX library on the dev board, and integrating the KNX library into the SHC.

### 4.1 Mbed OS 5

During the study, the SHC used Mbed OS version 5.10.4, while the most recent version available was 6.16.0. To import Mbed OS 5.10.4 into Mbed Studio for building and compilation, the version needed to be retrieved from the Github repository. However, executing any program on the development board using version 5.10.4 was unsuccessful. Consequently, the decision was made to use the latest release from the Mbed OS 5 series, specifically Mbed OS 5.15.9. This alternative version successfully executed an example project on the first attempt [14, 15].

Introducing a newer version of Mbed OS necessitated an update to the SHC's Mbed OS version, specifically to 5.15.9, as this is the latest Mbed OS 5 version at the time of this study [14]. This introduced an additional step to the step-by-step approach.

The process of updating the SHC's Mbed OS version to 5.15.9 involved updating the Mbed OS and I2CEeprom libraries, renaming certain existing variables that were now utilized by these libraries, relocating variable initialization outside of case statements, and modifying certain configuration settings.

### 4.2 Mbed OS 6

Updating the SHC to a version 6.x.x of Mbed OS was also explored. The primary motivations behind this exploration were future-proofing the system and gaining access to the non-blocking I/O functions introduced in Mbed OS 6 [15].

However, the LPC4088 microcontroller used in the SHC is not officially supported by Mbed OS 6. Despite this limitation, the possibility of creating a custom board using LPC4088 and adapting

Mbed OS 6 to work with it was considered. It should be acknowledged that porting Mbed OS 6 to an unsupported platform can be a complex and time-consuming task [16].

The process of adapting Mbed OS 6 for LPC4088 involves firmware development and porting Mbed OS 6 to the LPC4088 microcontroller. In the firmware development stage, a custom Mbed OS target is created for the LPC4088-based board, including the configuration of the Mbed OS target, definition of pin mappings, establishment of clock configurations, and integration of any necessary peripheral drivers specific to the board [15, 16].

The integration of the LPC1768, an LPC board officially supported by Mbed OS 6, was used to simplify the process. By identifying relevant files from the Mbed OS 6 codebase of the LPC1768 board and incorporating them into the custom LPC4088-based board project, the foundation for adaptation was established. Subsequent adjustments were made to these files and configurations to ensure compatibility with the LPC4088 microcontroller and its specific features, such as modifying pin mappings, peripheral drivers, and clock configurations.

However, a significant challenge was encountered with the ethernet drivers in Mbed OS 6. It became apparent that each microcontroller has its own Ethernet MAC (EMAC) drivers in this version. Due to the scope limitations of the study, it was decided that developing these drivers was beyond the intended scope, leading to the decision to drop the update to Mbed OS 6.

Exploring the possibility of upgrading the microprocessor to one supported by Mbed OS 6 was also considered. However, no processors were found that matched the footprint, functionality, and output capabilities of the LPC4088, making this option unviable [17].

# Chapter 5

## Software

This chapter explores key aspects of expanded code for the SHC. These key aspects include the implementation of circular buffers, mapping data points to SHC types, and handling KNX temperature, dimming, and blinds commands. Furthermore, the chapter addresses the resolution of timing issues through the use of threads and interrupts. The chapter also describes the flow of data, including the creation and parsing of outgoing data, the handling of incoming data, and the retrieval of configuration settings from the Weinzierl module during startup. Finally, it provides some details on the programming process and code documentation.

### 5.1 Circular buffers

By using buffers, or circular buffers, data can be received, processed, and executed in different parts of the code, which is particularly important in home automation systems. These systems often experience bursts of information coming in from various sensors and devices. The use of buffers allows the system to maintain its responsiveness without causing delays in its outputs.

The circular buffers used in this study consist of three essential components: a write index, a read index, and a maximum size. The write index indicates the position where the next element will be written, while the read index indicates the position from where the next element will be read. The maximum size represents the total number of elements that the buffer can store.

To maintain the circular behavior of the buffer, the indices are incremented appropriately. When writing data to the buffer, the write index is increased by one. Similarly, when reading data from the buffer, the read index is incremented. However, with circular buffers, a crucial consideration is the wrap-around behavior when an index reaches the end of the buffer.

Whenever an index is incremented, a check is performed to determine if it has reached the maximum size of the buffer. If the index is equal to the maximum size, the next index is set to 0, effectively wrapping around to the beginning of the buffer. This wrap-around behavior allows the circular buffer to reuse previously allocated memory locations, ensuring efficient utilization of resources.

Another benefit of the circular buffer is that with the read index, write index and buffer size, the state of the buffer can be checked by doing at most three comparisons.

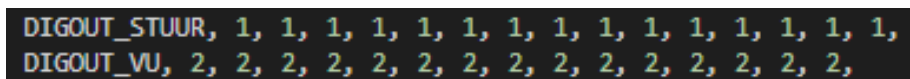
- If the is equal to the write index, it indicates that the buffer is empty, and there is no data available for reading.
- When the read index is less than the write index, it signifies that the data in the buffer is not wrapped around. In this case, the difference between the write index and read index represents the remaining amount of data to be read.
- If the read index is greater than the write index, it suggests that the data wraps around the end of the buffer. To calculate the remaining data, the total buffer size is subtracted by the difference between the read index and write index.

These calculations provide an efficient and straightforward way to monitor the circular buffer and determine if data is available for reading. This approach offers a convenient means of tracking the buffer's status and effectively managing data availability within the circular buffer.

## 5.2 Mapping datapoints to SHC types

To ensure the accurate execution of KNX commands in the SHC, it is necessary to verify the type of each command based on its index or datapoint. The `datapointToSHCMap` array serves as a mapping mechanism, associating KNX datapoints with their corresponding SHC command types for verification purposes.

The `datapointToSHCMap` array, consisting of 1024 elements of type `uint8_t`, is initialized with specific values that establish the mapping between KNX datapoints and SHC command types. Each pair of values within the array represents a unique mapping, where the first value denotes the KNX datapoint index, and the subsequent values represent the associated SHC command types.



```
DIGOUT_STUUR, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
DIGOUT_VU, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
```

Figure 5.1: Screenshot of part of the `datapointToSHCMap` array.

Figure 5.1 shows a part of the array, where the KNX datapoint `DIGOUT_STUUR` is mapped to the SHC command type 1, with subsequent repetitions indicating that all indexes of this datapoint correspond to the same SHC command type. Similarly, the KNX datapoint `DIGOUT_VU` is associated with the SHC command type 2. This pattern continues throughout the array, establishing mappings for other KNX datapoints and their respective SHC command types.

During the verification process, when a command is received or processed, the corresponding KNX datapoint index is used as an index into the `datapointToSHCMap` array. The associated SHC command type can then be retrieved, enabling the system to validate and execute the command appropriately based on its intended type.

While the approach of using a static array provides a fast and efficient means of verifying command types based on their index, it may lead to significant memory usage. However, an alternative approach can be used to reduce the memory requirements by utilizing a calculating function. Instead of storing the entire mapping in memory, this function can dynamically determine the SHC command type based on the KNX datapoint index. By calculating the type in runtime, the memory usage can be reduced to only two values per SHC command type: the amount of

datapoints that correspond to the command and the SHC command type. This calculating function eliminates the need for a static array or dynamic memory allocation, resulting in improved memory efficiency. However, it's important to consider that this approach may introduce some computational overhead due to the additional calculations required during runtime.

In the context of mapping data points, the identification of incoming information is essential. However, for outgoing information, it becomes necessary to determine the corresponding data point based on the type. Nonetheless, this aspect is of lesser importance due to the existence of update functions specific to each type. These update functions allow the definition of the starting data point and the number of data points associated with a particular type.

To streamline the editing process of these values, a header file named "KNXAddressTypes.h" was created. This header file contains definitions for all data point types and the corresponding number of data points. Additionally, a third constant value, derived from the previously mentioned values, calculates the starting data point for each type. This approach enables convenient modification of the data point indexes during development and beyond.

However, it should be acknowledged that the data point indexes have been embedded in the product database entry. Therefore, any changes made to the data point indexes also necessitate editing the corresponding entries in the product database to ensure proper functionality.

## 5.3 KNX temperature, dimming and blinds

KNX supports a wide range of datapoint types, available at [10]. Among the datapoints the SHC uses, three required significantly more effort to implement compared to the others. These three types are temperature, dimming commands, and blinds commands [10].

### 5.3.1 Temperature

Temperature in KNX is commonly transmitted as a two-octet float value. The two octet float uses a four bit exponent E and a 12 bit two's complement multiplier M. Formula 5.1 can be used to decode the temperature value [10].

$$\text{FloatValue} = (0.01 \cdot M) \cdot 2^E \tag{5.1}$$

The bit allocation of the two octet float can be described as follows: the initial part consists of the sign bit of M, followed by four E bits, and concludes with the remaining 11 bits of M. Figure 5.2 visualizes this bit assignment [10].

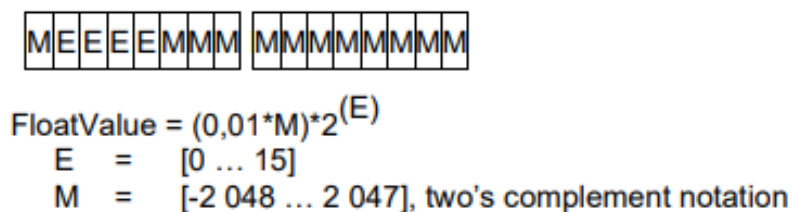


Figure 5.2: KNX 2-Octet Float Value bit assignment and details [10]



To allow the transmission of temperature values from the SHC to a KNX system, both decoding and encoding functions were developed. These functions utilize bit shift operations as their main mechanism. The decoding function serves the purpose of converting a KNX temperature value into a readable format, while the encoding function enables the transmission of SHC temperature values to the KNX system. To ensure the accuracy and reliability of these functions, they were tested in a separate C++ project. The testing process involved consecutive application of both functions and verification of the input and output.

### 5.3.2 Dimming

The dimming datapoint utilizes a four-bit value to control the dimming state of a light. The first bit, *c*, determines the direction of the dimming process, where 0 represents a decrease and 1 represents an increase. The next three bits indicate the StepCode, which determines the speed of dimming. A StepCode of zero indicates a break, causing the dimming action to stop. Any other value can be used to specify the desired dimming speed by indicating the number of intervals over which the dimming should occur [10].

The actual dimming speed, considering a time interval *t* between controlling the light, can be calculated as *t* multiplied by 2 raised to the power of (StepCode - 1). It is important to note that there is no standardized value provided for the delay parameter *t*. Figure 5.3 provides a visual representation of this bit assignment [10].

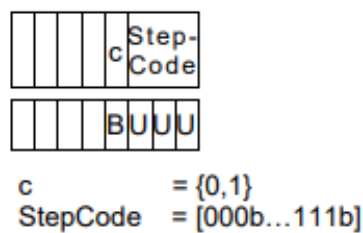


Figure 5.3: KNX four bit assignment and details of dimming type. [10]

During the integration of dimming functionality into the SHC, the dimming duration parameter was excluded due to its infrequent usage. Instead, standard values were established: a one-second transition for both turning on and off, and a ten-second transition while dimming. These values were chosen to ensure smooth and gradual transitions when controlling the light.

### 5.3.3 Blinds

The implementation of blinds in the KNX system differs from their conventional application in the SHC. In the SHC, separate up and down buttons are used for controlling the blinds, whereas in KNX, a single button is used. When this button is held, it toggles between upward and downward movement, and pressing it halts the motion. In the KNX system, pressing the button to stop the blind's movement triggers a slight adjustment in the opposite direction, allowing for precise control over the angle of blinds like venetian blinds.

It is worth noting that KNX provides a 4-bit blind control datapoint, which is similar to the dimming datapoint mentioned in section 5.3.2. However, testing some switches from Electron, Weinzierl and Basalte indicated that most manufacturers prefer using two separate bit objects

for blind control. Since the SHC does not feature single-button blind control, and the objective of this study was to implement all SHC functions in KNX, the blind control mechanism in KNX was implemented to replicate the operation observed in the SHC.

## 5.4 Timing, threads and interrupts

Timing was important for this study, as minimizing the impact on the existing system was a key objective. In their current system, Bits & Bytes utilized the *putc* and *getc* functions for serial read and write operations. Consequently, the initial test was designed using these functions, as they were already in use, along with processing and executing functions. In order to avoid placing any strain on the existing system, software threads were used. This necessitated investigating the threading of read, write, processing, and executing operations. However, the initial test exhibited delays of up to 200 ms, which were quite noticeable and unintended.

The problem with this approach came from the fact that the processor used in their system only had a single hardware core. As a result, the threading did not enable simultaneous execution of the operations, but instead slowed down the entire system. The reason behind this delay was the improper functioning of the *wait\_us* function in the mbed thread, as it did not put a thread to sleep. Consequently, as Mbed OS uses a round robin scheduler, a thread waiting with the *wait\_us* function effectively halved the system's performance. With separate threads for receiving and sending, this slowdown was further increased, resulting in a threefold decrease in performance [14].

To rectify this issue, the deprecated wait function was replaced with *ThisThread::sleep\_for*, resolving the problem of the thread not entering the sleep state. Another problem that emerged was the blocking nature of the data transfer during the send operation, which caused the thread to be blocked for several milliseconds, further decreasing system performance by half. [14].

Considering these problems, the utilization of threads was dropped in the integration process. Upon realizing that the read and write delays were caused by the blocking nature of the *putc* and *getc* functions, efforts were made to explore non-blocking or asynchronous read and write functions. However, these functions were not supported for our processor in Mbed OS 5. Although Mbed OS 6 offered standard non-blocking read and write functions, it was ultimately discarded as elaborated in Chapter 4 [14, 15].

The delay caused by a blocking write function can be calculated to show its significance. A message from knx to the SHC to turn on all 16 relays of the SHC is 93 bytes long when using the services explained in chapter 3. To calculate the time it would take to send an *All relay off* command over the 19600 baud connection, formula 2.1 can be used. Since one byte equals eight bits, and every byte is accompanied by a stop byte and a parity byte, it takes 10 bits to send a single byte of data. Therefore, the time it would take to send 93 bytes over a 19600 baud connection would be:

$$\text{Time} = \frac{93 \cdot 10 \text{ bits}}{19600 \text{ bits/s}} = 0.04745 \text{ s or } 47.5 \text{ ms}$$

It would take approximately 47.5 milliseconds to send 93 bytes over a 19600 baud connection with a blocking write function. This is five times longer than the current time it takes the SHC to react to an input when not under load.

The proposed solution involved utilizing interrupts, one triggered upon data arrival and another upon completion of the sending operation. This significantly accelerated system performance to the extent that the integration of KNX no longer had any measurable impact on the system, as measured in milliseconds.

## 5.5 Flow of the data

Figure 5.4 shows a flowchart of how incoming and outgoing data is read, created and parsed.

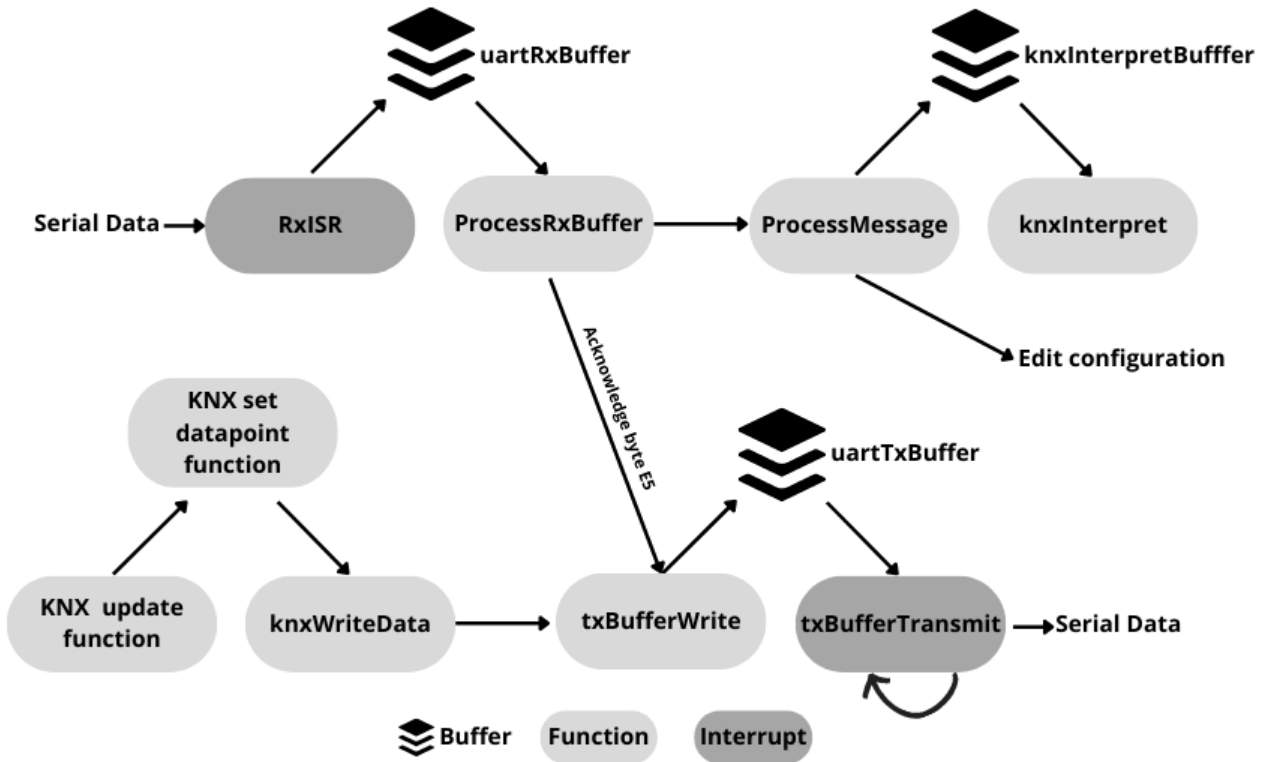


Figure 5.4: Dataflow of the incoming and outgoing KNX serial data

### 5.5.1 Outgoing data

Outgoing data is created by KNX update functions. One function exists for each type of IO. Each time an update function is called, it compares the current value of the IO to the previously stored value. If they do not match, the stored value is updated and the value, KNX datapoint id and type of the data are passed to the *knxSetDatapoints* function matching the value's length. This function parses the data into the Weinzierl *SetDataPointValue* request and calls the *knxWriteData* function, which parses the data into the FT1.2 format, writes it to the *uartTxBuffer* and starts the *txBufferTransmit*. The flow of data between these functions is visualized in figure 5.4.

### 5.5.2 Incoming data

The *rxIsr* puts the incoming data into the *uartRxBuffer*. Next, the *processRxBuffer* reads the *uartRxBuffer* one byte at a time, detects valid FT1.2 frames. When a valid frame is detected, the function uses *uartTxBuffer* and *txBufferTransmit* to transmit an acknowledge byte and passes the frames data to *processMessage*. The *processMessage* then handles the data. If the data is

a datapoint, it is parsed into the correct SHC command and put into the *knxInterpretBuffer*, which is executed by the SHC when *knxInterpret* is called. When the data is not a datapoint, it is related to editing the SHC's configuration. This is further explained in Section 5.5.3. The flow of data between these functions is visualized in figure 5.4.

### 5.5.3 Configuration settings from KNX

The operation of the Weinzierl module, which is powered by the KNX bus, can continue even when the SHC is still starting or turned off. As a result, ETS may provide a successful response to programming, leading to a misconception that the data has been sent to the SHC when it hasn't. To address this issue, a decision was made to retrieve the configuration data from the Weinzierl module every time the SHC starts up.

The configuration settings of the SHC are stored in its EEPROM memory. Proper management of incoming configuration data is crucial to avoid misconfigured EEPROM memory, which can cause undesirable behavior in the SHC. Determining when to read the data involves two steps in the SHC configuration: reinitializing variables that are typically loaded on startup and writing the configuration data to the EEPROM memory for the subsequent startup.

If the module has not been configured in ETS, the parameter bytes associated with the configuration will not have been written, and the contents of the memory will be unknown. If the SHC requests these parameters, the data retrieved may cause unwanted behavior. To solve this problem, a startup sequence of events was implemented using the *getServerItem* request to obtain the ETS application manufacturer data. In case the device has been previously programmed, this data will correspond to the Bits & Bytes manufacturer code. If an incorrect code is returned, the KNX functions will be disabled, and the parameter bytes of the Weinzierl module will not be requested.

This approach, however, has the drawback that when programming settings using the SHC configuration tool, any settings already present in ETS will always overwrite the ones programmed by the tool. Nevertheless, this was not considered a problem, as someone using ETS should not need the configuration tool in the first place.

Furthermore, a mechanism had to be developed to detect when the Weinzierl module was programmed by ETS, so that the parameter bytes could be retrieved again. When the device is programmed, it undergoes a reset, briefly interrupting its communication on the KNX bus. This reset state is recognized as a server item, triggering a *ServerItem* indication. When the SHC receives the indication with subservice code C2, it will send a *getParameterBytes* request to obtain the new configuration data.

## 5.6 Data stored in the SHC

The SHC internally stores all its states and configuration either in its memory or on EEPROM memory. However, in order to maintain compatibility with older versions of the module, booleans are utilized to store the majority of states. These booleans are often stored in into bytes, where each byte index represents a boolean value. To determine the corresponding function for each boolean index, documentation had to be referenced. The process of utilizing these states and

transforming them between internally addressed functions, based on individual bits, and byte-addressed data from the Weinzierl module proved to be time-consuming during the study.

## 5.7 Code documentation

The code comments were created in a style supported by doxygen. This allowed the doxygen software to be used to generate a HTML pages and pdf version of the code documentation, making it easy accessible. A custom CSS theme was used to modernize the doxygen HTML pages. The PDF version of the documentation can be found in appendix A [18, 19].

# Chapter 6

## Product database entry

This section focuses on the development of a product database entry that serves two essential purposes. Firstly, it allows importing the SHC as a KNX device, which allows its IO to be coupled to other KNX devices. Secondly, it enables the configuration of the SHC through ETS, providing convenient access to various configuration settings.

### 6.1 Creating a product database entry

To create a product database entry, also known as a .prod file, the KNX manufacturer tool is essential. The development process for a product database entry can be initiated in two ways: starting a new project from scratch or importing an existing .prod file from any other KNX device. In this case, the objective was to create a new product database entry for the KNX BAOS 832. To start the development, the existing .prod file for the BAOS 832 was imported and used as a foundation for the SHC's product database entry.

A product database entry comprises three main components: a catalog file, a hardware file, and an application program file. The catalog file allows the definition of properties and product categories, influencing how the SHC is displayed in the online ETS catalog once it receives certification. The hardware file contains information specific to the hardware associated with the product. The application program file defines the ETS UI, ETS datapoints, and memory allocation for the device it is developed for.

### 6.2 Parts of the application program

Before utilizing the application program, it is necessary to import a binary file called SREC, which contains information regarding the register size and addresses of the hardware's memory. This crucial step enables the assignment of parameters to the memory within the application program. More info about the SREC file format is available at [20].

The application program is divided into two main sections: the static part and the dynamic part. In the static part, there are two types of parameters commonly used: virtual parameters and memory parameters. Virtual parameters are not stored in the device memory and are typically used for temporary calculations or intermediate values. On the other hand, memory parameters

are saved in the device memory and can be retrieved from the module, allowing for storage of the configuration settings.

The static part also encompasses other vital components such as com objects and data types. Com objects refer to the group objects displayed in ETS, while data types can be customized to represent various information formats, such as RGB color, a list of enums (known as a restriction), or a limited number field. These data types provide a structured and defined approach to handling different types of data, and it is possible to create multiple data types with different restrictions.

For example, a number field can be created with a specific restriction, such as allowing only even numbers between 0 and 10. This restriction ensures that only valid values within the defined range and criteria can be entered for that particular parameter. By defining different data types with various restrictions, the application program can enforce specific rules and constraints for the parameters, enhancing data integrity and usability.

Parameters and com objects are referenced through parameter references and com references respectively. A reference can override most properties of the parameters and com objects, except for the data type of the parameter object. Multiple references can exist per com object or parameter, enabling the modification of text or properties based on configuration settings.

The dynamic part of the application program contains the elements that serve as the UI, enabling interaction with the program and allowing parameter configuration. These elements are known as parameter reference references (refrefs). To include a parameter or com object in the dynamic part, their references (com reference or parameter ref) must be referred to using com refref or parameter refref, respectively. This can be achieved by utilizing a choose block, which functions similarly to an if statement for parameter values. The choose block allows for the conditional display or hiding of parameter refs and com object refs based on the values of other parameters. By leveraging the choose block, the dynamic part can dynamically adapt its appearance and behavior based on the user's inputs and selected parameter values.

### **6.3 Maximum amount of parameters**

The initial product database entry of the BAOS 832 module by Weinzierl included a default allocation of 250 configurable bytes. While the current SHC product database entry did not require more than 250 bytes, it is worth noting that according to [21], it is possible to use the remaining memory of the module for extra parameter bytes. By using this approach, the total number of parameter bytes can be expanded to 76885.

### **6.4 Testing in ETS**

A product database entry can only be imported into ETS once the certification process has been started. To test it during development, an example ETS project can be created from within the Manufacturer tool. It can be imported into ETS and contains the device under the "devices" panel. The device can then be copied to other ETS projects. Figure 6.1 shows how to finished UI of the SHC looks in ETS.

Initially, programming the device in ETS was not possible due to an error message indicating a mismatch between the device manufacturer and the application program manufacturer. In re-

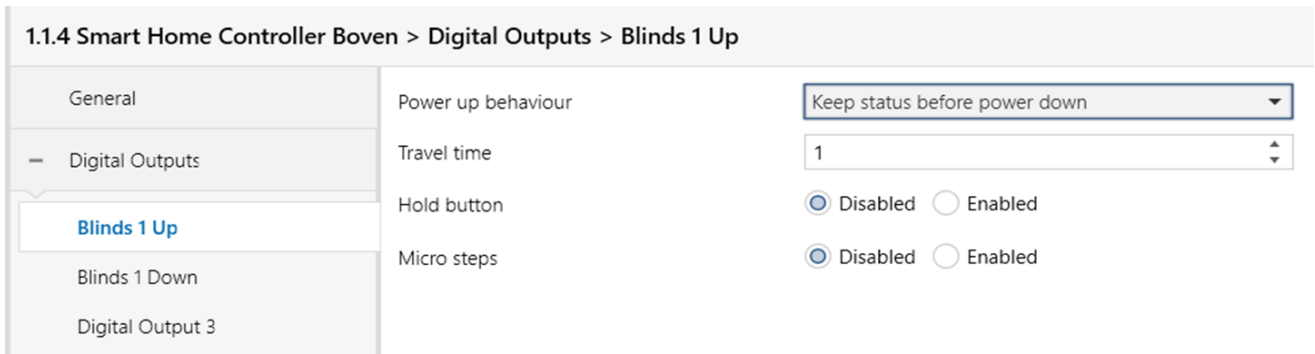


Figure 6.1: Screenshot of the SHC configuration panel in ETS

sponse to this issue, Bits & Bytes reached out to Weinzierl for assistance. Weinzierl provided tools that allowed the reprogramming of the manufacturer within their BAOS 832 module, resolving the compatibility problem.





# Chapter 7

## Conclusion

This study has conducted comprehensive research on various KNX development possibilities, specifically focusing on the creation of a serial interface from scratch. After careful evaluation, it was determined that utilizing an existing OEM serial interface while developing a new product database entry would be the optimal approach for Bits & Bytes.

Out of the qualifying options, namely the SIM-KNX250 by Tapko and the KNX BAOS Module 832, the KNX BAOS Module 832 was chosen for its compatibility and suitability for the application. The communication protocols of this module were successfully implemented to facilitate the integration process.

The expanded SHC code was written based on interrupts, as threading proved insufficient for achieving non-blocking IO on a single-core system. The analysis revealed that the overhead of the new code was minimal, taking less than a millisecond to execute.

Although upgrading to MBED OS 6 was not feasible due to the lack of native support for the LPC 4088, it was discovered that custom support could be achieved by redefining drivers. This finding opens up possibilities for future customization and optimization of the system.

The acquisition of the KNX Manufacturer Tool proved instrumental in developing the new product database entry, enabling seamless configuration and integration within the KNX ecosystem.

Combining all these elements, thorough testing and verification were conducted, resulting in the successful and validated functioning of the KNX integration within the SHC.

For future studies, it is recommended to explore custom board support for the LPC 4088 in MBED OS, which would enable non-blocking IO support. This further improvement would enhance the system's performance and capabilities, contributing to a more efficient and seamless home automation experience.

In conclusion, this study has demonstrated the possibility and effectiveness of integrating KNX into the SHC through the development of a serial interface, code implementation, and selection of compatible modules. The findings and insights obtained from this research lay the foundation for enhancing the capabilities of home automation systems, with the potential for further customization and optimization in future studies.



# Bibliography

- [1] *B&B SMART HOME CONTROLLER PRODUCT HANDBOOK*, Bits & Bytes nv, February 2022, (Last accessed on 11 June 2023). [Online]. Available: <https://bitsbytes.be/en/products/smart-home-controller>
- [2] *Transceiver for KNX Twisted Pair Networks NCN5130*, Onsemi, Jun 2022, (Last accessed on 11 June 2023). [Online]. Available: <https://www.onsemi.com/products/interfaces/wired-transceivers-modems/ncn5130>
- [3] *KNX EIB TP-UART 2+ IC*, Siemens, Aug 2013, (Last accessed on 11 June 2023). [Online]. Available: [https://www.opternus.com/fileadmin/\\_migrated/content\\_uploads/TPUART2\\_Datenblatt\\_20130806\\_01.pdf](https://www.opternus.com/fileadmin/_migrated/content_uploads/TPUART2_Datenblatt_20130806_01.pdf)
- [4] *KNX EIB Transceiver Family E981.03 / E981.23 / E981.33*, Elmos Semiconductor AG, Jun 2015, (Last accessed on 11 June 2023). [Online]. Available: [https://www.elmos.com/fileadmin/elmos-website/products/interface/knx\\_eib\\_transceiver/elmos-knx-transceiver\\_with\\_hardware\\_current-programming-e98123-is.pdf](https://www.elmos.com/fileadmin/elmos-website/products/interface/knx_eib_transceiver/elmos-knx-transceiver_with_hardware_current-programming-e98123-is.pdf)
- [5] *KNX BAOS Binary Protocol*, WEINZIERL ENGINEERING GmbH, April 2021, (Last accessed on 11 June 2023). [Online]. Available: [https://weinzierl.de/images/download/development/830/knxbaos\\_protocol\\_v2.pdf](https://weinzierl.de/images/download/development/830/knxbaos_protocol_v2.pdf)
- [6] Bits & Bytes, “Smart home controller,” (Last accessed on 11 June 2023). [Online]. Available: <https://bitsbytes.be/en/products/smart-home-controller>
- [7] *KNX BASICS*, KNX Association, (Last accessed on 11 June 2023). [Online]. Available: [https://www.knx.org/wAssets/docs/downloads/Marketing/Flyers/KNX-Basics/KNX-Basics\\_en.pdf](https://www.knx.org/wAssets/docs/downloads/Marketing/Flyers/KNX-Basics/KNX-Basics_en.pdf)
- [8] C. Parthoens, “Knx system – knx association,” 2017, (Last accessed on 11 June 2023). [Online]. Available: <https://support.knx.org/hc/en-us/categories/115000252249-KNX-System>
- [9] *Data sheet KNX BAOS Module 832 Serial Interface and ObjectServer for KNX Bus*, WEINZIERL ENGINEERING GmbH, May 2022, (Last accessed on 11 June 2023). [Online]. Available: [https://weinzierl.de/images/download/development/832/weinzierl\\_832\\_knx\\_baos\\_module\\_datasheet\\_en.pdf](https://weinzierl.de/images/download/development/832/weinzierl_832_knx_baos_module_datasheet_en.pdf)
- [10] *Interworking Datapoint Types*, KNX Association, May 2021, (Last accessed on 30 May 2023). [Online]. Available: [https://www.knx.org/wAssets/docs/downloads/Certification/Interworking-Datapoint-types/03\\_07\\_02-Datapoint-Types-v02.02.01-AS.pdf](https://www.knx.org/wAssets/docs/downloads/Certification/Interworking-Datapoint-types/03_07_02-Datapoint-Types-v02.02.01-AS.pdf)

- [11] “Certified system components,” KNX Association, (Last accessed on 15 April 2023). [Online]. Available: <https://www.knx.org/knx-en/for-manufacturers/development/system-components/>
- [12] P. Hauner, *SIM-KNX Serial Interface for KNX Technical & Application Description*, TAPKO technologies GmbH, May 2019, (Last accessed on 11 June 2023). [Online]. Available: [https://www.tapko.de/fileadmin/user\\_upload/Downloads/SIM-KNX/TAPKO\\_TAD\\_EN\\_SIM-KNX\\_R1-4.pdf](https://www.tapko.de/fileadmin/user_upload/Downloads/SIM-KNX/TAPKO_TAD_EN_SIM-KNX_R1-4.pdf)
- [13] *KNX BAOS Module 830 Serial Interface and ObjectServer for KNX Bus*, WEINZIERL ENGINEERING GmbH, May 2022, (Last accessed on 11 June 2023). [Online]. Available: <https://weinzierl.de/images/download/development/830/weinzierl-830-knx-baos-module-5171-datasheet-en.pdf>
- [14] “Introduction - mbed os 5,” Arm Ltd, (Last accessed on 27 May 2023). [Online]. Available: <https://os.mbed.com/docs/mbed-os/v5.15>
- [15] “Introduction - mbed os 6,” Arm Ltd, (Last accessed on 27 May 2023). [Online]. Available: <https://os.mbed.com/docs/mbed-os/v6.16>
- [16] M. Salazar, “Custom and community board support for mbed os 6,” May 2020, (Last accessed on 27 May 2023). [Online]. Available: <https://os.mbed.com/blog/entry/Custom-and-community-board-support>
- [17] “Mid-range Microcontrollers (MCUs) based on Arm® Cortex®-M4 Cores,” NXP Semiconductors, May 2020, (Last accessed on 27 May 2023). [Online]. Available: [https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc4000-arm-cortex-m4:MC\\_1403790399405#/](https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc4000-arm-cortex-m4:MC_1403790399405#/)
- [18] jothepro, “doxygen-awesome-css 2.2.1,” <https://github.com/jothepro/doxygen-awesome-css>, 2023, (Last accessed on 11 June 2023).
- [19] D. van Heesch., “doxygen 1.9.7,” <https://github.com/doxygen/doxygen>, 2023, (Last accessed on 11 June 2023).
- [20] S. Bergmans, “Motorola sxx records format,” December 2021, (Last accessed on 3 June 2023). [Online]. Available: <https://www.sbprojects.net/knowledge/fileformats/motorola.php>
- [21] “Knx baos starter kit user’s guide,” WEINZIERL ENGINEERING GmbH, February 2023, (Last accessed on 4 June 2023). [Online]. Available: [https://weinzierl.de/images/download/documents/baos/weinzierl\\_knx\\_baos\\_users\\_guide.pdf](https://weinzierl.de/images/download/documents/baos/weinzierl_knx_baos_users_guide.pdf)

# Appendix A

## Doxygen

# SHC KNX Documentation

2.1

By Siebe Nijs

Generated by Doxygen 1.9.6





<b>1 KNX SHC Library</b>	<b>1</b>
1.1 Integrating the function in the main program	1
1.2 Function flowchart	1
1.3 Adding a datapoint	1
<b>2 Module Index</b>	<b>3</b>
2.1 Modules	3
<b>3 File Index</b>	<b>5</b>
3.1 File List	5
<b>4 Module Documentation</b>	<b>7</b>
4.1 Buffers	7
4.1.1 Detailed Description	8
4.1.2 Function Documentation	8
4.1.2.1 rxBufferHasData()	8
4.1.2.2 rxISR()	8
4.1.2.3 txBufferHasData()	8
4.1.2.4 txBufferTransmit()	8
4.1.2.5 txBufferWrite()	8
4.2 GetShcData	9
4.2.1 Detailed Description	9
4.2.2 Function Documentation	9
4.2.2.1 floatToKnxTwoBytes()	9
4.2.2.2 knxConvertStatusIntUInt8Array()	10
4.2.2.3 knxConvertSwappedStatusIntUInt8Array()	10
4.2.2.4 knxUpdateAll()	11
4.2.2.5 knxUpdateAnalogIn()	11
4.2.2.6 knxUpdateDigitalIn()	11
4.2.2.7 knxUpdateDigitalOut()	11
4.2.2.8 knxUpdateDimmerPushButtons()	11
4.2.2.9 knxUpdateDimmers()	12
4.2.2.10 knxUpdateIndicators()	12
4.2.2.11 knxUpdateTemps()	12
4.2.2.12 log2OfUInt16()	12
4.2.3 Variable Documentation	12
4.2.3.1 analogInValues	13
4.2.3.2 dimmerVoor	13
4.2.3.3 ingangVoor	13
4.2.3.4 OUT1	13
4.2.3.5 OUT2	13
4.2.3.6 statusVoor	13
4.2.3.7 tempsMeasured	13

---

4.3 MessageProcessing	13
4.3.1 Detailed Description	14
4.3.2 Function Documentation	14
4.3.2.1 checkSumControl()	14
4.3.2.2 knxSetOneByteDatapoints()	15
4.3.2.3 knxSetTwoByteDatapoints()	15
4.3.2.4 knxWriteData()	15
4.3.2.5 processMessage()	16
4.3.2.6 processRxBuffer()	16
4.3.3 Variable Documentation	16
4.3.3.1 datapointToSHCMap	17
4.3.3.2 frameDetected	17
4.3.3.3 frameLength	17
4.3.3.4 frameReadIndex	17
4.3.3.5 knx_cr	17
4.3.3.6 knxFrameData	17
4.4 ExternalUse	18
4.4.1 Detailed Description	18
4.4.2 Function Documentation	18
4.4.2.1 knxComfortBufferHasData()	18
4.4.2.2 knxInit()	18
4.4.2.3 knxInterpret()	19
4.4.2.4 knxInterpretBufferHasData()	19
4.4.2.5 knxProcessMultiple()	19
4.4.2.6 knxReadComfortBuffer()	19
4.4.2.7 knxReset()	20
<b>5 File Documentation</b>	<b>21</b>
5.1 C:/KNX_MBedOs/SHC_V2_Imported/KNX/knx.cpp File Reference	21
5.1.1 Detailed Description	23
5.1.2 Macro Definition Documentation	23
5.1.2.1 HIBYTE	23
5.1.2.2 LOBYTE	23
5.1.3 Function Documentation	23
5.1.3.1 rxBufferReadByte()	24
5.1.4 Variable Documentation	24
5.1.4.1 knxDigOutStatus	24
5.2 knx.h	24
5.3 KNXAddressTypes.h	24
<b>Index</b>	<b>27</b>

# Chapter 1

## KNX SHC Library

### 1.1 Integrating the function in the main program

Create the KNX Serial at 19600 Baud as a global, under the name `knx_uart` as this library will expect it to be defined externally. Put `knxInit()` anywhere in the startup where it will only be executed once.

Put the following functions in a run loop where they will be executed in the following order: `knxUpdateFunctions` -> `processRxBuffer()` or `knxProcessMultiple()` -> `knxInterpret()`. Delay between the functions does not matter.

The `knxComfortBuffer` is read externally and `knxReadComfortBuffer()` should be implemented in `interpret.cpp`

### 1.2 Function flowchart

Receive: `rxISR()` -> `uartRxBuffer` -> `processRxBuffer()` -> `processMessage()` -> `knxInterpretBuffer` -> `knxInterpret()`

Some commands are not executed through interpret. For these details check the `processMessage()` documentation.

Transmit: (any) `knxUpdateFunction` -> (any) `knxSetDatapoints` -> `knxWriteData()` -> `txBufferWrite()` -> `uartTxBuffer` -> `txBufferTransmit()`

Multiple `knxUpdateFunctions` and `knxSetDatapoint` functions exist for different data types.

The `rxISR()` and `txBufferTransmit()` functions are interrupts.

### 1.3 Adding a datapoint

Add the datapoint `START`, `NAME` and `AMOUNT` to `KNXAddressTypes.h`

Next, in `knx.cpp`: add datapoint to `datapointToSHCMap[]`

If the datapoints gets a value from knx:

Add a case for the received value in `processMessage()`

If the datapoints writes data from the SHC to knx:

Create an update function for the value that formats the data to a `uint8` array. The other update functions can serve as examples. Next, use the appropriate `knxSetDatapoint`(based on value byte size) to transmit the data.



# Chapter 2

## Module Index

### 2.1 Modules

Here is a list of all modules:

Buffers . . . . .	7
GetShcData . . . . .	9
MessageProcessing . . . . .	13
ExternalUse . . . . .	18



## Chapter 3

# File Index

### 3.1 File List

Here is a list of all documented files with brief descriptions:

<a href="#">C:/KNX_MBedOs/SHC_V2_Imported/KNX/knx.cpp</a> . . . . .	21
<a href="#">C:/KNX_MBedOs/SHC_V2_Imported/KNX/knx.h</a> . . . . .	24
<a href="#">C:/KNX_MBedOs/SHC_V2_Imported/KNX/KNXAddressTypes.h</a> . . . . .	24





# Chapter 4

## Module Documentation

### 4.1 Buffers

All buffers are circular buffers. They each have a read and write index. The read index indicates the next data that will be read. The write index indicates the next index that will be written to. When the indexes are modified, a software overflow should be included when they reach the max buffer length.

#### Macros

- `#define UART_RX_BUFFER_MAX_LENGTH 1000`
- `#define UART_TX_BUFFER_MAX_LENGTH 100`
- `#define KNX_INTERPRET_BUFFER_MAX_LENGTH 40`
- `#define KNX_COMFORT_BUFFER_LENGTH 50`

#### Functions

- void `rxISR ()`
- void `txBufferTransmit ()`
- bool `rxBufferHasData ()`
- void `txBufferWrite (uint8_t *data, uint16_t datalength)`
- bool `txBufferHasData ()`

#### Variables

- `uint16_t uartRxBufferNextReadIndex = 0`
- `uint16_t uartRxBufferNextWriteIndex = 0`
- `uint8_t uartRxBuffer [UART_RX_BUFFER_MAX_LENGTH]`
- `uint16_t uartTxBufferNextReadIndex = 0`
- `uint16_t uartTxBufferNextWriteIndex = 0`
- `uint8_t uartTxBuffer [UART_TX_BUFFER_MAX_LENGTH]`
- `uint16_t knxInterpretBufferNextReadIndex = 0`
- `uint16_t knxInterpretBufferNextWriteIndex = 0`
- `char knxInterpretBuffer [KNX_INTERPRET_BUFFER_MAX_LENGTH][16]`
- `uint16_t knxComfortBufferNextReadIndex = 0`
- `uint16_t knxComfortBufferNextWriteIndex = 0`
- `uint16_t knxComfortBuffer [KNX_COMFORT_BUFFER_LENGTH]`

### 4.1.1 Detailed Description

All buffers are circular buffers. They each have a read and write index. The read index indicates the next data that will be read. The write index indicates the next index that will be written to. When the indexes are modified, a software overflow should be included when they reach the max buffer length.

### 4.1.2 Function Documentation

#### 4.1.2.1 rxBufferHasData()

```
bool rxBufferHasData ( )
```

Checks if data is available in the UART Rx buffer by comparing its read and write index.

##### Returns

True if unread data is present in array

#### 4.1.2.2 rxISR()

```
void rxISR ( )
```

Interrupt that reads the next serial char and puts it in the uartRxBuffer

#### 4.1.2.3 txBufferHasData()

```
bool txBufferHasData ( )
```

Check if all data in the UART Tx buffer has been sent

##### Returns

true if data is available

#### 4.1.2.4 txBufferTransmit()

```
void txBufferTransmit ( )
```

UART Transmit function. Transmits the first byte of the uart buffer. An interrupt will be generated on transmit, which will trigger sending the next byte. Workaround for non-blocking io functions not being supported for mbed OS 5

#### 4.1.2.5 txBufferWrite()

```
void txBufferWrite (
    uint8_t * data,
    uint16_t datalength )
```

Writes data to the TxBuffer

## Parameters

<i>data</i>	the data to be written
<i>datalength</i>	the size of the data argument

## 4.2 GetShcData

Functions used to get status data from the SHC and formatting it to usable data. Update functions check for a change in current value compared to the stored values. KnxSetDataPoint functions send the changed values to the knx Buffer. Other functions are for formatting SHC variables into Byte Arrays.

### Functions

- `uint8_t * knxConvertSwappedStatusIntUint8Array` (int intvalue, uint8\_t \*array)
- `uint8_t * knxConvertStatusIntUint8Array` (int intvalue, uint8\_t \*array, uint8\_t arraysize)
- `uint16_t log2OfUint16` (uint16\_t value)
- `uint16_t floatToKnxTwoBytes` (float value)
- void `knxUpdateDigitalOut` ()
- void `knxUpdateDigitalIn` ()
- void `knxUpdateAnalogIn` (uint8\_t analogIn)
- void `knxUpdateTemps` (uint8\_t tempsensor)
- void `knxUpdateDimmers` ()
- void `knxUpdateIndicators` ()
- void `knxUpdateDimmerPushButtons` ()
- void `knxUpdateAll` ()

### Variables

- int \* `statusVoor`
- int \* `ingangVoor`
- int \* `tempsMeasured`
- int \* `analogInValues`
- int \* `dimmerVoor`
- DigitalOut `OUT1`
- DigitalOut `OUT2`

### 4.2.1 Detailed Description

Functions used to get status data from the SHC and formatting it to usable data. Update functions check for a change in current value compared to the stored values. KnxSetDataPoint functions send the changed values to the knx Buffer. Other functions are for formatting SHC variables into Byte Arrays.

### 4.2.2 Function Documentation

#### 4.2.2.1 floatToKnxTwoBytes()

```
uint16_t floatToKnxTwoBytes (
    float value )
```

Converts a standard float to the knx 2 byte float. KNX 2 byte float: Multiplier M in 2s complement, Exponent E. Example: MEEEEMMMMMMMMMM (4 bit E, 11+1 sign bit M). KNX value =  $M \cdot 2^E$ . Used to export temperature to KNX.

## Parameters

<i>value</i>	Float to convert
--------------	------------------

## Returns

KNX 2 byte Float

#### 4.2.2.2 knxConvertStatusIntUint8Array()

```
uint8_t * knxConvertStatusIntUint8Array (
    int intvalue,
    uint8_t * array,
    uint8_t arraysize )
```

Converts a SHC status int to an array of uint8 values. Used for the SHC digital in status, push button in. Example of how bits are stored in the last 2 bytes: (15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0). Numbers are the corresponding output.

## Parameters

<i>intvalue</i>	Status int value to convert
<i>array</i>	Array to write back to
<i>size</i>	Amount of statuses in the int value

## Returns

Array of uint8 corresponding with the value of each bit, where the index of the array == the number of the digital output

#### 4.2.2.3 knxConvertSwappedStatusIntUint8Array()

```
uint8_t * knxConvertSwappedStatusIntUint8Array (
    int intvalue,
    uint8_t * array )
```

Converts a nibble swapped SHC status int to an array of uint8 values. Used for the SHC digital out status. Example of how bits are stored in the last 2 bytes: (3,2,1,0,7,6,5,4,11,10,9,8,15,14,13,12). Numers are the corresponding digital output.

## Parameters

<i>intvalue</i>	Status int value to convert
<i>array</i>	Array to write back to

**Returns**

Array of uint8 corresponding with the value of each bit, where the index of the array == the number of the digital output

**4.2.2.4 knxUpdateAll()**

```
void knxUpdateAll ( )
```

Call all update functions

**4.2.2.5 knxUpdateAnalogIn()**

```
void knxUpdateAnalogIn (
    uint8_t analogIn )
```

Compares the current status with the stored status, and implements [knxSetOneByteDatapoints\(\)](#) when they differ.

**Parameters**

<i>analogIn</i>	SHC id of the analog in to update
-----------------	-----------------------------------

**4.2.2.6 knxUpdateDigitalIn()**

```
void knxUpdateDigitalIn ( )
```

Compares the current status with the stored status, and implements [knxSetOneByteDatapoints\(\)](#) when they differ.

**4.2.2.7 knxUpdateDigitalOut()**

```
void knxUpdateDigitalOut ( )
```

Compares the current status with the stored status, and implements [knxSetOneByteDatapoints\(\)](#) when they differ.

**4.2.2.8 knxUpdateDimmerPushButtons()**

```
void knxUpdateDimmerPushButtons ( )
```

Compares the current status with the stored status, and implements [knxSetOneByteDatapoints\(\)](#) when they differ.

#### 4.2.2.9 knxUpdateDimmers()

```
void knxUpdateDimmers ( )
```

Compares the current status with the stored status, and implements [knxSetOneByteDatapoints\(\)](#) when they differ.

#### 4.2.2.10 knxUpdateIndicators()

```
void knxUpdateIndicators ( )
```

Compares the current status with the stored status, and implements [knxSetOneByteDatapoints\(\)](#) when they differ.

#### 4.2.2.11 knxUpdateTemps()

```
void knxUpdateTemps (
    uint8_t tempsensor )
```

Compares the current status with the stored status, and implements [knxSetTwoByteDatapoints\(\)](#) when they differ.

##### Parameters

<i>tempsensor</i>	SHC id of the temperature sensor to update
-------------------	--

•

#### 4.2.2.12 log2OfUint16()

```
uint16_t log2OfUint16 (
    uint16_t value )
```

Will return the floored log2 of value for all values > 1, otherwise returns 0. Used for the [FloatTo16Bit\(\)](#) function

##### Parameters

<i>value</i>	Status int value to convert
--------------	-----------------------------

##### Returns

floored Log2 of value

### 4.2.3 Variable Documentation

#### 4.2.3.1 analogInValues

```
int* analogInValues
```

Pointer to SHC status, gets updated by SHC and assigned in [knxInit\(\)](#)

#### 4.2.3.2 dimmerVoor

```
int* dimmerVoor
```

Pointer to SHC status, gets updated by SHC and assigned in [knxInit\(\)](#)

#### 4.2.3.3 ingangVoor

```
int* ingangVoor
```

Pointer to SHC status, gets updated by SHC and assigned in [knxInit\(\)](#)

#### 4.2.3.4 OUT1

```
DigitalOut OUT1 [extern]
```

SHC status, gets updated by SHC. Can be read and write

#### 4.2.3.5 OUT2

```
DigitalOut OUT2 [extern]
```

SHC status, gets updated by SHC. Can be read and write

#### 4.2.3.6 statusVoor

```
int* statusVoor
```

Pointer to SHC status, gets updated by SHC and assigned in [knxInit\(\)](#)

#### 4.2.3.7 tempsMeasured

```
int* tempsMeasured
```

Pointer to SHC status, gets updated by SHC and assigned in [knxInit\(\)](#)

## 4.3 MessageProcessing

Contains all functions and variables used in the processing of the FT1.2 messages. These functions often makes use of the buffer functions to access and store data.

## Functions

- bool `checksumControl` (uint8\_t \*data, uint16\_t length)
- void `processMessage` (uint8\_t \*data)
- void `processRxBuffer` ()
- void `knxWriteData` (uint8\_t \*data, uint16\_t datalength)
- void `knxSetOneByteDatapoints` (uint16\_t datapointId[], uint8\_t shcType[], uint8\_t value[], uint8\_t datapointsAmount)
- void `knxSetTwoByteDatapoints` (uint16\_t datapointId[], uint8\_t shcType[], uint16\_t value[], uint8\_t datapointsAmount)

## Variables

- uint8\_t `knx_cr` = 0b01110011
- uint8\_t `datapointToSHCMap` [1023]
- uint8\_t \* `knxFrameData`
- uint16\_t `frameReadIndex` = 0
- uint16\_t `frameLength` = 0
- bool `frameDetected` = false

### 4.3.1 Detailed Description

Contains all functions and variables used in the processing of the FT1.2 messages. These functions often makes use of the buffer functions to access and store data.

end of GetShcData

### 4.3.2 Function Documentation

#### 4.3.2.1 checksumControl()

```
bool checksumControl (
    uint8_t * data,
    uint16_t length )
```

Checks the sum of all received bytes of the ft1.2 message to detect if is the message is valid.

#### Parameters

<i>data</i>	Array of all bytes of the message (end byte is optional)
<i>length</i>	Length of data array without the end byte

#### Returns

True if message is valid



#### 4.3.2.2 knxSetOneByteDatapoints()

```
void knxSetOneByteDatapoints (
    uint16_t datapointId[],
    uint8_t shcType[],
    uint8_t value[],
    uint8_t datapointsAmount )
```

Parses one byte of data into a FT1.2 Message and calls [knxWriteData\(\)](#) to send it to the Tx Buffer.

##### Parameters

<i>datapointId</i>	Array of SHC Datapoint ids. Always starts at 0. (ex digout 0-15)
<i>shcType</i>	Array of types of SHC data. Has to be one of the defined SHC data types.
<i>value</i>	Array of one byte values that contains the new KNX values.
<i>datapointsAmount</i>	Amount of datapoints in the arrays.

#### 4.3.2.3 knxSetTwoByteDatapoints()

```
void knxSetTwoByteDatapoints (
    uint16_t datapointId[],
    uint8_t shcType[],
    uint16_t value[],
    uint8_t datapointsAmount )
```

Parses two bytes of data into a FT1.2 Message and calls [knxWriteData\(\)](#) to send it to the Tx Buffer.

##### Parameters

<i>datapointId</i>	Array of SHC Datapoint ids. Always starts at 0. (ex temp 0-2)
<i>shcType</i>	Array of types of SHC data. Has to be one of the defined SHC data types.
<i>value</i>	Array of one byte values that contains the new KNX values.
<i>datapointsAmount</i>	Amount of datapoints in the arrays.

#### 4.3.2.4 knxWriteData()

```
void knxWriteData (
    uint8_t * data,
    uint16_t dataLength )
```

Write data to Tx Buffer in FT1.2 format

##### Parameters

<i>data</i>	the data to send to the KNX module
<i>dataLength</i>	the length of the data in bytes

#### 4.3.2.5 processMessage()

```
void processMessage (
    uint8_t * data )
```

Executes or buffers the command received from the FT1.2 frame depending on the SHC command.

Digout stuur: Prints to interpret buffer in the form of "%c%X", where c = A or U and X = 0-15

Digout VU: Prints to interpret buffer in the form of "VU%X%02X", where X = 0-15 (index) and X02 = swap↔ HexalIndex(value)

Digout VS: Prints to interpret buffer in the form of "VS%X%02X", where X = 0-15 (index) and X02 = swap↔ HexalIndex(value)

Dim toggle, hold and percent: Prints to interpret buffer in the form of "D%02X%02X%02X", where 02X(first) = 0-15 (index), 02X(second) = 0-255 (percentage) and 02X(third) = 0-255 (dimtime)

dimaction: 09 = to 100%, 08 = stop and next dim down, 01 =dim to 0, 00 = stop and next dim up

Indicator: changes bool to status

Comfort: Prints data to comfort buffer. Datapointvalue = 200 \* Index + value

##### Parameters

<i>data</i>	<p>Array of all bytes of the ft1.2 frame.</p> <p>Indexes of <i>data</i>: currentDatapointIndex: +0;1 = id of first datapoint (again) , +2 = state byte , +3 = lenght of value, +4-x = value .</p> <p>Raw indexes: 6=subservice code, 7;8=id of first datapoint , 9;10=number of datapoints , 11;12 = id of first datapoint (again) , 13 = state byte , 14 = lenght of value, 15+ = value.</p>
-------------	---

#### 4.3.2.6 processRxBuffer()

```
void processRxBuffer ( )
```

Process a single byte from the RxBuffer and saves it in a global variable. Calls [processMessage\(\)](#) when a message is completed and valid.

### 4.3.3 Variable Documentation

### 4.3.3.1 datapointToSHCMap

```
uint8_t datapointToSHCMap[1023]
```

#### Initial value:

```
= {0,
    DIGOUT_STUUR, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    DIGOUT_VU, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    DIGOUT_VS, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    DIGOUT_STATUS, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    DIGIN_STATUS, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
    DIM_ON_OFF, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    DIM_HOLD, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
    DIM_PERCENT, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
    DIM_FEEDBACK, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,
    PB_DIM_STATUS, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
    ANAIN_VALUES, 11,
    TEMP_VALUES, 12, 12,
    INDOUT_STUUR, 13, 13, 13, 13, 13,
    INDOUT_STATUS, 14, 14, 14, 14, 14, 14,
    COMFORT_STUUR, 15, 15, 15, 15, 15,
    TEMP_ENABLED, 16, 16, 16, 16}
```

Array used to map the knx datapoints to their corresponding SHC type.

### 4.3.3.2 frameDetected

```
bool frameDetected = false
```

Global variable used by [processRxBuffer\(\)](#)

### 4.3.3.3 frameLength

```
uint16_t frameLength = 0
```

Global variable used by [processRxBuffer\(\)](#)

### 4.3.3.4 frameReadIndex

```
uint16_t frameReadIndex = 0
```

Global variable used by [processRxBuffer\(\)](#)

### 4.3.3.5 knx\_cr

```
uint8_t knx_cr = 0b01110011
```

Control byte of baos protocol that is flipped after outgoing message

### 4.3.3.6 knxFrameData

```
uint8_t* knxFrameData
```

Global variable used by [processRxBuffer\(\)](#)

## 4.4 ExternalUse

Functions that are probable and recommended to be used externally.

### Functions

- void [knxReset](#) ()
- bool [knxInterpretBufferHasData](#) ()
- bool [knxComfortBufferHasData](#) ()
- void [knxInterpret](#) (uint8\_t commands)
- void [knxProcessMultiple](#) (uint8\_t max)
- uint16\_t [knxReadComfortBuffer](#) ()
- void [knxInit](#) (int \*status\_voor, int \*ingang\_voor, int \*temp\_measured, int \*analog, int \*dimmer\_voor)

### 4.4.1 Detailed Description

Functions that are probable and recommended to be used externally.

### 4.4.2 Function Documentation

#### 4.4.2.1 knxComfortBufferHasData()

```
bool knxComfortBufferHasData ( )
```

#### Returns

True if unread data in Comfort Buffer

#### 4.4.2.2 knxInit()

```
void knxInit (
    int * status_voor,
    int * ingang_voor,
    int * temp_measured,
    int * analog,
    int * dimmer_voor )
```

Initializes KNX by setting pointer values, formatting the serial connection, attaching the interrupts and sending a knx reset.

#### Parameters

<i>status_voor</i>	SHC digital output status
<i>ingang_voor</i>	SHC digital input status
<i>temp_measured</i>	SHC temperature status
<i>analog</i>	SHC analog status
<i>dimmer_voor</i>	SHC dimmer status

#### 4.4.2.3 knxInterpret()

```
void knxInterpret (
    uint8_t commands )
```

Interpret KNX commands from the interpret buffer if data is available

##### Parameters

<i>commands</i>	Max number of commands to execute
-----------------	-----------------------------------

#### 4.4.2.4 knxInterpretBufferHasData()

```
bool knxInterpretBufferHasData ( )
```

##### Returns

True if unread data in Interpret Buffer

#### 4.4.2.5 knxProcessMultiple()

```
void knxProcessMultiple (
    uint8_t max )
```

Process multiple received knx bytes

##### Parameters

<i>max</i>	Max number of bytes to process
------------	--------------------------------

#### 4.4.2.6 knxReadComfortBuffer()

```
uint16_t knxReadComfortBuffer ( )
```

Reads a byte from the knx comfort buffer

##### Returns

Returns the next knx comfort value if there is one

#### 4.4.2.7 knxReset()

```
void knxReset ( )
```

Send a reset request to the Baos module. Should be done on system start

# Chapter 5

## File Documentation

### 5.1 C:/KNX\_MBedOs/SHC\_V2\_Imported/KNX/knx.cpp File Reference

```
#include "mbed.h"
#include "KNXAddressTypes.h"
#include "hoofd.h"
#include "knx.h"
#include "string.h"
#include "dim.h"
```

#### Macros

- #define [HIBYTE](#)(nWord) ((uint8\_t)((nWord >> 8) & 0x00FF))
- #define [LOBYTE](#)(nWord) ((uint8\_t)(nWord & 0x00FF))
- #define [UART\\_RX\\_BUFFER\\_MAX\\_LENGTH](#) 1000
- #define [UART\\_TX\\_BUFFER\\_MAX\\_LENGTH](#) 100
- #define [KNX\\_INTERPRET\\_BUFFER\\_MAX\\_LENGTH](#) 40
- #define [KNX\\_COMFORT\\_BUFFER\\_LENGTH](#) 50

#### Functions

- bool [txBufferHasData](#) ()
- void [txBufferWrite](#) (uint8\_t \*data, uint16\_t datalength)
- void [rxISR](#) ()
- void [txBufferTransmit](#) ()
- bool [rxBufferHasData](#) ()
- uint8\_t [rxBufferReadByte](#) ()
- bool [checkSumControl](#) (uint8\_t \*data, uint16\_t length)
- void [processMessage](#) (uint8\_t \*data)
- void [processRxBuffer](#) ()
- void [knxWriteData](#) (uint8\_t \*data, uint16\_t datalength)
- void [knxSetOneByteDatapoints](#) (uint16\_t datapointId[], uint8\_t shcType[], uint8\_t value[], uint8\_t datapointsAmount)
- void [knxSetTwoByteDatapoints](#) (uint16\_t datapointId[], uint8\_t shcType[], uint16\_t value[], uint8\_t datapointsAmount)

- uint8\_t \* [knxConvertSwappedStatusIntUint8Array](#) (int intvalue, uint8\_t \*array)
- uint8\_t \* [knxConvertStatusIntUint8Array](#) (int intvalue, uint8\_t \*array, uint8\_t arraysize)
- uint16\_t [log2OfUint16](#) (uint16\_t value)
- uint16\_t [floatToKnxTwoBytes](#) (float value)
- void [knxUpdateDigitalOut](#) ()
- void [knxUpdateDigitalIn](#) ()
- void [knxUpdateAnalogIn](#) (uint8\_t analogIn)
- void [knxUpdateTemps](#) (uint8\_t tempsensor)
- void [knxUpdateDimmers](#) ()
- void [knxUpdateIndicators](#) ()
- void [knxUpdateDimmerPushButtons](#) ()
- void [knxUpdateAll](#) ()
- void [knxReset](#) ()
- bool [knxInterpretBufferHasData](#) ()
- bool [knxComfortBufferHasData](#) ()
- void [knxInterpret](#) (uint8\_t commands)
- void [knxProcessMultiple](#) (uint8\_t max)
- uint16\_t [knxReadComfortBuffer](#) ()
- void [knxInit](#) (int \*status\_voor, int \*ingang\_voor, int \*temp\_measured, int \*analog, int \*dimmer\_voor)

## Variables

- uint16\_t [uartRxBufferNextReadIndex](#) = 0
- uint16\_t [uartRxBufferNextWriteIndex](#) = 0
- uint8\_t [uartRxBuffer](#) [UART\_RX\_BUFFER\_MAX\_LENGTH]
- uint16\_t [uartTxBufferNextReadIndex](#) = 0
- uint16\_t [uartTxBufferNextWriteIndex](#) = 0
- uint8\_t [uartTxBuffer](#) [UART\_TX\_BUFFER\_MAX\_LENGTH]
- uint16\_t [knxInterpretBufferNextReadIndex](#) = 0
- uint16\_t [knxInterpretBufferNextWriteIndex](#) = 0
- char [knxInterpretBuffer](#) [KNX\_INTERPRET\_BUFFER\_MAX\_LENGTH][16]
- uint16\_t [knxComfortBufferNextReadIndex](#) = 0
- uint16\_t [knxComfortBufferNextWriteIndex](#) = 0
- uint16\_t [knxComfortBuffer](#) [KNX\_COMFORT\_BUFFER\_LENGTH]
- int \* [statusVoor](#)
- int \* [ingangVoor](#)
- int \* [tempsMeasured](#)
- int \* [analogInValues](#)
- int \* [dimmerVoor](#)
- DigitalOut [OUT1](#)
- DigitalOut [OUT2](#)
- uint8\_t [knx\\_cr](#) = 0b01110011
- uint8\_t [datapointToSHCMap](#) [1023]
- uint8\_t \* [knxFrameData](#)
- uint16\_t [frameReadIndex](#) = 0
- uint16\_t [frameLength](#) = 0
- bool [frameDetected](#) = false
- uint8\_t [knxDigOutStatus](#) [AMOUNT\_DP\_DIGOUT\_STATUS] = {0}
- uint8\_t [knxDigInStatus](#) [AMOUNT\_DP\_DIGIN\_STATUS] = {0}
- uint8\_t [knxDimOnOffStatus](#) [AMOUNT\_DP\_DIM\_ON\_OFF] = {0}
- uint8\_t [knxAnalogInValues](#) [AMOUNT\_DP\_ANAIN\_VALUES] = {0}
- uint8\_t [knxIndicatorStatus](#) [AMOUNT\_DP\_INDOUT\_STATUS] = {0}
- uint8\_t [knxDimmerPushButtonStatus](#) [AMOUNT\_DP\_PB\_DIM\_STATUS] = {0}
- int [knxTempValues](#) [AMOUNT\_DP\_TEMP\_VALUES] = {0}



## 5.1.1 Detailed Description

Adds KNX functionality to the SHC

## 5.1.2 Macro Definition Documentation

### 5.1.2.1 HIBYTE

```
#define HIBYTE(  
    nWord ) ((uint8_t)((nWord >> 8) & 0x00FF))
```

Function to extract the most significant (higher) byte from a uint16

#### Parameters

<i>nWord</i>	uint16_t to split
--------------	-------------------

#### Returns

The most significant byte

### 5.1.2.2 LOBYTE

```
#define LOBYTE(  
    nWord ) ((uint8_t)(nWord & 0x00FF))
```

Function to extract the least significant (lower) byte from a uint16

#### Parameters

<i>nWord</i>	uint16_t to split
--------------	-------------------

#### Returns

The least significant byte

## 5.1.3 Function Documentation

### 5.1.3.1 rxBufferReadByte()

```
uint8_t rxBufferReadByte ( )
```

Read a single byte from Uart Rx Buffer

#### Returns

The next unread byte in the UART Rx buffer

## 5.1.4 Variable Documentation

### 5.1.4.1 knxDigOutStatus

```
uint8_t knxDigOutStatus[AMOUNT_DP_DIGOUT_STATUS] = {0}
```

end of MessageProcessing

## 5.2 knx.h

```
00001
00004 #ifndef KNX_H
00005 #define KNX_H
00006
00007 #include <stdint.h>
00008 #include "mbed.h"
00009
00010 void knxInit(int* , int* , int* , int* ,int*);
00011 void knxInterpret(uint8_t);
00012 void processRxBuffer();
00013 void knxUpdateDigitalOut();
00014 void knxUpdateDigitalIn();
00015 void knxUpdateAnalogIn(uint8_t);
00016 void knxUpdateTemps(uint8_t);
00017 void knxProcessMultiple(uint8_t);
00018 extern RawSerial knx_uart;
00019 uint16_t knxReadComfortBuffer();
00020 bool knxComfortBufferHasData();
00021 void knxUpdateDimmers();
00022 void knxUpdateIndicators();
00023 #endif
```

## 5.3 KNXAdresTypes.h

```
00001
00002 // datapunten out = data van knx naar SHC, in = data van SHC naar KNX
00003 #define AMOUNT_DP_DIGOUT_STUUR 16 // 16 datapoints 1-16 out
00004 #define AMOUNT_DP_DIGOUT_VU 16 // 16 datapoints 17-32 out
00005 #define AMOUNT_DP_DIGOUT_VS 16 // 16 datapoints 33-48 out
00006 #define AMOUNT_DP_DIGOUT_STATUS 16 // 16 datapoints 49-64 in
00007 #define AMOUNT_DP_DIGIN_STATUS 16 // 16 datapoints 65-80 in
00008 #define AMOUNT_DP_DIM_ON_OFF 8 // 8 datapoints 81-88 out
00009 #define AMOUNT_DP_DIM_HOLD 8 // 8 datapoints 89-96 out
00010 #define AMOUNT_DP_DIM_PERCENT 8 // 8 datapoints 97-104 out
00011 #define AMOUNT_DP_DIM_FEEDBACK 8 // 8 datapoints 105-112 in
00012 #define AMOUNT_DP_PB_DIM_STATUS 8 // 8 datapoints 113-120 in
00013 #define AMOUNT_DP_ANAIN_VALUES 2 // 2 datapoints 121-122 in
00014 #define AMOUNT_DP_TEMP_VALUES 3 // 3 datapoints 123-125 in
00015 #define AMOUNT_DP_INDOUT_STUUR 6 // 6 datapoints 126-131 out
00016 #define AMOUNT_DP_INDOUT_STATUS 6 // 6 datapoints 132-137 in
00017 #define AMOUNT_DP_COMFORT_STUUR 5 // 5 datapoints 138-142 in
00018 #define AMOUNT_DP_TEMP_ENABLED 3 // 3 datapoints 143-145 out
```

```
00019
00020 #define DIGOUT_STUUR 1
00021 #define DIGOUT_VU 2 // 16 datapoints 1-16 out
00022 #define DIGOUT_VS 3 // 16 datapoints 33-48 out
00023 #define DIGOUT_STATUS 4 // 16 datapoints 49-64 in
00024 #define DIGIN_STATUS 5 // 16 datapoints 65-80 in
00025 #define DIM_ON_OFF 6 // 8 datapoints 81-88 out
00026 #define DIM_HOLD 7 // 8 datapoints 89-96 out
00027 #define DIM_PERCENT 8 // 8 datapoints 97-104 out
00028 #define DIM_FEEDBACK 9 // 8 datapoints 105-112 in
00029 #define PB_DIM_STATUS 10 // 8 datapoints 113-120 in
00030 #define ANAIN_VALUES 11 // 2 datapoints 121-122 in
00031 #define TEMP_VALUES 12 // 3 datapoints 123-125 in
00032 #define INDOUT_STUUR 13 // 6 datapoints 126-131 out
00033 #define INDOUT_STATUS 14 // 6 datapoints 132-137 in
00034 #define COMFORT_STUUR 15 // 5 datapoints 138-142 in
00035 #define TEMP_ENABLED 16 // 3 datapoints 143-145 out
00036
00037 const uint16_t START_DIGOUT_STUUR = 1;
00038 const uint16_t START_DIGOUT_VU = START_DIGOUT_STUUR + AMOUNT_DP_DIGOUT_STUUR;
00039 const uint16_t START_DIGOUT_VS = START_DIGOUT_VU + AMOUNT_DP_DIGOUT_VU;
00040 const uint16_t START_DIGOUT_STATUS = START_DIGOUT_VS + AMOUNT_DP_DIGOUT_VS;
00041 const uint16_t START_DIGIN_STATUS = START_DIGOUT_STATUS + AMOUNT_DP_DIGOUT_STATUS;
00042 const uint16_t START_DIM_ON_OFF = START_DIGIN_STATUS + AMOUNT_DP_DIGIN_STATUS;
00043 const uint16_t START_DIM_HOLD = START_DIM_ON_OFF + AMOUNT_DP_DIM_ON_OFF;
00044 const uint16_t START_DIM_PERCENT = START_DIM_HOLD + AMOUNT_DP_DIM_HOLD;
00045 const uint16_t START_DIM_FEEDBACK = START_DIM_PERCENT + AMOUNT_DP_DIM_PERCENT;
00046 const uint16_t START_PB_DIM_STATUS = START_DIM_FEEDBACK + AMOUNT_DP_DIM_FEEDBACK;
00047 const uint16_t START_ANAIN_VALUES = START_PB_DIM_STATUS + AMOUNT_DP_PB_DIM_STATUS;
00048 const uint16_t START_TEMP_VALUES = START_ANAIN_VALUES + AMOUNT_DP_ANAIN_VALUES;
00049 const uint16_t START_INDOUT_STUUR = START_TEMP_VALUES + AMOUNT_DP_TEMP_VALUES;
00050 const uint16_t START_INDOUT_STATUS = START_INDOUT_STUUR + AMOUNT_DP_INDOUT_STUUR;
00051 const uint16_t START_COMFORT_STUUR = START_INDOUT_STATUS + AMOUNT_DP_INDOUT_STATUS;
00052 const uint16_t START_TEMP_ENABLED = START_COMFORT_STUUR + AMOUNT_DP_COMFORT_STUUR;
00053
00054 #define DIGOUT_STUUR_AAN 'A'
00055 #define DIGOUT_STUUR_UIT 'U'
00056 #define DIGOUT_STUUR_VU 'VU'
00057 #define DIGOUT_STUUR_VS 'VS'
```



# Index

- analogInValues
  - GetShcData, [12](#)
- Buffers, [7](#)
  - rxBufferHasData, [8](#)
  - rxISR, [8](#)
  - txBufferHasData, [8](#)
  - txBufferTransmit, [8](#)
  - txBufferWrite, [8](#)
- C:/KNX\_MBedOs/SHC\_V2\_Imported/KNX/knx.cpp, [21](#)
- C:/KNX\_MBedOs/SHC\_V2\_Imported/KNX/knx.h, [24](#)
- C:/KNX\_MBedOs/SHC\_V2\_Imported/KNX/KNXAddressTypes.h,  
[24](#)
- checksumControl
  - MessageProcessing, [14](#)
- datapointToSHCMap
  - MessageProcessing, [16](#)
- dimmerVoor
  - GetShcData, [13](#)
- ExternalUse, [18](#)
  - knxComfortBufferHasData, [18](#)
  - knxInit, [18](#)
  - knxInterpret, [19](#)
  - knxInterpretBufferHasData, [19](#)
  - knxProcessMultiple, [19](#)
  - knxReadComfortBuffer, [19](#)
  - knxReset, [19](#)
- floatToKnxTwoBytes
  - GetShcData, [9](#)
- frameDetected
  - MessageProcessing, [17](#)
- frameLength
  - MessageProcessing, [17](#)
- frameReadIndex
  - MessageProcessing, [17](#)
- GetShcData, [9](#)
  - analogInValues, [12](#)
  - dimmerVoor, [13](#)
  - floatToKnxTwoBytes, [9](#)
  - ingangVoor, [13](#)
  - knxConvertStatusIntUInt8Array, [10](#)
  - knxConvertSwappedStatusIntUInt8Array, [10](#)
  - knxUpdateAll, [11](#)
  - knxUpdateAnalogIn, [11](#)
  - knxUpdateDigitalIn, [11](#)
  - knxUpdateDigitalOut, [11](#)
  - knxUpdateDimmerPushButtons, [11](#)
  - knxUpdateDimmers, [11](#)
  - knxUpdateIndicators, [12](#)
  - knxUpdateTemps, [12](#)
  - log2OfUInt16, [12](#)
  - OUT1, [13](#)
  - OUT2, [13](#)
  - statusVoor, [13](#)
  - tempsMeasured, [13](#)
- HIBYTE
  - knx.cpp, [23](#)
- ingangVoor
  - GetShcData, [13](#)
- knx.cpp
  - HIBYTE, [23](#)
  - knxDigOutStatus, [24](#)
  - LOBYTE, [23](#)
  - rxBufferReadByte, [23](#)
- knx\_cr
  - MessageProcessing, [17](#)
- knxComfortBufferHasData
  - ExternalUse, [18](#)
- knxConvertStatusIntUInt8Array
  - GetShcData, [10](#)
- knxConvertSwappedStatusIntUInt8Array
  - GetShcData, [10](#)
- knxDigOutStatus
  - knx.cpp, [24](#)
- knxFrameData
  - MessageProcessing, [17](#)
- knxInit
  - ExternalUse, [18](#)
- knxInterpret
  - ExternalUse, [19](#)
- knxInterpretBufferHasData
  - ExternalUse, [19](#)
- knxProcessMultiple
  - ExternalUse, [19](#)
- knxReadComfortBuffer
  - ExternalUse, [19](#)
- knxReset
  - ExternalUse, [19](#)
- knxSetOneByteDatapoints
  - MessageProcessing, [14](#)
- knxSetTwoByteDatapoints
  - MessageProcessing, [15](#)
- knxUpdateAll

- GetShcData, 11
- knxUpdateAnalogIn
  - GetShcData, 11
- knxUpdateDigitalIn
  - GetShcData, 11
- knxUpdateDigitalOut
  - GetShcData, 11
- knxUpdateDimmerPushButtons
  - GetShcData, 11
- knxUpdateDimmers
  - GetShcData, 11
- knxUpdateIndicators
  - GetShcData, 12
- knxUpdateTemps
  - GetShcData, 12
- knxWriteData
  - MessageProcessing, 15
- LOBYTE
  - knx.cpp, 23
- log2OfUint16
  - GetShcData, 12
- MessageProcessing, 13
  - checkSumControl, 14
  - datapointToSHCMap, 16
  - frameDetected, 17
  - frameLength, 17
  - frameReadIndex, 17
  - knx\_cr, 17
  - knxFrameData, 17
  - knxSetOneByteDatapoints, 14
  - knxSetTwoByteDatapoints, 15
  - knxWriteData, 15
  - processMessage, 16
  - processRxBuffer, 16
- OUT1
  - GetShcData, 13
- OUT2
  - GetShcData, 13
- processMessage
  - MessageProcessing, 16
- processRxBuffer
  - MessageProcessing, 16
- rxBufferHasData
  - Buffers, 8
- rxBufferReadByte
  - knx.cpp, 23
- rxISR
  - Buffers, 8
- statusVoor
  - GetShcData, 13
- tempsMeasured
  - GetShcData, 13
- txBufferHasData
  - Buffers, 8
- txBufferTransmit
  - Buffers, 8
- txBufferWrite
  - Buffers, 8