KNX for OPC UA

Patrick Ruß
Reg.#: 0728142
Automation Systems Group
Vienna University Of Technology

November 14, 2011

Abstract

The existence of several protocols and products in the area of home and building automation brings along the question of interoperability. There is a need for an overlaying system which can easily integrate and control all kinds of fieldbus protocols. This paper gives an insight into OPC UA and KNX, on one hand, and, on the other hand, presents an approach to combine these technologies. Thereby the strong information modelling capabilities of OPC UA will be used, which is ideally suited for modelling different data models.

Statement

Hereby I declare that this work has been written autonomously, that all used sources and utilities are denoted accordingly and that these points of the work - including tables, maps and figures - which where taken from other creations or the Internet have been marked as borrowing by quoting the original sources. This document at hand will not be submitted to any other course.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Contents

1 Introduction

1.1 Motivation

As home and building automation is getting used more and more, a variety of automation products of different vendors are coming up. These different products use different technologies that are based on communication protocols that are incompatible with each other. Therefore a proprietary mapping between these technologies (by e.g., using gateways) is necessary. Moreover, this can only be done at great expense. This being the case, the need for interoperability becomes apparent. Naturally, a possible solution for this problem should be extensible and platform independent. These requirements are fulfilled by OPC Unified Architecture (OPC UA) [1], which, in this paper, will be combined with KNX [2] using information modelling.

1.2 Aim of this work

The main objective of this work is to showcase a possible implementation of how KNX systems can be modelled and represented using OPC UA. As a prerequsite, this paper aims at conveying a basic understanding of the used technologies.

1.3 Structure

The paper is structured as follows: In the following section the reader gains an insight into the functionality of and the possibilities of KNX. Section 2 describes OPC UA and exposes details of information modelling. The third section presents the implementation of a KNX-OPC UA interface using information modelling.

2 KNX

2.1 Introduction to KNX

In May 1999, the members of the EHSA (European Home System Association), the EIBA (European Installation Bus Association) and the BCI (BatiBUS Club International) founded the KNX Association and decided to merge their three standards, EHS, EIB and BatiBUS, into KNX with the goal to define a standardized and open communication standard for home and building automation. The founding members are as follows: Albrecht JUNG GmbH & Co.KG, Bosch Telecom GmbH, Delta Dore S.A, Électricité de France, Electrolux AB, Hager Holding GmbH, Merten GmbH & Co.KG, Siemens AG, Division A&D ET and Siemens Building Technologies Ltd., Landis&Staefa Division.

Since December 2003, the KNX protocol has been included in the EN 50090 series of standards. In November 2006, the specification became part of the international standard ISO/IEC 14543-3-x [3]. The current valid standard is the KNX specification 2.0 [2]. Furthermore the KNX Association is the publisher of the so called "KNX Handbook", which contains the complete system specifications and is available for members of the KNX Association.

KNX was designed to control

- Lighting and
- HVAC (heating, ventilating and air condition)

in a comfortable, flexible, energy-efficient and economic way.

To ensure full mutual compatibility, all KNX products have to pass several certification tests performed by third party testing labs, which test whether a certain product conforms to the KNX specification. In this way, it is possible to combine KNX products of different manufacturers in a home or building.

2.2 Physical media

KNX defines several ways for how to communicate over physical media, which is also depicted in the overview of the KNX architecture in Figure 1.

- Twisted Pair 1 (TP1)
- KNX RF (Radio Frequency)
- Powerline (PL 110)
- KNXnet/IP
- KNX/IP

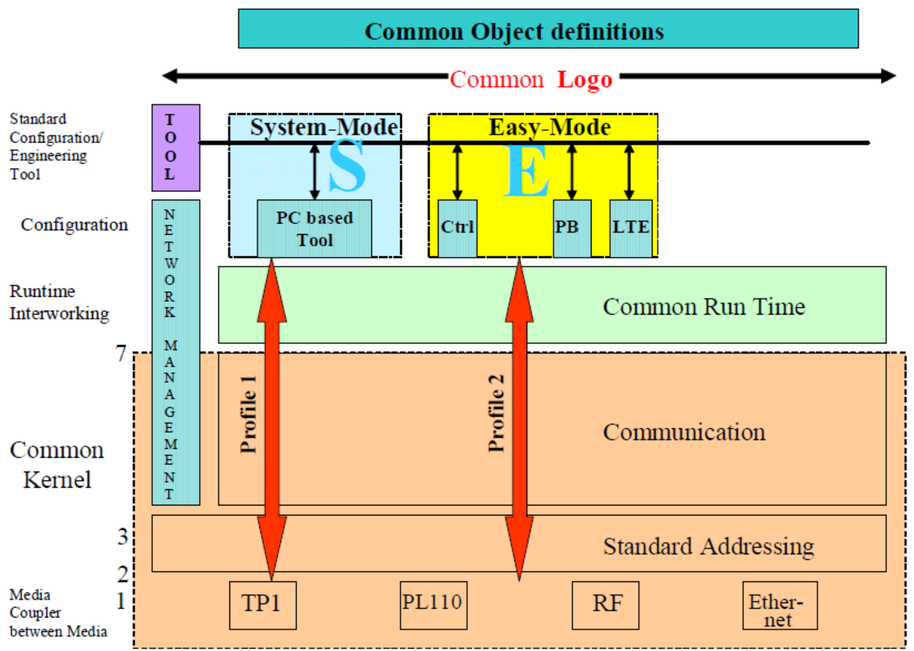


Figure 1: KNX architecture [2]

Currently, KNX TP1 is the most used physical medium. It uses shielded Twisted Pair cables, through which the signal as well as 30V DC link power is transferred. For data transmission, a balanced baseband signal coding is used with a baud rate of 9600 bits per second. TP1 allows a total cable length of 1000 m per physical segment and a maximum distance of 700 m between two devices. There are no restrictions with regard to topology wiring. Some details on possible KNX topologies will be illustrated in Section 2.4.3. Medium access on TP1 is controlled by using CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) with bitwise arbitration.

KNX RF enables KNX devices to communicate via radio by using a medium frequency of 868.3 MHz in the ISM (Industrial, Scientific and Medical)-Band for short range devices. KNX RF devices can transfer data with up to 16.4 kbit/s, using Frequency Shift Keying (FSK) Modulation. It allows bidirectional as well as unidirectional communication.

With KNX Powerline 110, KNX devices can be connected within a 230V power grid. PL110 modulates signals with Spread Frequency Shift Keying (SFSK) and can reach a speed of up to 1200 bits per second. The wiring of PL110 devices is unrestricted by specification, but in real-world applications it is constrained by the given power network. For medium access control, TDMA (Time Division Multiple Access) is used.

KNX/IP devices communicate directly via IP, which has several advantages: First, there is no need to install additional cables, as already existing office or home LANs can be used easily. Moreover, due to the high bandwidth of IP, new possibilities like multimedia transmission are imaginable. It may be possible that KNX/IP will replace TP1 in the future.

In comparison with KNX/IP, KNXnet/IP routers are used to interconnect KNX networks over IP networks. The KNX devices themselves are not directly connected with an IP network. Both KNXnet/IP and KNX/IP use Ethernet as medium and control access to it via CSMA/CD (Carrier Sense Multiple Access with Collision Detection).

2.3 Topology

KNX allows up to 65536 devices to be installed. Each device gets its own individual address in a 16-bit address space. Subtracting the addresses reserved for couplers, there remain 61455 addresses for KNX terminal devices. This amount can decrease in consequence of implementation or environmental factors.

Logical subnetworks can be built with the aid of lines which can contain 256 devices in each case. Lines may be grouped together with a main line into an area. Up to 15 of these areas, connected with a backbone line, make an entire domain. In order to link lines and areas, KNX uses line repeaters, line couplers and backbone couplers. A possible logical KNX topology is depicted in Figure 2. This logical structure originates from the structure of a building.

2.4 Interworking

Since devices in a KNX network have to share process control data, Interworking is the major part of the KNX technology and is defined as

> *The situation where products sending and receiving messages can properly understand signals and react on them without additional equipment.* [4]

The phrase "without additional equipment" is particularly important for communication between products of different manufacturers, be it in the same application or across application borders (Cross Discipline Interworking). To achieve this goal, it is necessary to define the Interworking model on a very high level of abstraction, which is done in the "Application Interworking Specifications", where functional blocks and datapoints are defined.

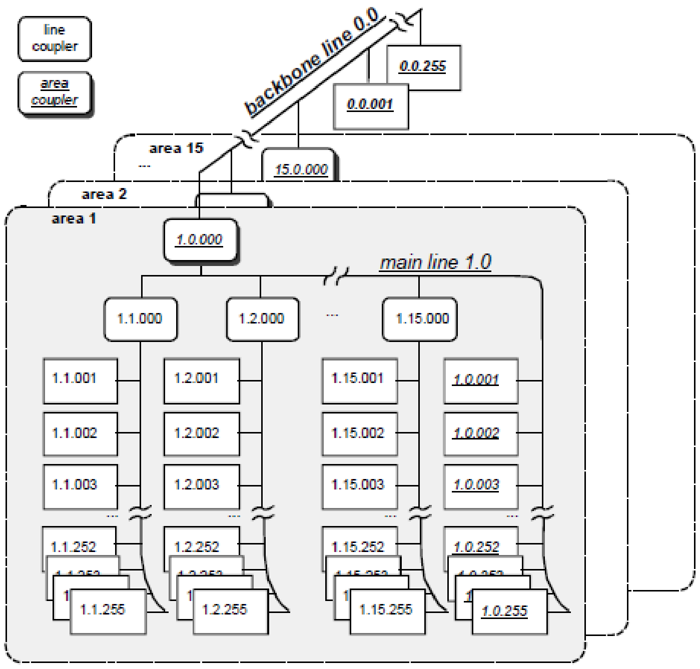


Figure 2: Logical topology of KNX [2]

2.4.1 Functional blocks

A functional block describes the standard specification of the chosen solution for one given task of an application. The implementation of the data points contained in and their functionality is left to product developers. Figure 3 shows the standard representation of a functional block.

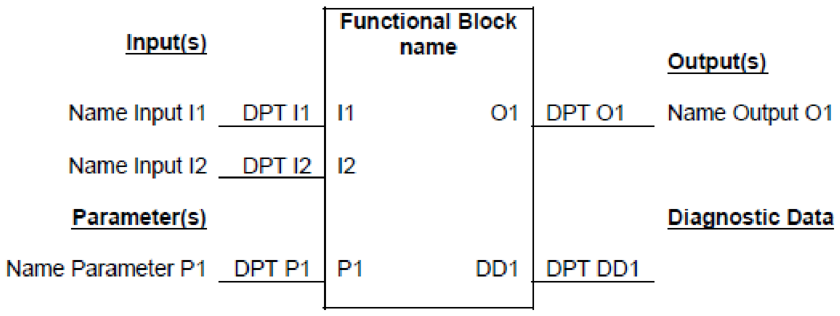
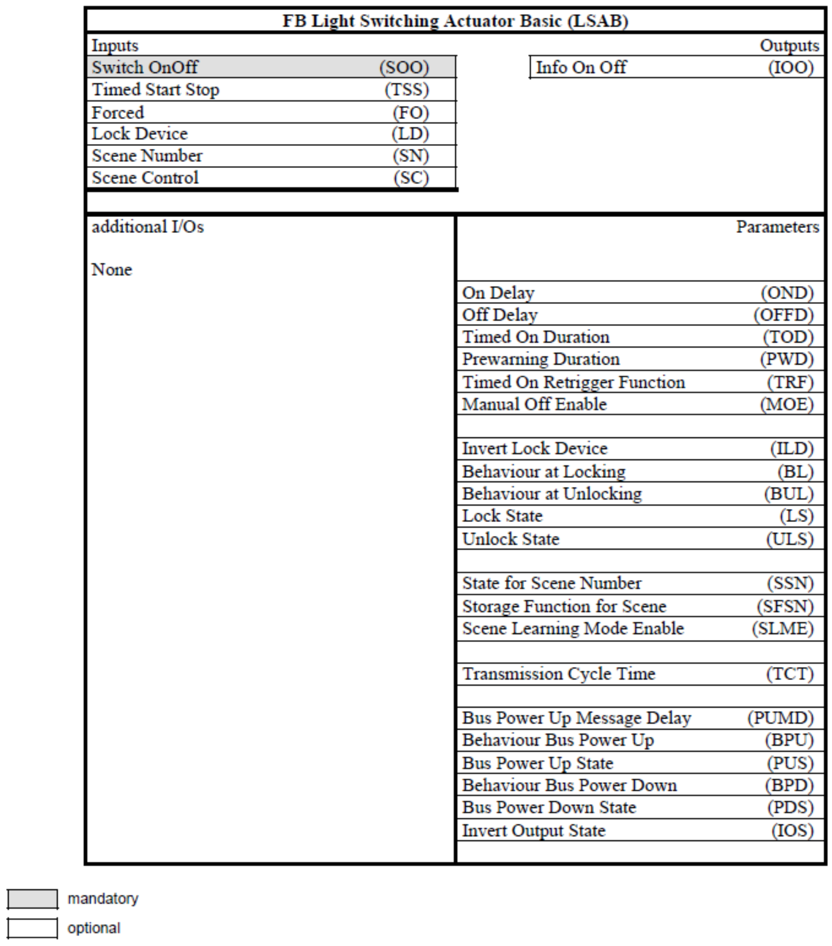


Figure 3: Standard representation of a functional block [2]

Functional blocks consist of data points which can be inputs, outputs, parameters or diagnostic data. Parameters are special inputs which determine the way outputs are generated out of inputs. Diagnostic Data is a special form of an output.

It is only used for debugging during installation, service and maintenance. One or more functional blocks can be grouped to form a device.

A specific example of a functional block is the Light Switching Actuator Basic (LSAB), as depicted in Figure 4. This functional block supports the binary switching of light.



FB Light Switching Actuator Basic (LSAB)

Inputs		Outputs	
Switch OnOff	(SOO)	Info On Off	(IOO)
Timed Start Stop	(TSS)		
Forced	(FO)		
Lock Device	(LD)		
Scene Number	(SN)		
Scene Control	(SC)		

additional I/Os	Parameters	
None		
	On Delay	(OND)
	Off Delay	(OFFD)
	Timed On Duration	(TOD)
	Prewarning Duration	(PWD)
	Timed On Retrigger Function	(TRF)
	Manual Off Enable	(MOE)
	Invert Lock Device	(ILD)
	Behaviour at Locking	(BL)
	Behaviour at Unlocking	(BUL)
	Lock State	(LS)
	Unlock State	(ULS)
	State for Scene Number	(SSN)
	Storage Function for Scene	(SFSN)
	Scene Learning Mode Enable	(SLME)
	Transmission Cycle Time	(TCT)
	Bus Power Up Message Delay	(PUMD)
	Behaviour Bus Power Up	(BPU)
	Bus Power Up State	(PUS)
	Behaviour Bus Power Down	(BPD)
	Bus Power Down State	(PDS)
	Invert Output State	(IOS)

▨	mandatory
☐	optional

Figure 4: Light Switching Actuator Basic (LSAB) [2]

2.4.2 Data points

Data points are the interfaces of a functional block via which data can be received and/or transmitted. Every data point definition consists of the following elements:

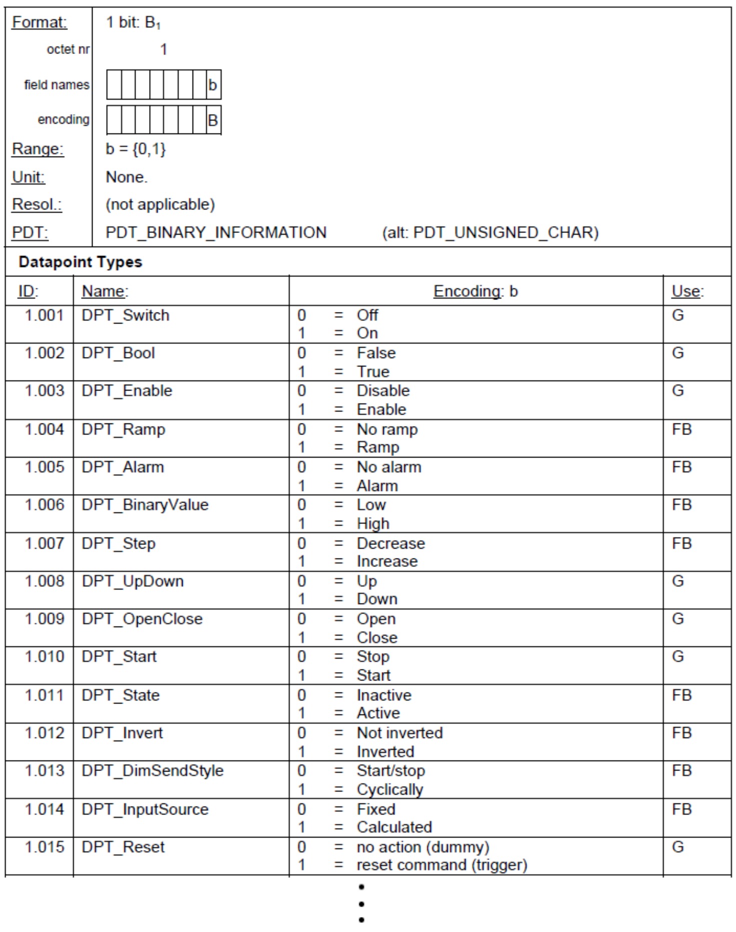
Format describes the length and semantics of the fields used to build up the data point type.

Encoding describes how data shall be coded using the given format.

Range describes the possible range of values that can may be used. This can be a maximum and a minumum or a list of possible values.

Unit describes the unit of the information carried by the data point type.

Figure 5 shows a part of the specification of the data point types B1, i.e., all data point types that represent one binary value. Although the data point value can only be "0" or "1", the semantics can differ in each case. The column "Use" specifies whether this datapoint type can be used without restrictions. Datapoint types with use G (General) can be used without any restrictions. Datapoint types with use FB (Functional Block) are subject to regulations and shall only be used for implementations of standard functional blocks where this datapoint type is used. [2]



Format:	1 bit: B_1
octet nr	1
field names	b
encoding	B
Range:	b = {0,1}
Unit:	None.
Resol.:	(not applicable)
PDT:	PDT_BINARY_INFORMATION (alt: PDT_UNSIGNED_CHAR)

Datapoint Types

ID:	Name:	Encoding: b	Use:
1.001	DPT_Switch	0 = Off 1 = On	G
1.002	DPT_Bool	0 = False 1 = True	G
1.003	DPT_Enable	0 = Disable 1 = Enable	G
1.004	DPT_Ramp	0 = No ramp 1 = Ramp	FB
1.005	DPT_Alarm	0 = No alarm 1 = Alarm	FB
1.006	DPT_BinaryValue	0 = Low 1 = High	FB
1.007	DPT_Step	0 = Decrease 1 = Increase	FB
1.008	DPT_UpDown	0 = Up 1 = Down	G
1.009	DPT_OpenClose	0 = Open 1 = Close	G
1.010	DPT_Start	0 = Stop 1 = Start	G
1.011	DPT_State	0 = Inactive 1 = Active	FB
1.012	DPT_Invert	0 = Not inverted 1 = Inverted	FB
1.013	DPT_DimSendStyle	0 = Start/stop 1 = Cyclically	FB
1.014	DPT_InputSource	0 = Fixed 1 = Calculated	FB
1.015	DPT_Reset	0 = no action (dummy) 1 = reset command (trigger)	G

⋮

Figure 5: Data point types B1 [2]

2.4.3 Communication model

As already pointed out in Section 2.3, each KNX device must have its unique physical address. Once assigned, this address is stored permanently in its EEPROM. Physical addresses can be assigned randomly, but should factor in the building layout. This means, for example, three devices that are located physically next to each other, should receive consecutive addresses like 3.5.1, 3.5.2, 3.5.2. Besides clearly identifying a device, a physical address also provides information on the topographical position as an address follows the format area.line.device. For the given example this would mean that the three devices are part of line 5 and area 3.

Multicast group addresses are used to enable communication between two or more KNX devices. There are two types of them:

Two-level addressing A group address consists of a main group and a sub group.

Three-level addressing A group address consists of a main group, a middle group and a subgroup.

Group addresses are independent from a device's location. They represent logical communication links. If an actuator in a communication group changes a value, all other devices in this group are informed about the change and enabled to react. A simple example of communication via group objects would be a light switch and two lights. All devices have a data point which shares the same group address. When the light switch is pressed, this information is sent to all other devices in this group object, in this exemplary case, the two lights, which are turned on, or, if pressed again, turned off.

3 OPC Unified Architecture

3.1 History of OPC

Since PC- and software based automation systems became more popular in industrial automation in the 1990s, a variety of vendor specific field bus systems, protocols and interfaces have arisen. These different products often pursue the same goal, but accomplish it in different ways, a fact that makes it difficult to let them communicate with each other. Therefore, products of different vendors can be hardly used together in one big superior system. For this reason, the OPC Foundation was founded in 1995 consisting of Fisher-Rosemount, Rockwell Software, Opto 22, Intellution, und Intuitive Technology. As the first result of their work, they published the first OPC DA (Data Access) specification in August 1996. This specification heavily relies on the usage of Microsoft's COM (Component Object Model) and DCOM (Distributed COM).

As time passed, the OPC Foundation has published several other OPC specifications to react to new extended needs of their users. Among many others, three important specifications are:

- OPC Data Access (OPC DA)
- OPC Alarm & Events (OPC A&E)
- OPC Historical Data Access (OPC HDA)

As the name already implies, Data Access describes the read and write access of automation data, whereas Alarm & Events is responsible for event based communication and Historical Data Access provides services to access historized data of automation products. Each specification is implemented as an independent server.

By using Microsoft's COM and DCOM API, the OPC Foundation can release their products faster than others can do, which is the decisive factor of OPC's success and which makes the majority of automation products vendors join the foundation. But there are pros and cons either way. First, COM and DCOM are both platform dependent so they can only be used on Microsoft-based systems and, secondly, DCOM is highly sophisticated in its terms of configuration and cannot be used for communication over the Internet. [5]

OPC XML-DA is the first try to make OPC platform independent. As its name implies, XML-DA adopts the XML format which is widespread over all platforms. Instead of using COM and DCOM for communication it makes use of HTTP/SOAP and Web Services to provide the same functionality as OPC DA in a platform independent way. But also this new OPC specification has its disadvantages, for instance a very high resource consumption and, compared to COM, the bad performance of XML, which was evaluated in [6]. Hence, this new specification cannot entirely convince the users.

3.2 OPC Unified Architecture

In the course of time, OPC has become very popular and got used in many fields of application, including some where OPC was never thought to be used. As OPC XML-DA cannot satisfy in terms of both platform indepence and performance, OPC DA still needs to be used. Let us again summarize the insufficiencies in a neat arrangement.

1. OPC can only be used on Microsoft-based platforms since it uses COM/D-COM.
2. Due to the used communication protocol, DCOM cannot be used over the Internet.
3. The independent servers of DA, H&E and HDA are complex to handle.
4. It lacks in security.

Finally, OPC Unified Architecture shall overcome all these disadvantages and create a fast and platform independent technology that can become a worthy successor of OPC DA. The new OPC UA specification doesn't only remove the COM dependency, but also introduces a new way of modelling automation data, which is described in the following sections.

The OPC UA specification is a multi-part specification which is divided into 13 parts and replaces the OPC DA specification and all other OPC specifications in all respects. To be able to use existing OPC DA devices there exist so called OPC UA wrappers. Figure 6 shows an overview of the OPC UA specifications.

3.3 Information modelling in OPC UA

3.3.1 Why use information modelling?

In classic OPC DA, only pure data and no additional information on how to interpret the read value is available. An example of an OPC DA datapoint is the speed of a cooling fan. To interpret the speed value, nothing more than a tag name and some elementary information like the measuring unit will be provided. With the new concepts of OPC UA, it is now possible to provide more information on a datapoint's semantics. The exemplary cooling fan can now be an instance of a specific device type which allows clients to browse for specific types. As the usage of types and instances already discloses, OPC UA uses object orientated modelling structures wherewith even complex structures can be modeled.

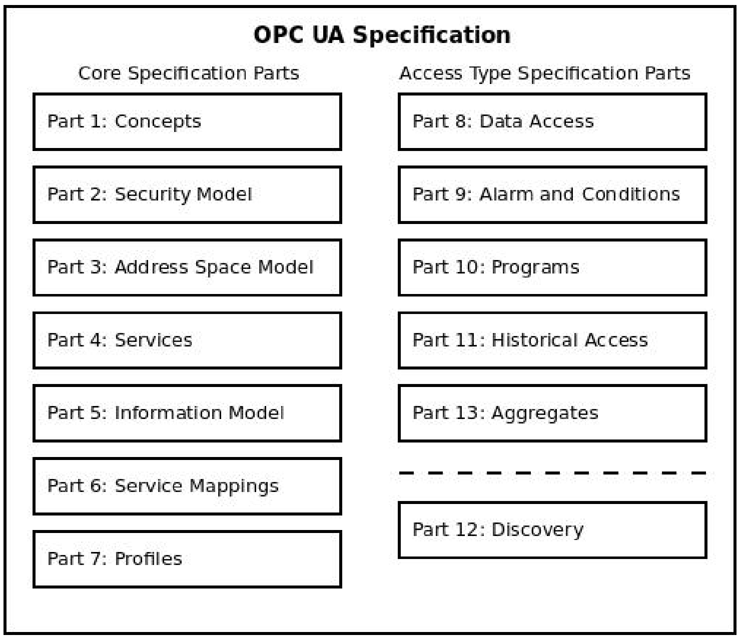


Figure 6: OPC UA multi-part specification [1]

The OPC UA specification only defines the fundamental infrastructure for information modelling. How information finally gets modeled in detail, is left to the vendors. Nevertheless the OPC Foundation wants to ensure a unified modelling style, which is the reason why there is a predefined base frame vendors can build on, i.e., there already exist some basic *DataTypes*, *VariableTypes* and more from the OPC Foundation, which can be extended by vendors to meet their requirements.

The following listing gives an overview of the basic principles of information modelling in OPC UA [5].

Object-orientated modelling Object-orientated modelling benefits from the fact that type definitions can be reused by creating an infinite number of instances. Type hierarchies and inheritance ease the modelling of data.

Type information Not only instances of a type get stored in the information model, but also their type definitions. This means that a client can browse for type definitions in exactly the same way it browses for datapoints. By having type information available, a client can filter datapoints for specific types.

Full meshed node networks OPC UA information modelling uses *Nodes* and *References*, which allows the creation of every thinkable model.

Extensibility Already existing types can be extended to fit your personal needs.

This doesn't only work for ObjectTypes, but also for *VariableTypes*, *ReferenceTypes*, *DataTypes*, etc..

No model limitations Since there are no limitations on how to make a model, it is very easy to port existing models to OPC UA.

Modelling on server side Only an OPC UA Server needs to save an information model. Clients or Servers with a Client part can browse a server's address space.

In the next sections, some basic concepts and components will be introduced, which will be needed for the practical part.

3.3.2 Nodes

The starting point of every information model is a *Node*, which contains *Attributes* and *References*. *References* again point to other *Nodes*. Figure 7 illustrates this node model.

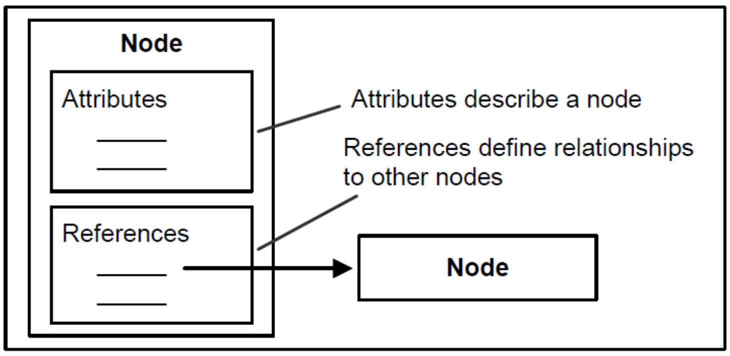

Node
Attributes
References
Attributes describe a node
References define relationships to other nodes
Node

Figure 7: Node model of OPC UA [1]

Attributes are data elements of a *Node*, which can be accessed with the Services *Read*, *Write*, *Query* and *Subscription/MonitoredItem*. The concepts of Services will be discussed in Section 3.4. Every *Attribute* consists of an *Attribute ID*, a *Description*, a *Name*, a *DataType* and an indicator, which determines if the attribute is mandatory or optional. There are some basic *Attributes* which appear in every *Node* and which are listed in Table 1. There may be additional *Attributes* depending on the *NodeClass* of the *Node*.

The *NodeId* is needed to identify a *Node* conclusively. Depending on the *NodeClass*, several additional *Attributes* are added to the basic *Attributes*. The *BrowseName* of a *Node* is used as textual representation which is shown and used when browsing nodes. It is a non-localized String, which is the reason why it shouldn't be used as node name. The localized name of a *Node* is stored in the Attributes

Attribute	DataType
NodeId	NodeId
NodeClass	NodeClass
BrowseName	QualifiedName
DisplayName	LocalizedText
Description	LocalizedText
WriteMask	UInt32
UserWriteMask	UInt32

Table 1: Standard Attributes of a *Node* [5]

DisplayName and *Description*, which are therefore usable as node name. The last two *Attributes* specify which *Attributes* of a *Node* are readable in general and individually by a user. [5]

3.3.3 References

References connect two *Nodes* with each other. *References* are not *Nodes*, but their type is expressed within the address space. Table 2 lists the *Attributes* of a *ReferenceType*.

Attribute	DataType
IsAbstract	Boolean
Symmetric	Boolean
InverseName	LocalizedText

Table 2: Standard attributes of a ReferenceType [5]

The *Attribute IsAbstract* defines whether this *ReferenceType* is abstract. Abstract *ReferenceTypes* are used to organize a type hierarchy and to group other *ReferenceTypes*. The *Attribute Symmetric* determines whether the meaning of the reference is the same for both directions. For non-abstract and asymmetric *RerefenceTypes* it is mandatory to define an *InverseName* for inverse navigation.

To take advantage of object orientated modelling, *ReferenceTypes* are organized in a type hierarchy that allows to extend existing reference types and to create more spezialized ones. [5]

3.3.4 Objects, Variables and Methods

Modelling the real world in an OPC information model is like modelling it in an object-orientated programming language with objects, variables and methods. *Objects* consist of one or more variables and can execute methods.

Nodes of the *NodeClass Variable* represent a value. Depending on the value type, a *Variable* is called a *Property* or a *DataVariable*. *Properties* are used for “virtual“ data, e.g. set points and engineering units. *DataVariables* are used to model real world data, for instance a room temperature. Table 3 summarizes the additional attributes for *Variables*.

Attribute	**DataType**
Value	Specified by other Attributes
DataType	NodeId
ValueRank	Int32
ArrayDimensions	UInt32[]
AccessLevel	Byte
UserAccessLevel	Byte
MinimumSamplingInterval	Duration
Historizing	Boolean

Table 3: Additional Attributes for Variables [5]

The three *Attributes DataType*, *ValueRank* and *ArrayDimensions* specify the data type of the *Value* Attribute. The *Attribute AccessLevel* specifies whether the *Value* and the history of the value is readable or writeable. *UserAccessLevel* has the same function as *AccessLevel* but takes user rights into account. The *Attribute MinimumSamplingInterval* defines the minimum time it takes to detect changes of the *Value* Attribute. This is useful for values not directly managed by the server, for example temperature values of a temperature sensor. The last *Attribute*, *Historizing*, specifies whether the *Value* shall be historized.

Nodes of the *NodeClass Method* represent methods that can be called by a client and returns a value. *Methods* are specified through their input and output arguments. They ought to be fast in execution, that is, if a client calls a method via the *Call* Service, the response of this call should already contain the return value of the invoked method. Otherwise *Programs* should be used which are designed for more time expensive tasks. Table 4 shows additional *Attributes* and Standard *Properties* for *Methods*.

Attribute	**DataType**
Executable	Boolean
UserExecutable	Boolean
InputArguments	Argument[]
OutputArguments	Argument[]

Table 4: Additional Attributes for Methods [5]

The *Attributes Executable* and *UserExecutable* define whether the *Method* is executable in general and by users. The optional Properties *InputArguments* and *OutputArguments* define arrays of input and output arguments.

Nodes of the *NodeClass Object* are used for structuring an information model. They do not contain data other than the standard node *Attributes*. Values of an *Object* get represented via *Variables*, so *Objects* do not contain a *Value Attribute* like *Variables*. Besides the standard node attributes, *Nodes* of the *NodeClass Object* have only one additional *Attribute*, which is named *EventNotifier* and which determines whether a client can subscribe to the *Object* to receive Events or not. [5]

3.3.5 Type definitions

Like instances of *Objects* and *Variables*, also their type definitions are defined in the information model. All advantages of object-orientated modelling can be used to model type hierarchies. This means, for example, that you can create a model of a couple of temperature sensors by defining a base temperature sensor type and then let other vendor-specific temperature sensors inherit the node structure they have in common. With this approach, for instance, OPC UA clients can be programmed using the knowledge of the used types to create a device tailored graphical UI.

OPC UA uses the *NodeClass ObjectType* for object definitions and the *NodeClass VariableType* for type definitions. *Methods* are defined by their browse names and arguments and are bound to a specific *ObjectType*, so they don't need to have their own type definitions.

ObjectTypes and *VariableTypes* can be divided into the simple and the complex ones. Simple *ObjectTypes* are used for organizing the address space. An example would be *FolderType* which is used to group other objects. Simple *VariableTypes* can be used to ease the interpretation of a *Variable*. Generally speaking, simple *ObjectTypes* and *VariableTypes* cannot contain other nodes in their definition. For modelling such cases you have to use complex type definitions.

Complex types define a structure of *Nodes* beneath them, that is available on each instance of the *Object- or VariableType*.

Besides the standard attributes for nodes, an object definition only adds the *Attribute IsAbstract*. This *Attribute* defines whether the *ObjectType* is abstract.

Table 5 lists all additional *Attributes* for *VariableTypes*.

Like *ObjectTypes*, *VariableTypes* can be simple and complex as well. The concept is the same as above. [5]

Attribute	DataType
Value	specified by other Attributes
DataType	NodeId
ValueRank	ValueRank
ArrayDimensions	UInt32[]
IsAbstract	Boolean

Table 5: Additional Attributes for VariableTypes [5]

3.3.6 DataTypes

As already said in Section 3.3.4, the *Attribute DataType* defines the type of a *Variable* together with *ValueRank* and *ArrayDimensions* which again specify whether the *Variable* is a scalar value or an array. This enables servers to define their own *DataTypes* and clients to access information about them browsing the information model.

There are four sorts of *DataTypes* in OPC UA [5]:

Built-in DataTypes These are types, predefined in the OPC UA specification, which cannot be extended by vendor-specific DataTypes. They include *DataTypes* like Int32, Boolean, Double and also OPC UA specific types like *NodeId*, LocalizedText and QualifiedName.

Simple DataTypes Simple *DataTypes* are subtypes of built-in *DataTypes*. An example of a Simple *DataType* is Duration which is a subtype of Double. It is handled on the wire as Double, but the user can read the *DataType* Attribute and interpret the value as duration.

Enumeration DataTypes These *DataTypes* represent a discrete set of named values.

Structured DataTypes Structured *DataTypes* are the most powerful tool when it comes to defining a complex, user-defined model.

3.4 Services

Services are used in the OPC UA client/server model to define data communication on application level. They provide methods for clients to access data of an information model stored on a server. Unlike OPC DA, OPC UA separates the definition of the services from the used transport protocol and programming environment. This is a fundamental difference and means that OPC UA is not bound to a specific transport protocol like COM.

To make this possible, it is necessary to define Services on a very abstract level. This abstract definition can then be applied to several transport mechanisms by the usage of OPC UA Stacks, which are available in different programming languages. Currently two transport mechanisms are defined: OPC UA Binary based on TCP/IP and a Web Service transport mechanism based on SOAP/HTTP.

The OPC UA Services are grouped into the following Service Sets.

- Discovery Service Set
- SecureChannel Service Set
- Session Service Set
- NodeManagement Service Set
- View Service Set
- Query Service Set
- Attribute Service Set
- Method Service Set
- MonitoredItem Service Set
- Subscription Service Set

3.4.1 Basic concepts

All Services in the Service Sets follow the request and response pattern. This means, in order to call a Service's method in a server, the client has to send a request. After this request has been processed by the server, it sends back a response. By using this pattern, all Services are asynchronous by definition. A client does not have to wait for the completion of its request and can invoke other requests in the meantime.

As the client and the server do not have to run in the same process or network node, there can always be transmission failures. For this reason, the client can configure its own timeouts to define how long it will wait before taking action.

4 KNX for OPC UA

The main objective of this paper is to show how to combine the two technologies, KNX and OPC UA, described in the previous sections, in Java. For this reason, the "KNX for OPC UA" driver, which will be presented in the following sections, is based on the KNX Java library *Calimero*, which offers all desired possibilities to easily access KNX data points via KNXnet/IP. The OPC UA part builds on the *Comet OPC UA Driver Framework* from HB-Softsolution [7]. This framework provides a driver interface and a server instance which integrates the KNX-specific driver.

4.1 Laboratory environment

For the practical realization of the data model presented in the following sections three lights were on hand which were integrated into a KNX network via a KNXnet/IP-Router. These three lights were assigned the group addresses 0/0/1, 0/0/2 and 0/0/3. Figure 8 shows the laboratory environment.

4.2 HB-Softsolution Model Designer

OPC UA information models are stored in XML format. This format can be easily processed by computers, but as the amount of elements in an XML file grows, the data gets harder and harder to read for the human eye. For this reason, HB-Softsolution provides a model designer which enables the user to create information models with a graphical user interface, which is depicted in Figure 9.

With the help of this tool it is possible to define OPC UA namespaces and add nodes and references. It allows to add new types and instances of references, variables and objects.

4.3 KNX information model for OPC UA

In order to build an information model for KNX, it is necessary to map the KNX Interworking model with its functional blocks and data point types to nodes and references in the OPC UA address space. As already stated in [8], the best way to do this is to use complex OPC UA objects.

As with the current version of the HB-Softsolution Model Designer it was not possible to create data types, the given model from [8] had to be adapted by replacing all complex data types with primitive data types like String or Boolean to map KNX data point types. Figure 10 shows the KNX information model which will be used in the following sections. It introduces new reference types, variable types and object types.

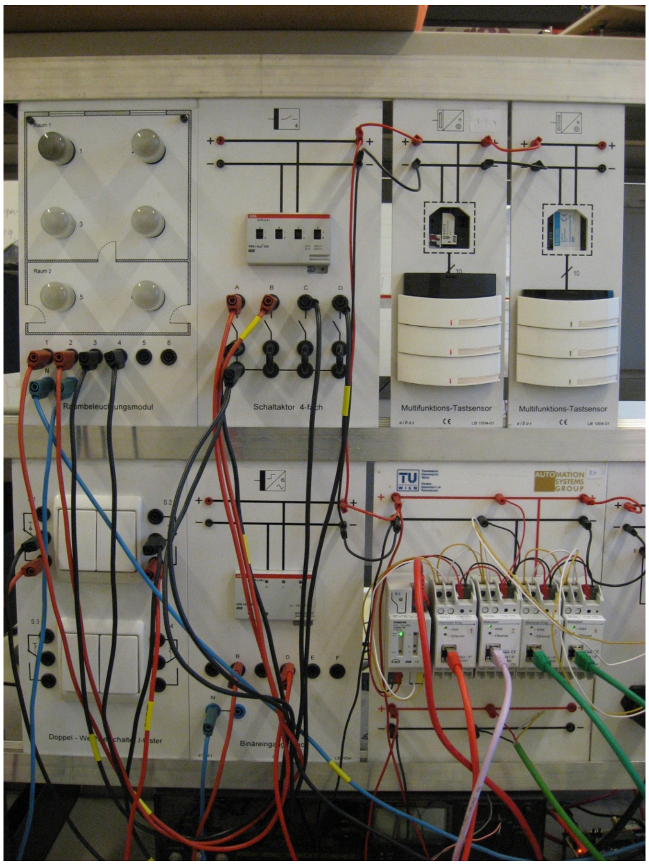
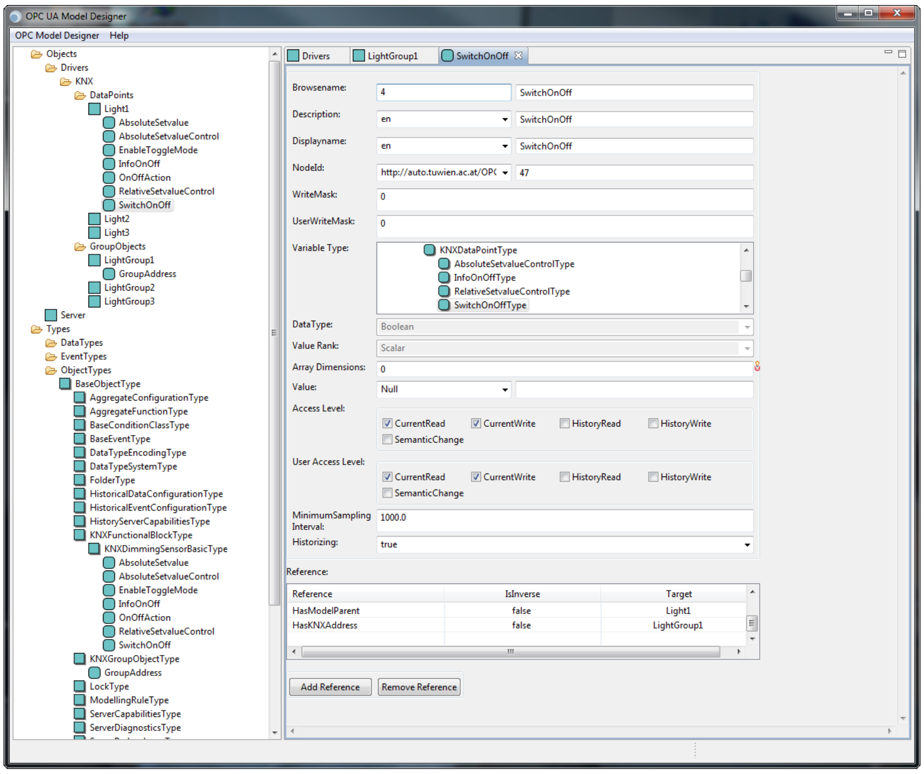


Figure 8: Laboratory environment

The reference type HasKNXAddress, a subtype of NonHierarchicalReferences, is one of the most important types. It connects a data point of a functional block with a GroupObject (KNXGroupObjectType) and its group address (KNXGroupAddressType) and therefore represents the KNX multicast communication model. Furthermore, there are three new subtypes of HierarchicalReferences called HasInputDatapoint, HasOutputDatapoint and HasParameter, which are used to connect the different types of data types (Input, Output, Parameter) with a functional block.

The new abstract variable types, KNXAddressType and KNXDataPointType, are both subtypes of the BaseDataVariableType. They are, in turn, supertypes of other non-abstract variable types like the KNXGroupAddressType mentioned above and some KNX data point types like SwitchOnOffType, InfoOnOffType, etc.. As object type, the model introduces the KNXFunctionalBlockType, which shall be the supertype of all KNX functional blocks, and the KNXGroupObjectType. The only modelled functional block for this work shall be the KNXDimmingSensorBasicType, which allows to switch on and off a light and view its current status. The dimming



Browsename:	4	SwitchOnOff
Description:	en	SwitchOnOff
Displayname:	en	SwitchOnOff
NodeId:	http://auto.tuwien.ac.at/OP(	47
WriteMask:	0	
UserWriteMask:	0	
Variable Type:	SwitchOnOffType	
DataType:	Boolean	
Value Rank:	Scalar	
Array Dimensions:	0	
Value:	Null	
Access Level:	[✓ CurrentRead] [✓ CurrentWrite] HistoryRead HistoryWrite SemanticChange	
User Access Level:	[✓ CurrentRead] [✓ CurrentWrite] HistoryRead HistoryWrite SemanticChange	
MinimumSampling Interval:	1000.0	
Historizing:	true	

Reference:

Reference	IsInverse	Target
HasModelParent	false	Light1
HasKNXAddress	false	LightGroup1

Figure 9: HB-Softsolution Model Designer

functionality was not implemented.

Figure 11 shows a specific example for how to use the newly introduced object, variable and reference types.

4.4 HB-Softsolution OPC UA Driver Framework

As mentioned above, the Comet OPC UA Driver Framework, developed by HB-Softsolution, was used as base for the OPC UA Server and its KNX driver. The framework provides an OPC UA Server instance which loads included driver JAR packages from the classpath.

4.4.1 Framework structure

With the help of a wizard for the IDE Eclipse which allows the creation of all needed Java files in just a few clicks, the user can easily start with a new Comet OPC UA Server instance and an empty driver project. The wizard generates two Eclipse projects, one for the OPC UA Server and one for the driver package.

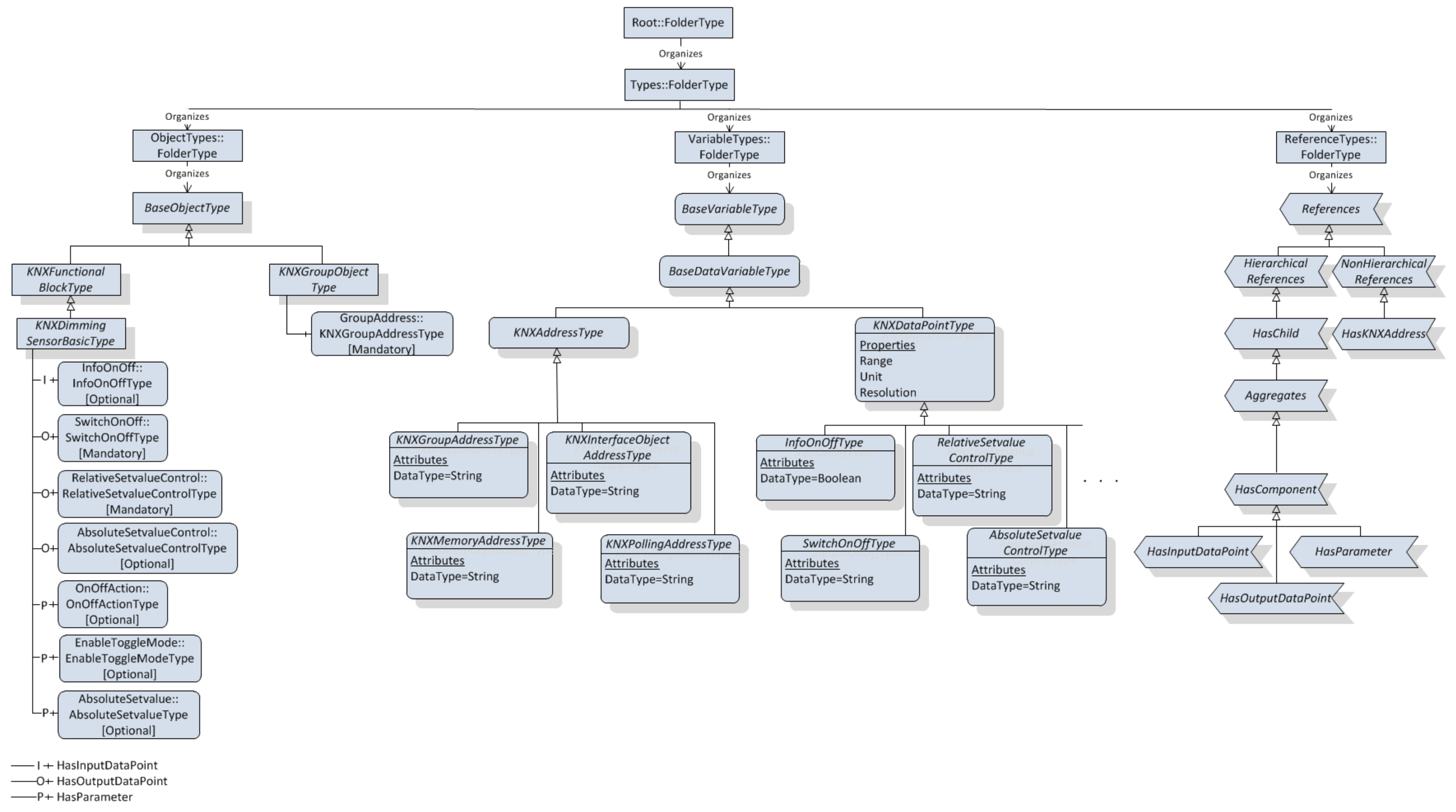
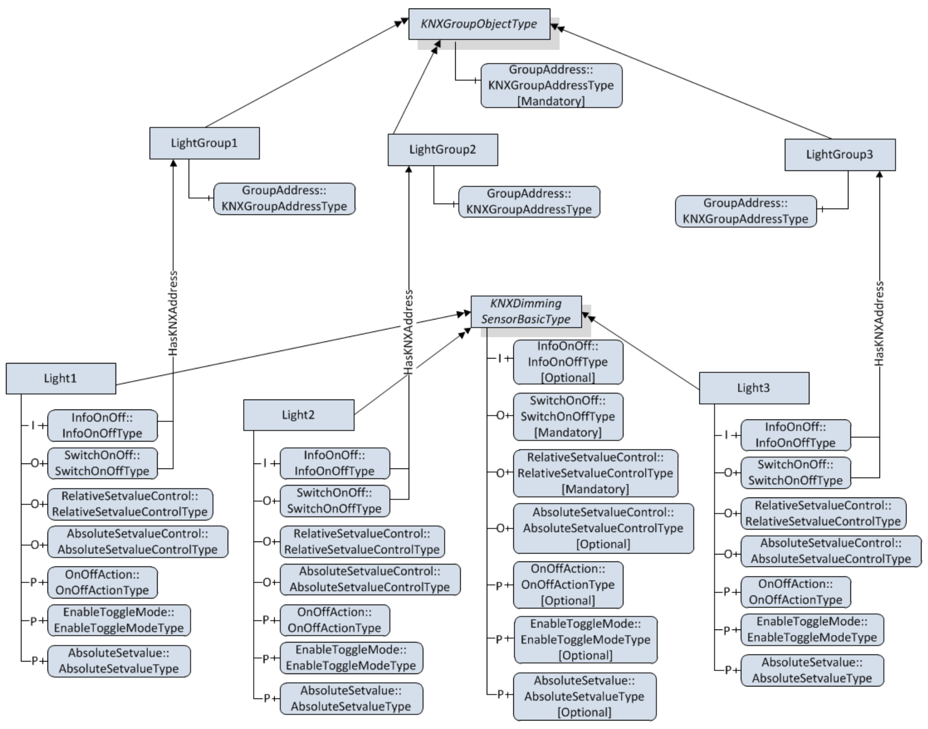

Root::FolderType
Organizes
Types::FolderType
Organizes
ObjectTypes:: FolderType
BaseObjectType
KNXFunctional BlockType
KNXDimming SensorBasicType
KNXGroupObject Type
GroupAddress:: KNXGroupAddressType [Mandatory]
I+ InfoOnOff:: InfoOnOffType [Optional]
O+ SwitchOnOff:: SwitchOnOffType [Mandatory]
O+ RelativeSetvalueControl:: RelativeSetvalueControlType [Mandatory]
O+ AbsoluteSetvalueControl:: AbsoluteSetvalueControlType [Optional]
P+ OnOffAction:: OnOffActionType [Optional]
P+ EnableToggleMode:: EnableToggleModeType [Optional]
P+ AbsoluteSetvalue:: AbsoluteSetvalueType [Optional]
I+ HasInputDataPoint
O+ HasOutputDataPoint
P+ HasParameter
VariableTypes:: FolderType
BaseVariableType
BaseDataVariableType
KNXAddressType
KNXGroupAddressType Attributes DataType=String
KNXInterfaceObject AddressType Attributes DataType=String
KNXMemoryAddressType Attributes DataType=String
KNXPollingAddressType Attributes DataType=String
KNXDataPointType Properties Range Unit Resolution
InfoOnOffType Attributes DataType=Boolean
RelativeSetvalue ControlType Attributes DataType=String
SwitchOnOffType Attributes DataType=String
AbsoluteSetvalue ControlType Attributes DataType=String
. . .
ReferenceTypes:: FolderType
References
Hierarchical References
NonHierarchical References
HasChild
HasKNXAddress
Aggregates
HasComponent
HasInputDataPoint
HasParameter
HasOutputDataPoint

Figure 10: KNX information model for OPC UA



KNXGroupObjectType
GroupAddress:: KNXGroupAddressType [Mandatory]
LightGroup1
LightGroup2
LightGroup3
GroupAddress:: KNXGroupAddressType
HasKNXAddress
KNXDimming SensorBasicType
Light1
Light2
Light3
InfoOnOff:: InfoOnOffType
SwitchOnOff:: SwitchOnOffType
RelativeSetvalueControl:: RelativeSetvalueControlType
AbsoluteSetvalueControl:: AbsoluteSetvalueControlType
OnOffAction:: OnOffActionType
EnableToggleMode:: EnableToggleModeType
AbsoluteSetvalue:: AbsoluteSetvalueType
[Optional]
[Mandatory]

Figure 11: Model example

4.4.2 Server application

Figure 12 depicts the folder and file structure of the server project. The server application contains only one Java file which represents the OPC UA Server instance and therefore is the main class used to launch the server.

The other folders and files have the following purpose:

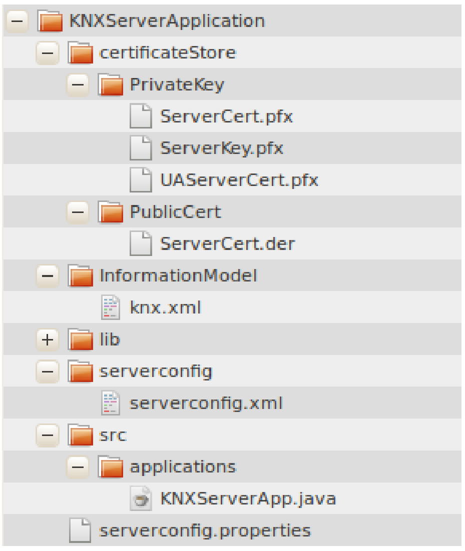
certificateStore stores all the public and private keys and certificates which are used for authentification.

InformationModel stores the information models for this server instance in XML format. The KNX Server discussed in this work will only load the information model depicted in Figure 11, which is stored in the file knx.xml.

lib contains all the necessary additional libraries, for instance, the HB-Softsolution OPC Server SDK or Calimero, the KNX Java library.

serverconfig contains a file named serverconfig.xml, which stores all server specific information like the server's address, its name, available security policies, etc..

serverconfig.properties stores the IP address of the computer that runs the server. This is necessary to successfully connect to the KNX router. If this file does



```
− KNXServerApplication
    − certificateStore
        − PrivateKey
              ServerCert.pfx
              ServerKey.pfx
              UAServerCert.pfx
        − PublicCert
              ServerCert.der
    − InformationModel
          knx.xml
    + lib
    − serverconfig
          serverconfig.xml
    − src
        − applications
              KNXServerApp.java
      serverconfig.properties
```

Figure 12: Server application structure

not exist or is empty, the user has to manually enter the IP on the server's startup.

4.4.3 Integrating a custom driver

A Comet driver project has the following structure:

lib contains all the libraries needed for the specific driver. In this case, only the Calimero Java library, the OPC UA Java stack and some logging utilities are needed.

src contains two Java files: the interface ICometDriverConnection, already implemented by HB-Softsolution, provides useful methods to work with nodes, which will be described above. The class CometDRVManager contains the implementation of the read, write and subscribe methods and will be discussed in the next section.

Integrating a custom driver can be easily done by just packing the driver project into a JAR and then adding this JAR to the server's classpath.

4.5 KNX driver implementation

As already mentioned, all communication between OPC UA and KNX takes place in the Java class CometDRVManager, which is depicted in Figure 13. This class then gets accessed from the server, which executes methods to read, write and subscribe nodes of the KNX information model. These methods are described in the following:

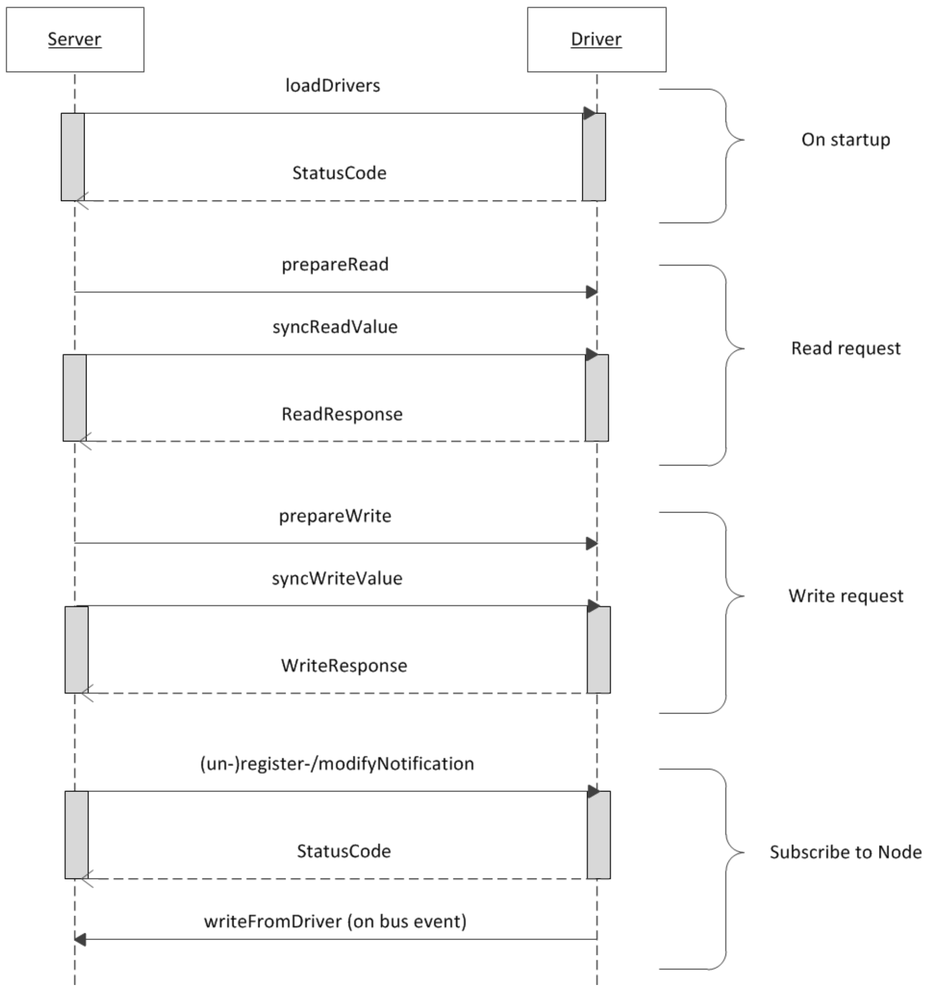


Figure 13: Communication between Server and Driver

```
public StatusCode loadDrivers()
```

This method gets called during the server startup. It stores the namespace index for the KNX driver and initializes the connection to the KNX network. To be able to do this via Calimero, it is necessary to know the computer's IP address. For this reason, the already mentioned serverconfig.properties file was created. Without this file the user has to manually enter the IP address.

```
public ReadResponse syncReadValue(NodeId nodeId, long senderState)
```

This method gets called at every read request of a client. As not all nodes are of interest to every driver, it comes to decide if the node belongs to the driver's information model. For KNX, this can be done by trying to find the node's group address since Calimero needs to know the corresponding group address to read a value.

```
public WriteResponse syncWriteValue(NodeId nodeId, DataValue value,
    long senderState)
```

Analogous to the read method, the write method also needs to know, if the node to be written belongs to the driver's sphere first. If so, the driver again uses Calimero to write a value to the corresponding group address.

```
public WriteResponse asyncWriteValue(NodeId nodeId, DataValue value
    , long senderState)
public ReadResponse asyncReadValue(NodeId nodeId, long senderState)
```

Besides the synchronous read and write methods, there are also their asynchronous counterparts.

```
public void prepareRead(NodeId nodeId)
public void prepareWrite(NodeId nodeId)
```

Before each read and write request, one of these methods gets called. When the desired node is ready to be read or written, a flag is set, which is the prerequisite for the read and write method to be called.

```
public StatusCode registerNotification(Node node,
    MonitoredItemCreateRequest itemToCreate)
public StatusCode unregisterNotification(NodeId nodeId)
public StatusCode modifyNotification(NodeId nodeId,
    MonitoredItemModifyRequest itemToModify)
```

These three methods are responsible for OPC UA subscriptions. As their names already imply, they allow to register, unregister and modify a subscription/notification. On every registration of a node, its node id is stored in a Java hash map. This map maps a group address to a list of nodes which have data points connected to this group address. The KNX driver uses a Calimero LinkListener to listen to changes on the KNX bus. On every change at a KNX group object, a method is called, which processes the change. This means, all nodes mapped to the group address are updated.

```
private String getGroupAddressValue(Node group)
private List<GroupAddress> getGroupAddress(NodeId nodeId) throws
    KNXFormatException
```

The last two methods are private helper methods. The first expects a node as parameter, which has a KNXGroupAddressType connected to it (see Figure 11) and returns the value of the GroupAddress variable. The latter calls the first method and converts the group address from String to Calimero objects.

5 Conclusion and Outlook

This paper showed how an information model for a KNX network can be created with the help of the possibilities given by OPC UA. Furthermore, it presented the implementation of a KNX driver for the Comet Driver Framework of HB-Softsolution, with the help of which it is possible to receive read-, write- and subscribe commands from OPC UA clients and further automatically convert these into KNX commands.

OPC UA, compared to OPC DA, offers many new features like platform independence and strong information modelling capabilities. But it will take some time until these features will be in wide use. In the meantime, the OPC Foundation offers a OPC DA wrapper that maps Classic OPC DA data to OPC UA base types and, in this way, ensures backward compatibility and eases the change to the new technology.

The data model used in this paper contains three lights which were modelled as KNX functional blocks of the type DimmingSensorBasic. In order to model a whole building, other KNX functional blocks such as light dimming actuators, sunblind sensors/actuators, HVAC sensors/actuators, etc. could be modelled as OPC UA objects. Besides KNX, other protocols like ZigBee [9], BACnet [10][11] or LonWorks [12] can be controlled via separate drivers[1]. Thus, various protocols could be operated platform independently and with only one management tool in a building.

[1]The implementation of a BACnet driver can be found in [13].

References

[1] OPC Unified Architecture Specification. OPC Foundation, 2009.

[2] KNX Specification Version 2.0. Konnex Association, 2009.

[3] Information technology - Home Electronic Systems (HES) Architecture. ISO/IEC 14543-3, 2006.

[4] KNX Association. Interworking. `http://www.knx.org/knx-standard/interworking/`, November 2011.

[5] Wolfgang Mahnke, Stefan-Helmut Leitner, and Matthias Damm. *OPC Unified Architecture*. Springer, 2009.

[6] Hidehiko Wada Katsuji Usami, Shin-ichi Sunaga. A Prototype Embedded XML-DA Server and its Evaluations. Technical report, Yokogawa Electric Corporation, October 2006.

[7] HB-Softsolution. `http://www.hb-softsolution.com/`, November 2011.

[8] Wolfgang Granzer, Wolfgang Kastner, and Paul Furtak. KNX and OPC UA. In *Konnex Scientific Conference*, November 2010.

[9] Zigbee specification 2007, 2007.

[10] BACnet – a data communication protocol for building automation and control networks. ANSI/ASHRAE 135, 2008.

[11] Building automation and control systems (BACS) – part 5: Data communication protocol. ISO 16484-5, 2010.

[12] Control Network Protocol Specification. ANSI/EIA/CEA 709 Rev. B, 2002.

[13] Andreas Fernbach, Wolfgang Granzer, and Wolfgang Kastner. Interoperability at the Management Level of Building Automation Systems: A Case Study for BACnet and OPC UA. In *Proc. of 16th IEEE Conference on Emerging Technologies and Factory Automation (ETFA '11)*, September 2011.