smartHomeControl

Towards ZigBee Home Automation – a proof of concept

Seminar/Project

Master's Programe: Telematics
Faculty of Computer Science
Graz University of Technology

Tim Genewein, BSc.

27.03.2011

Supervisor: Dipl.-Ing. Dr.techn. Steinbauer Gerald
Institute: Institute for Software Technology

External Supervisor: Dipl.-Ing. Neureiter Christian
Company: NTE Systems

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
Date (signature)

Preface

This work is the result of a project (more precisely: “Seminar/Project”), conducted during the winter semester 2010 in close cooperation with *NTE Systems*. Thus, the project has a strong focus on practical aspects and the implementation of a proof of concept. This documentation serves two major purposes:

- Giving an overview over the key concepts and an introduction to the most important methods necessary for working with ZigBee. The work tries to highlight important parts of the vast standards, but also to give some intuitive interpretations or practical examples of the abstract concepts – mostly from a programmer’s perspective.
- Providing an informative documentation of the implementation. Its main purpose is to help developers setting foot in this domain and steepening their learning curve but also giving enough detail to be able to work with the implemented software. A more technical documentation of the software can be generated from the in-code documentation.

I would especially like to thank Mr. Dipl.-Ing. Georg Stasny, director of NTE Systems, for making this project possible. The integration into the development team, getting to know the work in a professional environment and the possibility to actively express personal ideas and opinions has been very valuable. Especially, because these skills are hard to teach and can only be learned through constant practice in a fruitful environment.

Further, I want to thank Mr. Dipl.-Ing. Christian Neureiter, head of the embedded development team. He has always been a contact person for technical, organizational, but also personal questions.

My supervisor at the university was Dipl.-Ing. Dr.techn. Gerald Steinbauer. I would like to thank him and the Institute for Software Technology for the pragmatic and unbureaucratic supervision.

Tim Genewein

Graz, March 2011

Abstract

The company *NTE Systems* has developed a flexible, programmable controller, targeted at the solar water heating domain. Along with NTE's SCADA software for monitoring and telecontrol, the controller hardware can be used for a variety of applications. The system provides an interface that can easily be used without in-depth expert knowledge. Therefore, it seems well-suited for the consumer market.

In order to be compatible with the upcoming smart grid technology or wireless home automation products, a ***ZigBee*** interface for the existing hardware is needed.

Since NTE has had no prior experience with the ZigBee technology, this project was conducted as a proof of concept (POC), to gain some experience and know-how on using ZigBee. In a parallel hardware-POC, four devices have been developed:

- smartLight – a light that can be wirelessly switched on and off
- smartOutlet – an electrical outlet that can be turned on and off wirelessly; and is also able to measure the current power consumption and report these measurements
- smartSwitch – a wireless switch, which can operate the smartLight or the smartOutlet
- coordinator – a central device for setting up and managing a ZigBee network

These hardware developments have been completed with a software integration. It mainly consists of an embedded C# application using the .NET Micro Framework and a corresponding graphical user interface (GUI) on a PC. These two applications communicate via a web service interface. The GUI provides an overview of the ZigBee network nodes and their functionality. It can be used to switch the smartLight and the smartOutlet on and off and read their current switching state. Additionally, the measurement readings of the smartOutlet can be displayed in a chart.

In the course of this project, the software POC has been functionally completed. The result is a nice prototype system that gives a technological preview on some of the possibilities of the ZigBee technology. The basis of this system is a library that encapsulates the ZigBee communication, which essentially consists of exchanging byte arrays. This library is used on both, the embedded device as well as the desktop GUI.

The results of this project form the basis for a system integration of ZigBee technology into the existing controller hardware and SCADA software. The implemented library has been designed and implemented thoroughly and can be re-used and extended in consecutive project stages.

This POC has also helped in gaining practical know-how and ZigBee experience. It has cleared uncertainties and shown strengths and weaknesses of ZigBee. With this knowledge an upcoming architectural phase of the system integration can now be tackled.

The ZigBee technology with its vast specifications has proven to be very flexible and adaptable to a broad range of application scenarios. It is not very restrictive as far as design patterns or communication paradigms are concerned as long as the basic design principle "*low-power, low data-rate*" is kept in mind. On the other hand, there are several very specific *public application profile* definitions, which define ZigBee devices, their functionalities and properties for the corresponding application domain (such as home automation or smart grid). These definitions are exhaustive and ensure interoperability and compatibility.

Table of Contents

1 Introduction

1.1 Motivation

Household appliances and electrical installations have not changed much over the last few decades. In the earlier days, people would fantasize about the 21st century with flying cars, household robots and washing machines or dishwashers that “all the work by themselves”. Looking at light switches, for instance; not much has changed during the last 50 years – they still work the same way and they even look more or less the same way.

On the other hand, we have an arsenal of enabling technologies, which allow us to reinvent electrical and household appliances. With cheap processors and power-saving devices, it is no longer a matter of cost, to have a device being connected to other devices or even to the internet. The *Internet Of Things* is becoming a reality – interconnected devices are communicating with each other to augment their functionality. Smartphones keep us connected, allow us to interact with devices on the internet and with their displays and input-capabilities also provide a rich user-interface for a broad range of applications.

For industrial applications, the automation of new buildings has already become de facto standard. During the last years, *Home Automation* products have emerged – yet these technologies are still far from their full potential. Most current products lack some form of standardization, i.e. products of different manufacturers are incompatible. Moreover, most technologies rely on a hardwired bus-installation and can neither be set-up nor configured by a layman. This makes current products expensive and inflexible.

The most prominent technology to overcome these deficiencies is ***ZigBee*** – it bases on a strong standard and wireless-communication. ZigBee has been designed to support a broad range of building-automation applications and therefore has the following properties (by design):

> *ZigBee is inexpensive – it has no license fees and does not require special hardware (the radio-transmitters use a commonly used band).*
>
> *ZigBee is wireless and low-power – special care has been taken, to allow devices that can last on a battery power supply for years. E.g. a battery-powered light-switch that can simply be glued onto a wall, without the need of any wires.*
>
> *ZigBee runs on low computational resources – by design, ZigBee uses low data rates. This might seem like a severe constraint, but it ensures that ZigBee can be implemented on processors and micro controllers with very limited resources. Furthermore, this helps reducing the power-consumption which is crucial for battery powered devices.*

These properties make ZigBee an excellent choice for home automation. Manufacturers are starting to release ZigBee compatible household and electrical appliances. In the U.S. and some European countries, the emerging *Smart Grid* initiative has chosen ZigBee as the wireless technology. In the smart grid, every household has a smart electric meter, which is able to query the current rates and communicate via ZigBee with household appliances. This could be used to have the smart meter start your dishwasher when the rates are low. It also helps energy providers to reduce peak-demands.

1.2 Project Environment

The company **NET Systems** has developed a programmable controller for solar water heating systems. The controller can be programmed using a graphical language and is capable of reading data from sensors as well as driving actuators, both with various hardware-interfaces (analog/digital voltage, PWM, current-controlled …). Furthermore, the company has developed a data acquisition software, which is able to collect data and visualize it in reports or charts over the internet. The software also allows telecontrol of the remote plant.

To enhance the capabilities of the controller and to set foot into the home automation and smart energy domain, the only missing component was an interface for communicating with state of the art household appliances. The most reasonable choice was using *ZigBee* for this communication-interface. Since the technology was new for the company, it has been decided to conduct a *proof of concept (POC)* project, in order to get to know the characteristics of ZigBee, clarify uncertainties and reveal possible problems for later system integration.

As a proof of concept, it was never intended to produce production-ready solutions or even parts of it. The focus has clearly been set on fast development of certain functional features with a special interest on strengths and weaknesses of the taken approaches and possible inherent problems with the technology.
The project's functional requirements can be found in 3.1.2.

1.3 Structure of this document

Chapter 2 will give an overview and short introduction to the technologies used in the project. It starts with a more detailed look on *ZigBee*, especially from a programmer's perspective, and then continues with an overview over *Texas Instruments' Monitor & Test* Interface for ZigBee transceivers. The chapter ends with an introduction to the *.NET Micro Framework* and *Web services.*

Chapter 3 will present the details of the project. After a system-overview and the presentation of the goals of the project, the chapter continues with a detailed look on the implementation. After an overview over the (Visual Studio) solution – the most significant parts will be explained in detail.

Chapter 4 summarizes the project's findings, points out known issues with the implementation and presents the most important lessons learned in the course of the project. It further gives an outline over the next steps towards a system integration of ZigBee. The chapter concludes with a more subjective résumé.

The Appendix A completes this document with a very practical guideline on how to extend the implementation as well as a detailed description of the web service interface and a hands-on guide on how to use the implemented ZigBee Control Center GUI.

At the very end of the document you can find the Bibliography.

2 Fundamentals

2.1 ZigBee

2.1.1 What is ZigBee?

ZigBee™ is a wireless network standard for so called "*short range wireless networks*", more commonly known as *wireless personal area networks* (WPAN). One usually speaks of a WPAN when having communication ranges from a few meters up to one- or two-hundred meters. ZigBee is based on the popular **IEEE 802.15.4** standard, which is very common for wireless sensor network applications. Most ZigBee devices use the license-free 2.45 GHz ISM-band that is also used by Bluetooth™ and WiFi™.

In order to achieve its high robustness, ZigBee devices will scan all sub-channels of the 2.45 GHz band, to determine the channel with the lowest noise. Furthermore, ZigBee exploits *Offset-Quadrature Phase-Shift Keying (O-QPSK)* and *Direct Sequence Spread Spectrum (DSSS)* as defined in the IEEE 802.15.4 – both technologies are well suited for low-signal-to-noise ratio environments. ZigBee uses *CSMA-CA* for accessing a channel, and a *16-bit CRC* on each packet. If a packet gets lost or corrupted, it will be re-sent up to three times.

From a topological point of view, ZigBee achieves a high reliability by *Mesh Networking* – which means that network nodes will communicate with all neighboring nodes within range. If a node tries to communicate with a distant outside of its range, the packet will be forwarded via intermediate nodes – if an existing route fails to work (because nodes have left the network or moved to another location), a new (multi-hop) route will be established.

Among the great number of wireless network standards, ZigBee has very specific design-principles, which give rise to some very characteristic application scenarios such as *Home Automation*.

> *ZigBee is low-power and low-data-rate!*

Typical ZigBee devices do not require a large bandwidth – imagine a light switch that gets pressed a few times throughout the day. The bandwidth required to propagate this information is very small. Other ZigBee devices might measure certain values (imagine a wireless thermostat) and report them cyclically. For most home automation applications, it should be sufficient to have a very coarse temporal resolution; sending the current temperature once every ten minutes should be enough.

By introducing the constraint to have low-bandwidth devices, ZigBee becomes the best-suited choice for many usage scenarios. For instance, battery-powered devices implicitly require a low-data-rate design. A lot of ZigBee devices will only have small computational resources (some kind of micro-controller with limited memory and clock frequency). Again, for battery-powered devices, it is an implicit design constraint to use little processing power.

ZigBee also aims at networks with a huge number (say a few dozen up to a few hundred) of network-nodes. If the design-principle of all nodes is not to use too much bandwidth, even large networks become computationally feasible for devices with "weak" processors.

With the properties above, ZigBee is very well suited for inexpensive devices (because hardware-resources can be kept low), which is crucial for application scenarios such as home automation, where consumers are not willing to spend several hundred dollars on a single light-switch.

> *ZigBee is an open global standard.*
>
> *This standard has been developed by the* ***ZigBee Alliance****, a consortium of more than 230 companies, working on the introduction and spread of their global standards.*

The ZigBee specifications, as well as the IEEE 802.15.4 specs, are accessible without any charges. The ZigBee Alliance ensures that the technological progress is driven by the interest of all of its members and not just by a handful of the most influential corporations. One will not end up, being tied to a single manufacturer or having to deal with compatibility issues, since the specifications can be used as a basis for all developments.

ZigBee fills the gap between high-data-rate wireless networks, such as WiFi or cellular networks and short-range technology such as infrared, NFC or RFID. A broad range of usage scenarios of the same type, such as home automation or smart-grid applications, require exactly a technology like ZigBee. As a developer, you can rely on the vast specifications and do not have to worry about compatibility or interoperability. Also, ZigBee is cost-effective, since it requires no special hardware and it can be used free-of any charges. The given information is just an overview – for a more detailed discussion, see the sources [1] and [2].

2.1.2 Example Scenario

The following scenario describes the devices that have been developed in a parallel project at *NTE Systems*. All implementations of the software-project have been designed for these hardware-prototypes. This scenario and the devices involved will be referenced throughout the whole document. A detailed documentation of the hardware POC project can be found in [3].

In order to have a non-trivial and functional network, *NTE Systems* has decided to develop hardware-prototypes of the following devices (a picture of the devices can be seen in Figure 1).

Coordinator:

Every ZigBee network contains one (and only one) *coordinator*-node. This node initializes and starts up the network (it scans all channels for traffic and then chooses one channel to start the network). It also allows other nodes to join the network.

In the given setup, the coordinator is equipped with an interface for communicating with the GUI software that displays a graphical representation of the network and provides means to interact with the network-nodes. In this case the coordinator acts as a gateway to the ZigBee network.

smartLight:

The smartLight consists of an energy-saving lamp, coupled with a ZigBee device. The lamp can wirelessly be switched on and off. Furthermore, the smartSwitch (see below) can be used to operate the light, after an appropriate *binding* has been set up. Since the smartLight is main-powered, it will also act as a ZigBee router (more on routers and bindings can be found in 2.1.3).

smartSwitch:

The smartSwitch is a battery-powered double-rocker switch with ZigBee connectivity. Each rocker can be used to operate a different device (like the smartLight) or a group of devices. Using a built-in ZigBee mechanism, it is also possible to find all “switchable” devices in the network and automatically set up the switch to operate all of these devices (main-switch functionality).

smartOutlet:

The fourth hardware-prototype consists of a power outlet and a power meter, coupled with a ZigBee device. The power meter is able to sense the current power consumption at the outlet and is configured to transmit these measurements every two seconds to the coordinator-node (actually it transmits the load-current, mains-voltage, effective power and the cumulated energy consumption). Also the outlet, and any connected load, can be switched on and off – similar to the smartLight. The smartOutlet is main-powered and acts as a ZigBee router.

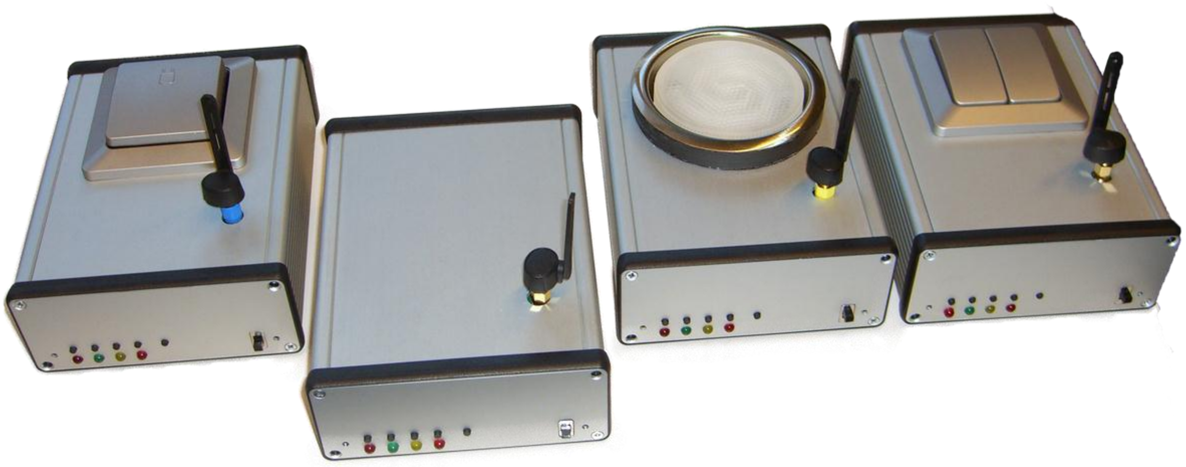


Figure 1 - ZigBee devices of the Hardware POC

One possible use-case of the described devices looks like this:

The coordinator has set up the network and the three remaining devices have joined it. The smartSwitch has been configured to control the smartOutlet on one rocker and the smartLight on the other one. All of this can be set up by using push-buttons on the hardware-prototypes and of course the built-in ZigBee functionality; no additional software tool is necessary.

Some kind of electrical device (e.g. a temperature controlled fan) is plugged into the smartOutlet. The outlet measures the current load and reports these data periodically to the coordinator. Using the software, developed in this project, the user is able to visualize all network nodes and retrieve information on the device type and the functions of the device (can it be switched on and off, does it report measurement values?). Switchable devices can be operated from the GUI and the reported measurement values can be displayed on the GUI.

2.1.3 Endpoints, Cluster, Profiles – Terminology

This section gives an introduction of some of the key concepts of ZigBee and the corresponding terminology. It is intentionally kept brief to give a quick overview without too many details. A more extensive discussion on the following terminology and principles can be found in [1] or [2] (in German) – for a very practical point of view, consider [4].

2.1.3.1 ZigBee Network

The simplest device-type of a ZigBee network is the *end-device*. It is usually battery-powered and thus might be idle or “sleeping” for quite long periods. Therefore it might not be able to immediately keep track of the changes in the network. Due to that, it has to be connected to a router-device, which will store pending messages for associated end-devices until these devices wake up. The maximum sleep-time of an end-device depends on the application and is specified in the corresponding profile.

The second device-type found in a ZigBee network is the *router*. A main-powered device is usually configured to be a ZigBee router. It has the responsibility to keep track of all devices associated and store pending messages for sleeping end-devices. It will also build a “routing table” that plays the central role for the transmission of multi-hop messages. If the device notices that a certain route fails repeatedly, it will try to establish a new route. A coordinator also acts as a router.

As already mentioned, every ZigBee network contains one and only one *coordinator*. The coordinator sets up the network (called *PAN: Personal Area Network* or sometimes also *HAN: Home Area Network*) and assigns a PAN-ID. Note that several PANs can coexist on the same channel, but each PAN needs a separate coordinator and PAN-ID. However, inter-PAN communication between two different coordinators is possible. In that case, one coordinator becomes a router-device in the network of the other coordinator.

> *A ZigBee network is identified by a* ***PAN-ID*** *and contains a single* ***coordinator****. Other device types are* ***routers*** *(main-powered) and* ***end-devices*** *(battery-powered).*
>
> *End-devices may "sleep" and therefore be unable to immediately react – due to that, they have to be associated to a router-device that keeps track of changes in the network and stores pending messages for associated end-devices.*
>
> *To communicate with devices that are not within (direct) communication range, ZigBee uses multi-hop messaging (in a mesh-topology network). Every router-device builds and stores a routing-table.*

2.1.3.2 ZigBee Node

Every physical device in a ZigBee network, regardless of its device type, is a *node*. To uniquely identify a node, the so called *IEEE-Address* (or often *MAC-Address*) is used. This 64-bit address is globally unique, which means that there are (or at least should be) no two devices "in the world" with the same MAC-address. These addresses are administered by the IEEE: the upper 24 bits of the address are the so called *Organizationally Unique Identifier (OUI)* – a company can obtain one from the IEEE (requires a one-time fee). It then becomes your company's responsibility to ensure that the other 40 bits of the address are unique for every device released into public.

When a node joins a ZigBee network, a 16-bit address, the so called *Network-Address*, will be assigned to the device. This address is unique *within the current network* and is used to send, route and deliver messages. Note that this address **might change**, if the device changes its parents (e.g. a mobile end-device moving to another location within the network, so that its original parent is no longer within communication range). However, since the MAC-address of a device is unique, all routing-tables will be updated correspondingly.

Programmer's hint: When modeling a ZigBee node, do not (!) assume that the network address is fixed – the case of a network node, changing its network address can easily be overseen when designing test-environments. Another side note: most development kits for ZigBee devices cannot provide a globally unique IEEE-address. To force you to use your own addresses (anything else would not be too smart anyway), they will randomly generate an address on every reboot of the device which effectively means that the same physical device will have a different MAC-address after rebooting.

The ZigBee coordinator has always (by definition) the network address 0x0000. Together with a few reserved addresses (broadcast, groupcast …) this would make about 65000 nodes addressable in a single network. Note that this is only a theoretical figure – a more realistic number is maybe several hundred nodes; of course this strongly depends on the bandwidth-use of these nodes.

> *Every ZigBee network-node has a unique address within the network, the so called **network address** (16-bit) that is assigned during the network-join by the parent of the device. In some cases this address can change during operation. The coordinator-node always has the network address 0x0000.*
>
> *Every ZigBee node also has a globally unique 64-bit MAC-Address (or IEEE-address). When releasing devices into public, the manufacturer has to ensure the unambiguity of these addresses. The upper 24 bit of that address is the so called OUI which is a unique identifier for a specific organization – an OUI has to be obtained from the IEEE (with a one-time fee).*

2.1.3.3 Endpoints

Every physical ZigBee device may host ("contain") up to 240 logical devices, called endpoints. An example would be the smartSwitch, which is one physical device, i.e. one network-node. But it contains two separate rocker switches, where each of the switches is a separate logical device or in ZigBee terminology: the smartSwitch node contains two (OnOff) Switch-endpoints. Another example is the smartOutlet that contains a (switchable) outlet and a power-measurement device.

An endpoint is described by a one-byte identifier – the *Endpoint-ID*. This identifier is used to address within a node, i.e. when trying to send a message to a specific endpoint, one needs to specify the network address of the node as well as the endpoint-ID within the node.

Every endpoint is described by two more identifiers (both 16-bit): the *Profile-* and the *Device-ID*. ZigBee profiles will be covered in 2.1.3.5; a profile defines the application domain of the device and device-IDs are defined in the corresponding profile.

Finally, every endpoint has two lists of so called clusters; an *In-Cluster-* and an *Out-Cluster-List*. They provide a description about the devices capabilities – e.g. if one device has an OnOff-cluster in its out-cluster-list, it can act as a switch; if another device has an OnOff-cluster in its in-cluster-list, it can be switched on and off. Clusters are also defined in a ZigBee profile and will be covered in 2.1.3.4.

The information describing an endpoint can be retrieved via ZigBee and is encapsulated in the so called *Simple Descriptor*.

> *A physical ZigBee node can contain several logical devices, called endpoints, which are specified by a one-byte identifier: the **endpoint-ID**.*
>
> *An endpoint also contains information about its device-type and functionality via the profile- and device-ID, as well as the in- and out-cluster-lists.*

2.1.3.4 Clusters

ZigBee clusters are defined by a 16-bit identifier and describe the capabilities of an endpoint. Clusters are also well specified in ZigBee profiles, i.e. cluster-ID 0x06 in the home automation profile could mean something different than cluster-ID 0x06 in the smart energy profile (in this case they are actually the same because cluster-ID 0x06 – OnOff - is defined in the ZCL).

The device capabilities are described via attributes and commands – think of a cluster as an interface that defines methods and properties of an object. The properties are the attributes, which you can read or in some cases also write; and methods are the commands which can be invoked to execute certain routines. The concept should be quite familiar and becomes clear when looking at specific examples:

*One of the simplest and most used clusters is the **OnOff-Cluster**. It describes the functionality to turn something on or off. The ZCL definition also states clearly that every OnOff-cluster knows a single command – the OnOff message which has only one parameter, specifying*

whether to turn the device on or off or toggle its state. Furthermore, every OnOff-cluster has a single attribute which gives information about the current switching status.
Now, an endpoint is a logical device which "implements" this interface – for instance an ***OnOff-Switch*** *must contain an OnOff-cluster in its out-cluster-list, so it can produce OnOff-commands. An endpoint like the DimmableLight on the other hand, has an OnOff-cluster in its in-cluster-list, so it can process incoming OnOff-commands.*
Of course, an endpoint can implement multiple interfaces, thus the in- and out-cluster-LISTS.
When binding two devices together (the switch with the light), the corresponding in- and out-cluster-lists must match.

The ZigBee Alliance has published a set of general-purpose-clusters and commands to interact with these clusters in the so called *ZigBee Cluster Library (ZCL)*. The ZCL is the basis for all public profiles, e.g. the OnOff-cluster is also part of it. The ZCL also specifies a few very general commands, so called *cross-cluster-commands*, which can be used on many clusters. For instance, the command to read an attribute with a given identifier can be used on every ZigBee cluster – another example is the configuration of attribute reporting. Source: ZCL specifications [5].

2.1.3.5 Profiles

Every message on a ZigBee network is sent in the context of an *Application Profile*. There are public and private profiles; public profiles have been developed by the ZigBee Alliance and ensure standardization and interoperability for certain domains. They are all based on the ZigBee cluster library (ZCL). Private profiles can be defined by a manufacturer to satisfy his individual needs (for some applications, it might be desirable to have very simple profiles and one can relinquish on most of ZigBees more powerful mechanisms – in that case it makes sense to define a private profile; however, these profiles are usually intended to be used company-internal only).

The public profiles are a comprehensive extension to the "basic" ZigBee specifications. They define a broad range of device types and clusters. And of course, they also define attributes and commands for each cluster. The ZCL and the profile specifications will provide information on whether certain implementations are mandatory or optional.

For any developer/manufacturer who wants to ensure interoperability and compatibility, there is no way around (partially) implementing the ZCL and at least one public profile.

The home automation profile is defined in [6], the smart energy profile specifications are in [7] and the ZCL specs can be found in [5].

2.1.3.6 Other ZigBee Mechanisms

There are three other mechanisms worth mentioning, because they make ZigBee applications a lot more powerful – however; for this project they are only of secondary importance.

As already mentioned a few times, ZigBee has mechanisms for "connecting" two or more devices with each other. This mechanism is called *Binding*. Think of every endpoint as a wire with a certain plug on it (the type of connector is defined via the cluster lists). Binding is the process of connecting two plugs that match (again, think of the analogy with interfaces). The matching is done by comparing the in-clusters of one partner with the out-clusters of the other one and vice versa. This usually involves a comparison of both endpoints' simple descriptors. From a behavioral perspective, both partners will then act as if they were directly connected (even though they are connected wirelessly and might communicate over several intermediate hops) – e.g. binding the smartSwitch with the smartLight.

ZigBee bindings are even more powerful: a binding can be established to a group of devices or even all devices in the network that support the required clusters. One could simply bind a switch to all switchable devices, i.e. devices that have an OnOff-in-cluster, to create a "main-switch". Another possibility is to assign all ZigBee lights to a *group* and then bind a switch to this group – that way one can create a switch to turn off all lights when leaving the house.

When setting up a large-scale ZigBee networks with several dozen or even hundreds of nodes, the process of setting up bindings has to be done in a more structured way. The ZigBee term for this is *Commissioning* – it describes the procedure of allowing nodes to join the network, assigning them to groups and establishing the bindings. For industrial applications, one typically uses a separate commissioning network, where nodes will be provided with all information necessary using a PC-commissioning tool and then reboot the nodes; after booting up they are now able to join the defined PAN and establish the specified bindings. The ZigBee public profiles also describe several constraints regarding the process of commissioning.

Another mechanism worth mentioning is the so called *Reporting*. ZigBee has already built-in functionality for reporting attributes periodically or event-driven. A periodic reporting is used for the measurements of the smartOutlet, which are being sent every two seconds to the coordinator. For an event-driven reporting, one can specify a threshold of "significant change" and a report will only be triggered after observing such a change. A typical application of this would be a light, that sends a report every time its OnOff-status changes.

ZigBee reporting is very powerful and generic. It also allows changes to the reporting configuration during runtime. The main mechanisms are specified in the ZCL [5].

2.1.4 ZigBee from a programmer's perspective

Summarizing the sections above, ZigBee provides a very generic application framework, allowing quite sophisticated applications, such as a monitoring & control tool (using the attributes and the attribute reporting). It also allows the setup of robust bindings – once a binding has been established, it will work as long as the nodes involved are able to communicate with each other. The coordinator node has no special role as far as bindings are concerned – even when the coordinator node breaks down or is unreachable, the existing network is still fully functional. However, in most scenarios, a monitoring & control tool would use the coordinator as a gateway to the ZigBee network.

ZigBee is generic, allowing a broad range of devices, transmitting quite different data types. The encapsulation of this generic data model requires a well-designed programming model. Additionally, the **ZigBee profiles are very specific**, meaning that they provide a vast amount of very specific device- or command- definitions. Re-implementing all these definitions means a significant workload.

Encapsulating ZigBee in your data/programming model results in complexity (to cover all generic aspects) and workload (to cover the vast profile definitions). However, ZigBee is very well designed and partial implementations will usually work well. It is reasonable to decide which features or aspects should be implemented and which ones can be omitted (at least for the early development stages). Consider this proof of concept: the implementation of the very generic reporting or the binding-mechanisms was too complex and would have exceeded the scope of this project. Yet the implementation is fully functional as far as the functional requirements are concerned.

ZigBee nodes can be seen as a tree-structure, where each node has a number of endpoints and every endpoint has a number of in- and out-clusters. Every endpoint can be seen as an object, providing the interfaces defined by the out-clusters and requiring the interfaces defined by the in-cluster.

ZigBee also provides built-in security mechanisms, which have not been explored in-depth in the course of this project. [1] and [2] (in German) provide more details.

2.2 Texas Instruments – Monitor & Test Interface

2.2.1 Interaction with a ZigBee transceiver via Monitor & Test interface

All NTE hardware-prototype ZigBee devices use a transceiver solution from Texas Instruments™. It mainly consists of an antenna, a radio-circuit and a micro-controller. The controller runs a ZigBee application and provides a serial interface (using UART), called *Monitor & Test Interface*. The development kit of the transceiver ships with a neat program called Monitor & Test Tool. The tool uses the serial interface to monitor the ZigBee network (all messages that arrive at a node can be viewed). But it can also be used to invoke commands on the micro-controller (mostly but not only ZigBee commands). See [3] for more on the hardware prototypes.

In the setup for this project, the coordinator is connected (via UART) to NTE's hardware (*HCU Mainboard*) which exposes a web service interface that is used by the GUI to visualize the network and interact with its nodes (see Figure 2). Simply speaking, the Monitor & Test interface allows the HCU mainboard to send (and receive) messages via serial port and communicate (wirelessly) with ZigBee nodes as if they were directly connected via UART.
Texas Instruments has also published an API for the Monitor & Test interface; see [8]. Looking at the specifications, it is very well-suited for building a ZigBee gateway.

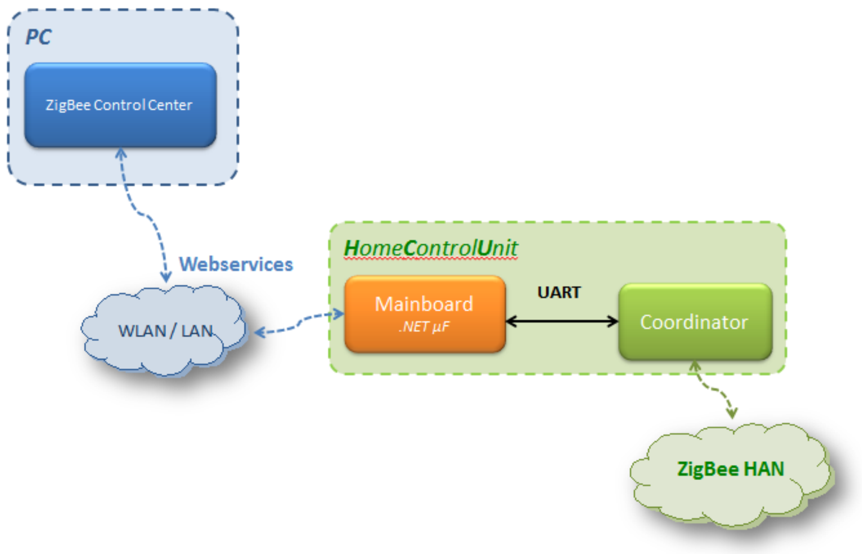


Figure 2 – HCU Interfaces

However, using the M&T interface introduces additional overhead, since ZigBee messages become the payload of another protocol (the M&T protocol). As far as the implementation is concerned, this means that the serial protocol has to be encapsulated first, and then ZigBee commands and messages can be abstracted and encapsulated.

2.2.2 Protocol format

The M&T protocol is based on the transmission of byte-arrays. Every command (or response) is wrapped in an *M&T Frame* – which consists of a start-indicator, the length of the frame, a command-identifier, the actual data and a checksum-field. See Figure 3 for more details. The complete protocol-specification can be found in the M&T API reference [8].

The M&T protocol specifies a lot of commands for interaction with the micro-controller or the running application on the ZigBee transceiver. Only a subset of commands allows sending commands to or receiving messages from other ZigBee nodes. The protocol also grants access (via UART) to internal parts of the ZigBee application, such as the network- or the ZDO-layer.

The meaning of the individual fields is given in Table 1 – a more extensive coverage can of course also be found in [8]. Note that the min**imum message length** of an M&T message is **five bytes** (one byte SOF, one byte LEN, two CMD-bytes and one byte FCS).

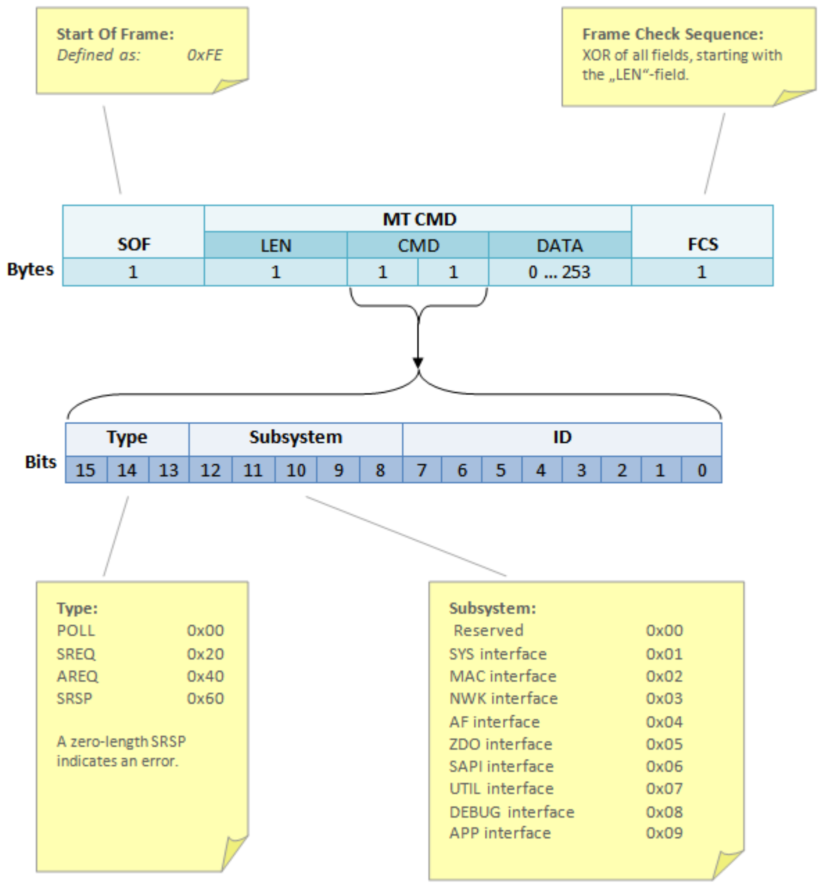


Figure 3 - Monitor & Test API Frame Format

SOF	Start of Frame – indicates the beginning of a new frame and is defined with 0xFE
LEN	Length-field – number of bytes in the data-field
CMD	Command Identifier – defines the "meaning" of the message and also the "meaning" of the payload data (which consists of command parameters). The command is formed of two bytes, where the meaning of the individual bits can also be seen in Figure 3.
DATA	Payload of the message – this field contains the command-parameters, which can also be interpreted as return-values for response-commands. The content of the data-field is can be found in the corresponding command definition (see [8])
FCS	Frame Checksum – checksum field for validating message integrity. Since this is computed by XOR-operation, this will only prevent accidental data corruption; a systematic manipulation cannot be detected.

Table 1 – Fields of the M&T frame

2.3 .NET Micro Framework

2.3.1 Overview

The .NET Micro Framework (MF) is the “smallest” platform for managed .NET code from Microsoft™. The minimal configuration runs on 32-bit processors with 64kB RAM and 256kb ROM and does not require an MMU. More realistic configurations (including the network stack and GUI display components) have 2 to 8 MB RAM and 4 to a 10 MB ROM. The best supported processor platform is the ARM 9 or the ARM 7, but ports to other platforms are also available.

In fall 2009, Microsoft has released the full source code of the Micro Framework into public. Thus, it can be used free of any charges and it can easily be adapted to the specific needs of a company. There is a growing community, releasing a variety of extensions to the framework and the original project division at Microsoft is working on the main branch of the project.

The .NET Micro Framework provides a runtime-environment for the interpretation (!) of C# code, where only a subset of the “full” .NET framework is being supported. One big advantage is that most parts are, as far as interfaces and naming conventions are concerned, fully compatible to the full .NET framework. This means that complete projects or at least large parts of them can either run on the micro-controller platform or on a PC or even in a web-application, without any changes to the code. Another result of this is that writing applications for the .NET Micro Framework is very similar to writing desktop-PC applications, which means that developers with great .NET skills but little embedded experience can easily write embedded applications (usually, skilled .NET developers can be found more easily than skilled embedded developers).

The MF does not require an underlying operating system, but can be integrated into an existing one. It can significantly reduce development times, because it comes with a vast number of very useful built-in functionality, like:

- RS232, I^{2}C and SPI-communication
- Ethernet, including HTTP(S)-sockets
- De- and encryption
- Support for graphical displays and touchscreens
- Huge library of graphical GUI components
- Analog and digital I/O’s (also PWM)
- Timers
- Multithreading
- Native XML-parsing
- Web service stack
- File system

Additionally it uses the excellent Visual Studio™ IDE and provides full debugging support (including conditional breakpoints and expression-evaluation during a halt). Other embedded platforms can often be extended to have the same functionality by using e.g. C-libraries. In practice, however, it usually takes some tuning to actually integrate several different libraries. The .NET Micro Framework on the other hand is all of a piece – developers can really focus on writing applications and do not have to worry about integrating libraries, which can significantly reduce the time-to-market.

More on the .NET Micro Framework can be found in [9] and [10]. In this project, the Micro Framework is the basis platform for the HCU mainboard (see Figure 2).

2.4 Web services

2.4.1 Overview

Web services are the implementation of a more general design principle called *SOA (Service Oriented Architecture)*. In SOA, a business process is divided into small, functionally complete, units, called services. By consecutive execution of these services, the whole business process can be reproduced. This principle enforces the design of loosely coupled units (services) that are very modular and can be re-used in other processes/projects as well. Another advantage of SOA is that services can easily be distributed on several computing machines. For a well-designed service, it should be enough to only know its interface and not have to worry about its internal structure or behavior.

Web services have been defined by the W3C and describe a framework for using distributed, service oriented applications on the web. Similar to XML RPC (remote procedure calls), web services use structured XML-documents (wrapped in an HTTP-envelope) to invoke actions on a service host and to return the results of that action. This concept is known as *service calls* and it is not exactly the same as remote procedure calls. Service calls require even less coupling, since they only specify the data-structures transmitted as parameters or return-values and the corresponding operations. The main protocol used by web services is known as SOAP (simple object access protocol).

Another advantage of web services is their use of standard technologies (HTTP, XML …), which makes them platform-independent and easy to use. Furthermore, almost every web service framework comes with a code-generator that handles the tedious work of writing wrappers and converters, to translate objects into valid XML service calls and vice versa.

It is therefore only necessary to specify an interface-like description of the service in the so called WSDL (web service description language), which is an XML dialect. This enables another popular design principle known as *design by contract*, where one designs the interface of the service first (the contract) and then client- and server-side can be implemented independently, as long as they abide by the contract.

The web service standards are divided into several sub-specifications (like WS-Addressing, WS-Security, WS-Events …) that are often simply called WS-* specifications. For an introduction to web services from the W3C, see [11].

2.4.2 Web services with the .NET Micro Framework

The .NET Micro Framework comes with a special web service stack and the corresponding code generator. The implemented stacks supports the *DPWS (Devices Profile for Web Services)*, which has been proposed by Microsoft and has become standardized by the OASIS. The profile aims at devices with limited resources, such as embedded platforms.

In web service terminology, a profile specifies which WS-* standards in which version need to be implemented. Profiles define mandatory and optional parts and have been introduced to enhance interoperability (since different versions of the same WS-* standard might be incompatible). If both parties support the same profile – they are guaranteed to be compatible.

Probably the biggest drawback of the DPWS-implementation on the Micro Framework is its lack of any form of WS-security (DPWS only defines WS-security as optional). That way, it cannot directly be used to transmit sensitive user data in the consumer market. However, since the code of the Micro Framework is open, one could simply add WS-security to the existing stack – with the corresponding workload, of course.

The DPWS-specifications can be found online, see [12].

3 Realization

3.1 System Level

3.1.1 "The proof of concept"

NTE Systems has already developed a controller, called the *ECU (Energy Control Unit)*, targeted at the solar water heating domain. The controller has been designed for small-size plants and has been geared towards being used by a non-expert. A graphical editor can be used to modify the control program. In a parallel project, NTE has developed a *structured control and data acquisition (SCADA)* software for small- to large-scale scenarios, called *SCADY*. This software allows monitoring and telecontrol of a control-system. It can also generate reports and charts and allows access via web-interface. In the current project-stage, the ECU is being integrated into this SCADA system.

The ECU has a flexible design and could easily be used for different scenarios (not only solar water heating); especially with the powerful SCADA software and the aim to be operated by a non-expert user, the system could have a unique market position.

In the U.S. and in some European countries, a new power supply technology, called smart grid, is advancing. The smart grid allows an electric supply company to have a certain control on the consumer's devices. E.g. if the consumer tells the system to switch on the dishwasher during the day, the supply company can choose the exact time to switch it on and by doing so it can flatten usage peaks. To enable such scenarios, households will be equipped with smart energy meters that have a ZigBee node built in. The industry has chosen ZigBee as *the* technology for wireless in-home communication and the ZigBee Alliance has reacted with a special profile – the ***Smart Energy Profile***.

By using this smart energy meter, the user could easily be able to display his at-the-moment energy consumption via a web-interface, which will hopefully increase the consumer's energy awareness (getting a bill once a year will not reveal "energy-hungry" household appliances). Moreover the smart meter can query the current rates and might one day be able to simply switch the energy supplier if another one has better price at the moment.

Raising the energy awareness is also one of NTE's goals. Their device seems naturally suited to play a role in the upcoming smart energy market. The only technological lack is the missing ZigBee interface. By having such an interface, the ECU might even be open to a lot more markets, such as home automation. Due to that, NTE has decided start a proof of concept project to get to know the ZigBee technology and gain some domain know-how.

The proof of concept system consists of the four hardware prototypes (that were developed in-house as well): coordinator, smartLight, smartSwitch and smartOutlet. For more details, see 2.1.2. To "connect" these hardware prototypes to some kind of software GUI and to get a preview on the upcoming integration into the ECU-software, the goal was to implement parts of the coordinator-protocol (Monitor & Test protocol) as well as parts of the ZigBee specifications on the existing ECU-hardware, using the .NET Micro Framework. An overview of the POC system can be seen in Figure 4 (note that the ECU equipped with a coordinator is called *Home Control Unit – HCU*).

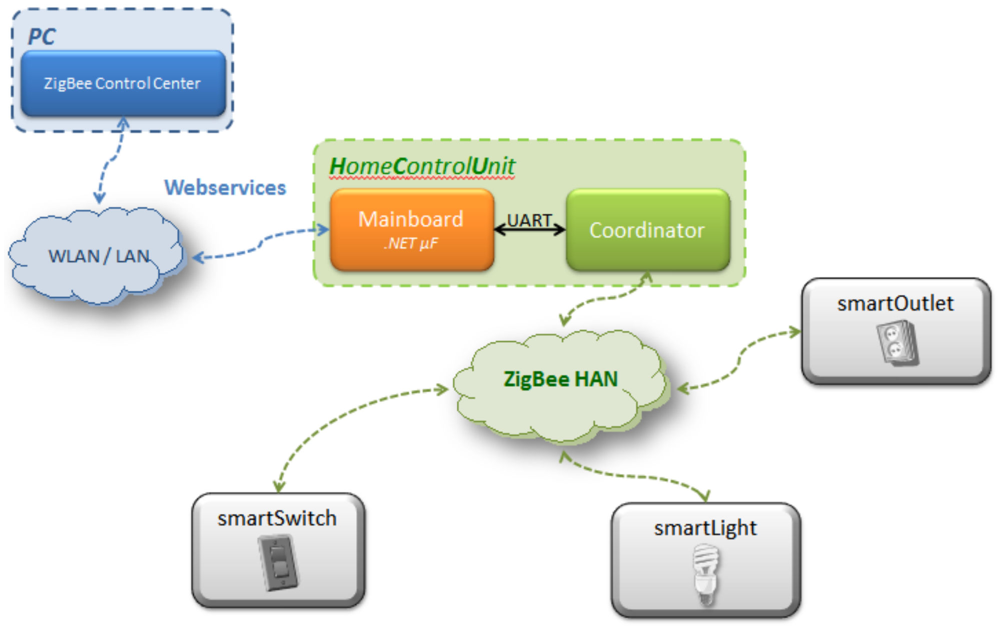


Figure 4 – POC System Overview

The HCU is composed of a mainboard (similar to the one on the ECU) and a ZigBee transceiver (acting as a coordinator for a home area network - HAN) connected via serial port. The other network nodes are the smartLight, the smartSwitch and the smartOutlet. Further, the software on the mainboard exposes web services that are being consumed by a GUI-client, the *ZigBee Control Center*. The client can run on any computer within the same network (LAN) as the HCU (which has an Ethernet connector, no WLAN).

The ZigBee Control Center can visualize the ZigBee network and its current nodes in a tree-structure. It can also display additional information on the functionality of the nodes (endpoints and corresponding simple descriptors). Via the GUI, the user can permit other network nodes to join the network (by opening a join-window for a specified amount of time). Other functionalities are the interaction with the smartLight and smartOutlet (switching it on and off, reading its switching state) and displaying the measurement values of the meter built into the smartOutlet.

3.1.2 Goals for the POC

After an initial research phase, it soon became clear, that the whole ZigBee functionality (like reporting or security) cannot be explored within the scale of this project; partially because its complexity was too high but mostly because the expected workload was simply too large. The following user stories (Table 2) have been agreed on, with the intention to narrow down the vast extensiveness of ZigBee while still leading to a demonstrative implementation and being able to gain some advanced insight on working with ZigBee.

ID	Requirement
US1	As a user, I want to get an overview of the ZigBee network, using the software interface (GUI) to see the network nodes available.
US2	As a user, I want to see the functionality of individual network nodes via the user interface, to determine the type of the node and its valid operations.
US3	As a user, I want to be able to allow new nodes to join the network for a certain amount of time, to have a secure way of adding nodes to the network.
US4	As a user, I want to be notified on the user interface (GUI) when a new node joins the network to be

	able to react accordingly.
US5	As a user, I want to read the logical switching state of the smartLight and the smartOutlet, to be able to inform myself about their current status.
US6	As a user, I want to be able to modify the switching states of the smartLight and the smartOutlet via the user interface, to be able to telecontrol them.
US7	As a user, I want to be able to visualize the current measurements of the smartOutlet on the user interface, to inform myself about the current power consumption.

Table 2 - POC User Stories

For those, not familiar with the concept of user stories, this might seem, to say the least, "a little diffuse". However, these user stories are intended to represent the user's (or customer's) view on the functional requirements of the system in a non-technical description (because most customers a non-technicians). Every user story has to answer "the three W's": Who wants to do what and why? These indistinct descriptions are then being translated into functional and non-functional requirements by the development team. In the agile development process SCRUM, these user stories are then prioritized and implemented in, usually, three-week cycles called sprints (according to the defined requirements). That way, the team makes sure to develop exactly the functionality and behavior that the customer wants – because the customer is usually not able to directly give concrete software-requirements.

The given user stories basically describe the same functionality as the POC system overview in the previous section.

To clearly define the boundaries of the implementation, the following issues have been explicitly omitted, thus are not part of the project's implementation:

- Binding via user interface (GUI) and readout of existing bindings
- Reporting – configuration, ZCL commands and their implementation in the firmware of the hardware prototypes (note that the measurements of the smart outlet use reporting though, but in this case the configuration is static – hardcoded – and cannot be changed).
- Notification of the user of key-press events on the push-buttons of the smartSwitch – the corresponding mechanism would be the reporting
- Extensive encapsulation of the ZCL including all their data types and commands (this would have simply exceeded the possible workload within this project)
- Security and the trust center as defined in the smart energy profile

3.2 Implementation Overview

3.2.1 Visual Studio solution structure

The project's Visual Studio solution is divided into several sub-projects. As an orientation, below is a short description of each of the projects.

ECUEmulator:

This project is a Micro Framework Emulator that mimics the current (winter 2010) ECU hardware. It has been used in the early stages of the project, because there seemed to be a problem with the event-driven UART-communication on the original emulator. After posting the problem to the .NET MF forums, it was stated, that this mode of operation is not supported on the emulator. However, in later project stages, the mode of operation for the serial-port has been changed to POLLING anyway.

MFZStackApp:

This is basically the application, running on the Micro Framework (ECU hardware). It boots up the system, starts the DPWS service infrastructure and also the web service. It also starts the ZStackCOMHandler, which takes care of communicating with the ZigBee coordinator via UART. This project does not contain crucial parts of the implementation, but can be seen as an example usage. This project requires the installation of the .NET Micro Framework SDK (developed using version 4.1)

UARTExtensionService:

This project contains the web service definition (the contract classes). It is only needed for generating the web service code (service code for the MF, using MFSvcUtil.exe and the client code using a service reference). Changes to the web service have to be made here but then require a re-generation of the code (on both the server and client side).

ZigBeeControlCenter:

This project is "the WPF GUI" (Windows Presentation Foundation). This GUI is not intended to be used for further developments; therefore it has been implemented “quick and dirty". Parts of it can be seen as a usage example of the web services and the ZStackCOM library. However, it is not recommended to extend this project, but rather REIMPLEMENT it, if needed. As far as the code is concerned, this program is very similar to the ZStackApp (the Windows Forms GUI). The theme used (see App.xaml), requires the installation of the WPF toolkit.

ZStackApp:

This project is "the Windows Forms GUI". It was used for testing purposes throughout the whole development and has been extended several times. As it never was intended to be used after that POC-project, the code is untidy with some copy & paste parts and quick hacks. This application has been **entirely replaced** by the ZigBeeControlCenter (which is in WPF and no longer "aircraft carrier grey").

ZStackCOM:

This project has (by far) the most code in this solution. Most of the developments have been made in this project – the implementation is well-structured and well-documented and intended to be used or extended in consecutive projects. It contains the ZSTackCOMHandler, which reads and writes (and processes) messages to and from the serial port. It also contains the MTCommands namespace, which is an (extensive) object hierarchy, encapsulating the raw bytes of the serial ports into "useful" messages (Monitor & Test Interface as well as ZigBee specific). *Use this namespace and the provided base classes to extend the command infrastructure*. Also, it defines an exception base-type and a few derived exceptions.

This project has been tested "by using it" thoroughly, a few bugs have been fixed and it runs very stable now - nonetheless this can never replace a test bed. Probably the very first step for extending this library would be the setup of a corresponding test bed (unit-tests for all commands, regression-tests for the whole library ...). This project requires the installation of the .NET Micro Framework SDK (developed using version 4.1).

The technical documentation for this project has been written into the source-code (using the C#-style XML-comments). It can easily be generated into a neat document using Microsoft’s Sandcastle tool.

3.2.2 COMHandler

The `COMHandler` is a class defined in the ZStackCOM project (its class diagram can be seen in Figure 5). As its name suggests it is responsible for handling all communication over the serial port (the COM port), i.e. communicating with the coordinator; sending and receiving M&T messages (Monitor & Test interface).

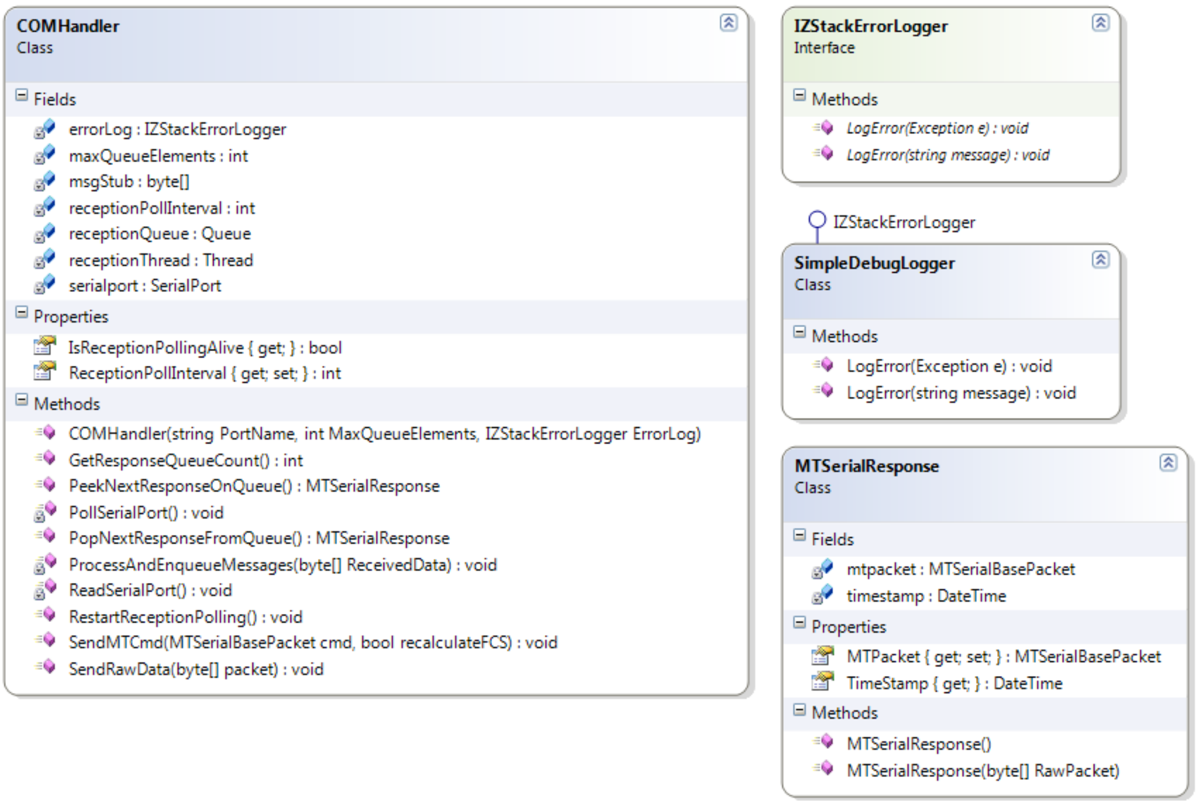

COMHandler
Class
Fields
errorLog : IZStackErrorLogger
maxQueueElements : int
msgStub : byte[]
receptionPollInterval : int
receptionQueue : Queue
receptionThread : Thread
serialport : SerialPort
Properties
IsReceptionPollingAlive { get; } : bool
ReceptionPollInterval { get; set; } : int
Methods
COMHandler(string PortName, int MaxQueueElements, IZStackErrorLogger ErrorLog)
GetResponseQueueCount() : int
PeekNextResponseOnQueue() : MTSerialResponse
PollSerialPort() : void
PopNextResponseFromQueue() : MTSerialResponse
ProcessAndEnqueueMessages(byte[] ReceivedData) : void
ReadSerialPort() : void
RestartReceptionPolling() : void
SendMTCmd(MTSerialBasePacket cmd, bool recalculateFCS) : void
SendRawData(byte[] packet) : void
IZStackErrorLogger
Interface
Methods
LogError(Exception e) : void
LogError(string message) : void
IZStackErrorLogger
SimpleDebugLogger
Class
Methods
LogError(Exception e) : void
LogError(string message) : void
MTSerialResponse
Class
Fields
mtpacket : MTSerialBasePacket
timestamp : DateTime
Properties
MTPacket { get; set; } : MTSerialBasePacket
TimeStamp { get; } : DateTime
Methods
MTSerialResponse()
MTSerialResponse(byte[] RawPacket)

Figure 5 – Class Diagram of ZStackCOMHandler (and ErrorLogger)

The following discussion will present a few mechanisms that are implemented by the `COMHandler` and will shed some light on its internal operation. A detailed description for all properties and methods can be found in the in-code documentation.

The main task of the `COMHandler` is to *poll the serial port* for incoming messages. It does this, by starting a separate thread (the `receptionThread`). This thread will periodically check the serial port for new messages (polling) by calling `ReadSerialPort`. If a new message has been received, the `COMHandler` will call `ProcessAndEnqueueMessages` where the received bytes will be used to instantiate a new `MTSerialResponse` object (which is essentially a timestamp plus an `MTSerialBasePacket` – see 3.2.3 for details). This instantiation will check, whether the read-bytes are a valid M&T frame, i.e. SOF-field, LEN-field and FCS have to contain valid values. If these checks fail, an exception will be thrown.

If the instantiation was successful, the object will be added to the `receptionQueue`. This queue can then be read, using the `PopNextResponseFromQueue` method. In the current implementation, this is only used by the web service (see 3.2.4), which provides operations for reading a single message or all messages from the queue.

If this queue was never read out or if more messages are received than read out over time, the queue would grow constantly. To prevent this, the `COMHandler` allows the specification of a maximum number of elements on the queue (`maxQueueElements`), which can be set when calling the constructor. If this maximum number of elements is reached, newly received messages will be discarded (the messages on the queue will still remain).

If an unhandled exception occurs in the `receptionThread`, it will terminate – however, there is a property to check, if the thread is still alive and a method to restart the thread if it has crashed (`RestartReceptionPolling`). The intended use can be seen in Code-fragment 1 – originating from "MainApp.cs of the MFZStackApp project" - without any actual exception handling in the parent.

```
while (true)
{
    //keep device alive
    System.Threading.Thread.Sleep(1000);

    //check if serial reception thread is still alive
    if (!com_handler.IsReceptionPollingAlive)
    {
        logger.LogError("[MainApp] Reception thread is dead - restarting reception polling!");
        com_handler.RestartReceptionPolling();
    }
}
```

Code-fragment 1 – Keeping the receptionThread alive

Processing of message stubs in the COMHandler

The `COMHandler` has a default polling interval of 100ms. In practice it happens quite often, that within this interval, more than one M&T message has been written to the reception buffer of the serial port. This case is not too problematic, since every M&T frame starts with an SOF-field (0xFE). The `ProcessAndEnqueueMessages` method can handle such a case and will simply process one message at a time and add it to the `receptionQueue`, as long as it is not full.

But what if the readout of the serial reception buffer happens exactly during message reception – what if only a part of the message is already in the buffer? This case is called processing of *message stubs* and is also being handled in the `ProcessAndEnqueueMessages` method.

Take a look at Figure 6 – the ideal case: several M&T frames have been written into the buffer of the serial port since the last readout. Fortunately, all frames are complete and can be processed. Since every frame (by definition) starts with the known SOF field (start of frame – 0xFE) the individual messages can easily be separated and handled.

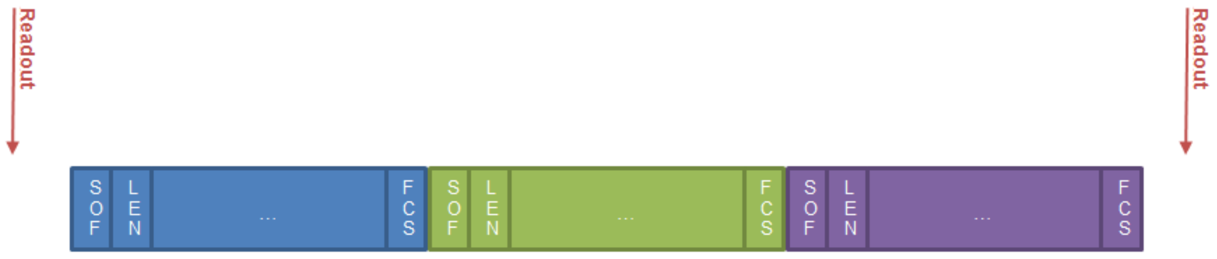


Figure 6 – Readout after message reception of multiple packets

In contrast to that, consider Figure 7, where a readout occurs during the reception of a message. The result is a message-stub at the end of the buffer. Because the SOF-field is present, the start of the new frame can be detected, but during instantiation of the corresponding `MTSerialBasePacket` an exception will occur, because the actual length of the data does not match the one given in the LEN-field (note that a message, shorter than 5 bytes is invalid anyway, since the minimum M&T message length is 5 bytes – see 2.2.2 for more details). If such an exception occurs when processing the last message of the buffer – this message-stub will be ***stored*** until the next readout – if the next readout does not start with an SOF-field, it contains the rest of the previous message at the beginning of the buffer – see Figure 8. This “rest” can easily be identified, since the next valid message (if multiple messages were received) has to start with an SOF-field.

If a stored stub is present and the current buffer starts with a stub – these two stubs are ***merged and processed***. If both stubs did not result from a single message being interrupted, the merged packet is very likely to raise an exception during instantiation; the `COMHandler` will then discard the data.

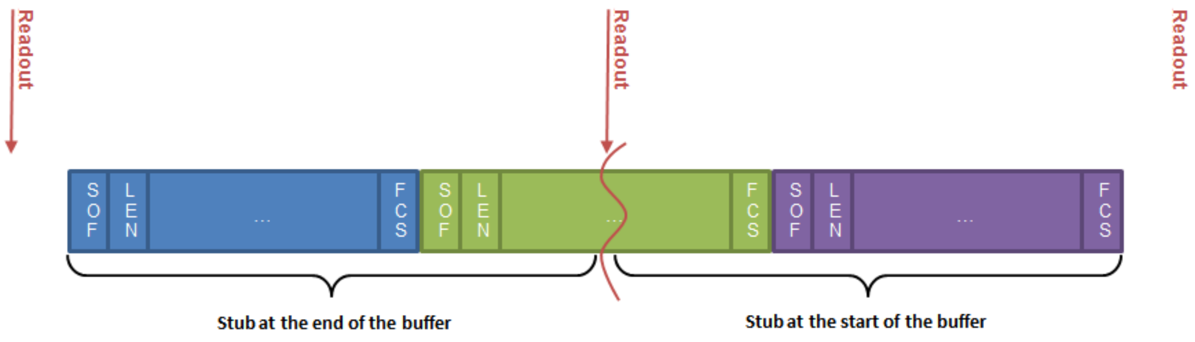

Readout
Readout
Readout
SOF
LEN
...
FCS
SOF
LEN
...
FCS
SOF
LEN
...
FCS
Stub at the end of the buffer
Stub at the start of the buffer

Figure 7 - Readout during message reception

> *A message stub can occur at the very end of the serial port's reception buffer. In that case, the LEN-field will not match the actual length of the data and the stub will be stored. The only other case, where a stub can occur is the beginning of the reception buffer. If there is such a stub and the `COMHandler` has already stored an older message-stub, both parts will be merged and processed.*

There are also cases, where the reception buffer does not contain multiple M&T frames but only a single stub – of course in that case the stub is both at the beginning and the end of the buffer.

In the case that a new stub to store occurs (because it is at the end of the buffer and there have been multiple frames) and another stub is stored already, the "old" stub will be overwritten, since there is obviously no chance that the second part of that stub (the merge-part) will arrive.

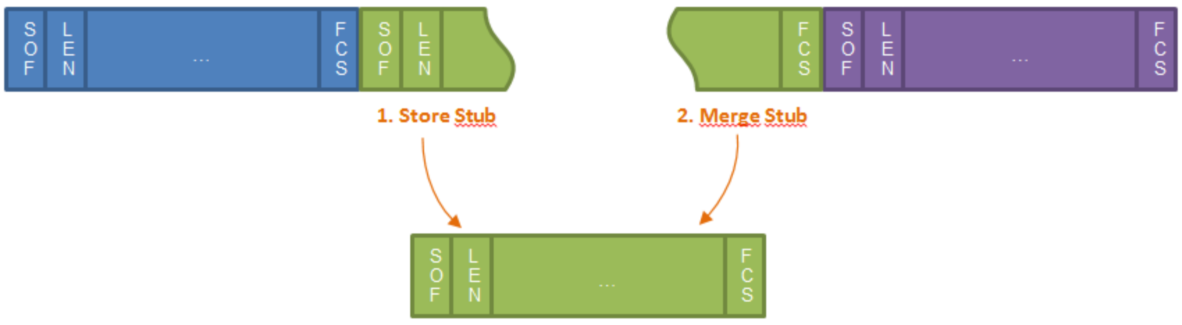


Figure 8 - Merging the new stub with the stored one to restore the packet

3.2.3 MTCommands

The `MTCommands` namespace in the ZStackCOM project contains class definitions for all the implemented M&T commands and responses. Some of the messages are further used to encapsulate a raw-byte ZigBee payload. See A.1 for an overview of the classes in the `MTCommands` namespace. All M&T messages are defined in the M&T API ([8]). The namespace is divided into sub-namespaces to make it easier to find commands given in the M&T API.

The basis of the communication via M&T is the exchange of byte arrays via serial port. This is very common in the embedded domain as it introduces very little overhead, but it does impose several weaknesses. The M&T protocol does not transmit any kind of type information – a 32-bit integer gets simply translated into the corresponding 4 bytes. The type information is only given in the M&T API definitions. Unfortunately, this is not very generic, thus every command has to be implemented specifically. (Sidenote: ZigBee messages are defined similar, but in some cases - like reporting – use an enumerator to transmit type-information).

The basic goal of the implementations in the `MTCommands` namespace is to provide (wrapper-) classes for simple access to the different data-fields. E.g. a class that has an integer-property which is being ***mapped*** to the right position of the raw byte array. In some cases this is not sufficient, since access to individual *bits* of the data is needed – of course, this also has to be encapsulated by the library.

> *The goal of the messages defined in the `MTCommands` namespace, is to provide **strongly typed** wrapper classes for the raw-byte data of the M&T frames.*
>
> *Since this namespace is part of a library, all issues dealing with the mapping to/from these underlying byte arrays must be encapsulated. A user of the library should have typed message classes that provide transparent access to the data fields.*

3.2.3.1 MTSerialBasePacket

This class is the base class for all implemented commands and messages. Therefore it contains most of the involved mechanisms. A class diagram can be seen in Figure 9.

As far as fields and properties are concerned, the `MTSerialBasePacket` provides access to the fields defined by the M&T API (see 2.2.2) – namely:

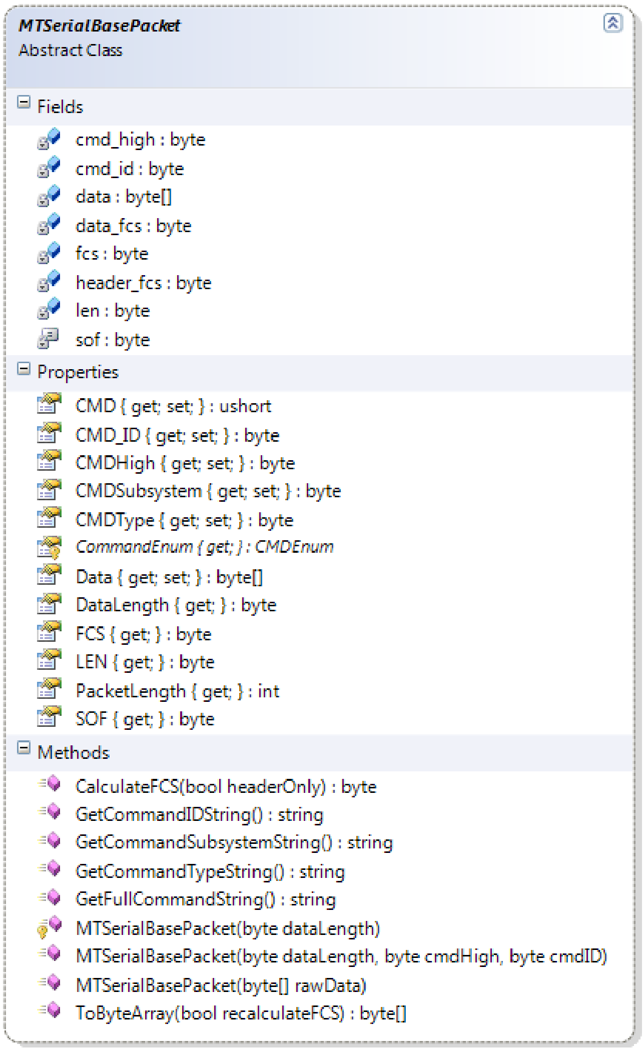
- LEN-field (data length)
- CMD-field (two command bytes encapsulated by several properties for accessing the individual sections, such as the high-byte or the command-subsystem)
- Data-field (actual payload data, containing parameters for the specified command)
- FCS-field (checksum)

The class provides three constructors:

- For internal use only – will allocate the underlying byte-array according to the given dataLength. This constructor is used to build commands to send – in that case one usually knows the length of the data but not the actual content.
- Taking a byte array as an argument – is used during message reception (where both the data length and the content is known) or for “copying” a message. Will perform a number of checks, to ensure the validity of the packet.
- Taking the data length and the command-bytes as an argument – this constructor can be used for instantiating a message, where no derived class exists.

To ensure that the underlying byte array is actually a valid M&T message, a number of constraints can be checked.

> *The maximum data-length is 254 bytes; the minimum length of the whole packet is five bytes (see 2.2.2).*
>
> *Every message has to start with an SOF field which has the value 0xFE.*
>
> *The value in the LEN-field must match the length of the actual data (the whole packet is five bytes larger than the value of the LEN field).*
>
> *The values in the CMD-field (two bytes) must match the values defined for that command (e.g. if the command is SYS_PING, the CMD-bytes must be 0x2101).*
>
> *The value in the FCS-field must match the checksum of the data which is computed by performing an "XOR" over all bytes, beginning with the LEN-field*



MTSerialBasePacket
Abstract Class

Fields
- cmd_high : byte
- cmd_id : byte
- data : byte[]
- data_fcs : byte
- fcs : byte
- header_fcs : byte
- len : byte
- sof : byte

Properties
- CMD { get; set; } : ushort
- CMD_ID { get; set; } : byte
- CMDHigh { get; set; } : byte
- CMDSubsystem { get; set; } : byte
- CMDType { get; set; } : byte
- *CommandEnum { get; } : CMDEnum*
- Data { get; set; } : byte[]
- DataLength { get; } : byte
- FCS { get; } : byte
- LEN { get; } : byte
- PacketLength { get; } : int
- SOF { get; } : byte

Methods
- CalculateFCS(bool headerOnly) : byte
- GetCommandIDString() : string
- GetCommandSubsystemString() : string
- GetCommandTypeString() : string
- GetFullCommandString() : string
- MTSerialBasePacket(byte dataLength)
- MTSerialBasePacket(byte dataLength, byte cmdHigh, byte cmdID)
- MTSerialBasePacket(byte[] rawData)
- ToByteArray(bool recalculateFCS) : byte[]

Figure 9- Fields, Properties and Methods of MTSerialBasePacket

In order to check the values of the CMD-field against the values defined for that command, every message-class has to implement the read-only property `CommandEnum` (actually it has to override the abstract property). In some cases it is not possible to know the command in advance, or it might be more useful to use a generic message class, that does not care about the content of the CMD-field. For this purpose, one can use the `MTGenericPacket` class - its `CommandEnum` property will always return the same values as given by the two CMD-bytes. This means that the CMD-check will always succeed.

The checksum of the packet has to be recalculated after any changes to one of the fields and the value in the FCS field has to be updated. The `MTSerialBasePacket` provides a method

(`CalculateFCS`) to (re-) calculate the FCS. Since the calculation of the checksum requires a certain computational effort and the library is intended for use on Micro Framework devices, this re-calculation will usually not be done automatically. When converting a packet into a byte array (by using `ToByteArray`), a Boolean parameter can be passed to specify whether to calculate the FCS before the conversion or not.

For more information on the internal mechanisms of the `MTSerialBasePacket`, see the in-code documentation or rather use it to generate the technical documentation.

3.2.3.2 Requests and Responses

The implemented classes can be split into three categories:

- Requests – denoted by the suffix "Req" in the class name
- Synchronous responses – denoted by the suffix "_Rsp" in the class name
- Asynchronous responses – denoted by the suffix "_ARsp" in the class name

However, note that a single request can involve a synchronous as well as an asynchronous response. All requests that the coordinator can process directly will usually result in a synchronous response, containing the requested results. If the coordinator has to send the request to another node in the network, it will confirm this by sending a synchronous response, indicating whether the message was transmitted successfully or not. The receiver of the request will process it and return the results with an asynchronous response.
In some cases, a ZigBee node will send a message to the coordinator without receiving a previous request; e.g. when a new device joins the network it "announces" this by sending a broadcast to the network.

The synchronous response to a request that needs to be sent to a node in the ZigBee network always looks the same and only contains a single byte of payload data (indicating success of failure). Because of that, there is a special base class for this kind of synchronous responses – the `MTGenericResponse`. Note that a "return value" of zero indicates a successful transmission and any non-zero value corresponds to an error code. Unfortunately these codes are not defined in the M&T API document (they might be in some firmware-related document of the ZStack).

Note that the coordinator might return a synchronous response indicating success, but the receiver node of the request cannot process the request and may (in some cases) not return an asynchronous response at all. The success-indication in the synchronous response does only regard the processing on the coordinator.

3.2.3.3 Sending and Receiving ZCL Commands

The M&T interface provides a special command for sending arbitrary payload data to a node in the ZigBee network: the `Af_DataReq`. The corresponding (asynchronous) response from the network will be wrapped into an `Af_Incoming_Msg_ARsp`. In Figure 10 you can see these two base classes with all of its derived classes. First there are wrappers, specific to ZCL payload data (all commands in use with one of the public profiles are based on the ZCL message format). Derived from these wrappers, there are implementations of specific ZCL commands.

Note that the ZCL distinguishes between *manufacturer-specific* and *non-manufacturer* commands. Some companies prefer to extend or adapt their solutions while still being compatible with the ZCL. Therefore, manufacturer-specific commands can be defined for these extensions. In the case of NTE Systems, only standard-commands are used.

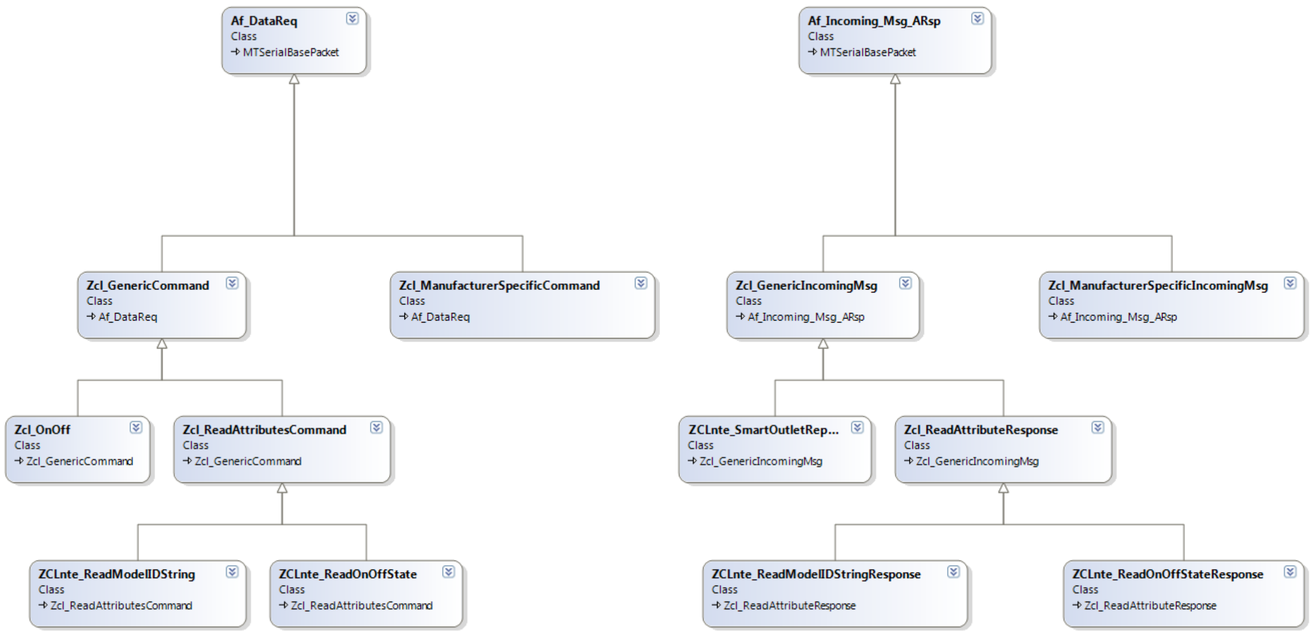
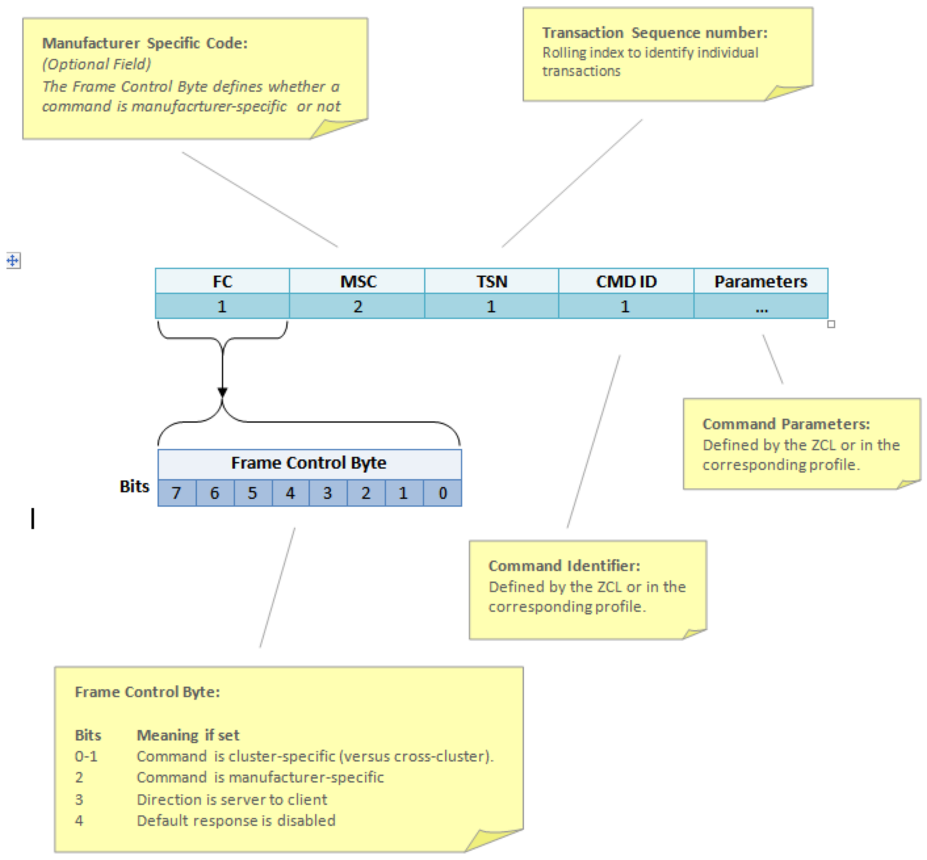


Figure 10 – Classes Involved for Sending and Receiving ZigBee messages

The general format of a ZCL message can be seen in Figure 11. Note that the ZCL header consist of either 5 bytes for manufacturer specific commands or only 3 bytes for non-specific commands. Also, the ZCL defines so called "cross-cluster" commands, which will work on every cluster (such as reading attributes or configuring the reporting), whereas cluster-specific commands can only be processed if the endpoints of the sender and receiver match (i.e. the corresponding in- and out-cluster(s) must match).



Manufacturer Specific Code:
(Optional Field)
The Frame Control Byte defines whether a command is manufacrturer-specific or not

Transaction Sequence number:
Rolling index to identify individual transactions

FC	MSC	TSN	CMD ID	Parameters
1	2	1	1	...

Frame Control Byte							
7	6	5	4	3	2	1	0

Command Parameters:
Defined by the ZCL or in the corresponding profile.

Command Identifier:
Defined by the ZCL or in the corresponding profile.

Frame Control Byte:

Bits	Meaning if set
0-1	Command is cluster-specific (versus cross-cluster).
2	Command is manufacturer-specific
3	Direction is server to client
4	Default response is disabled

Figure 11 - Format of ZCL Messages

3.2.4 Web Service Interface

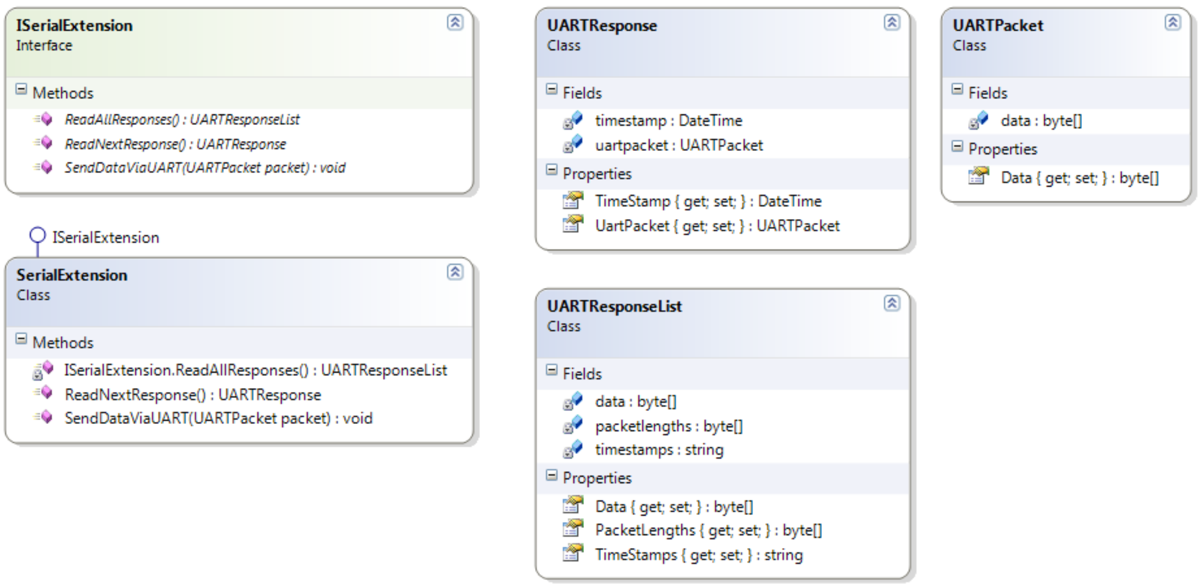
The ZigBee Control Center GUI (running on a PC) requires a communication interface to the HCU mainboard. Fort this POC project, a simple web service interface was chosen. Note that both, the ZigBee Control Center and this web service interface will be replaced by more sophisticated solutions in the course of the integration of the ECU/HCU into the SCADY-system. Therefore, both parts have been implemented with a focus on keeping development times short but without much concern on flexibility, extensive coverage or reusability.

The web service interface allows the readout of the `receptionQueue` of the `COMHandler` (see 3.2.2). It supports reading either a single packet, or all packets on the queue. Further, it also provides an operation for sending commands to the coordinator.

The definition of the web service, called `UARTExtensionService`, can be found in the UARTExtensionService-project. The implementation of the service host (on the MF device) is in the MFZ-STackApp-project (*UARTExtensionServiceImplementation.cs*). Implementations for the clients can be found in both GUI-projects – the client is implemented by a *Service Reference*.

As stated in 2.4, every web service is defined by a **contract** – written in WSDL. When using Microsoft .NET and the *Windows Communication Foundation (WCF)*, one can simplify this contract definition by writing an interface in code and using .NET attributes. If you are not familiar with WCF or web services in general, you can find many tutorials and introductory articles on the web, especially recommended are CodeProject or the MSDN (Microsoft Developer Network) – for a platform independent introduction, see [11].

The complete class diagram for the `UARTExtensionService` can be seen in Figure 12. The operations are defined in the `ISerialExtension` interface – the data types used as parameters or return values are also shown in the class diagram.



ISerialExtension
Interface
Methods
ReadAllResponses() : UARTResponseList
ReadNextResponse() : UARTResponse
SendDataViaUART(UARTPacket packet) : void
ISerialExtension
SerialExtension
Class
Methods
ISerialExtension.ReadAllResponses() : UARTResponseList
ReadNextResponse() : UARTResponse
SendDataViaUART(UARTPacket packet) : void
UARTResponse
Class
Fields
timestamp : DateTime
uartpacket : UARTPacket
Properties
TimeStamp { get; set; } : DateTime
UartPacket { get; set; } : UARTPacket
UARTResponseList
Class
Fields
data : byte[]
packetlengths : byte[]
timestamps : string
Properties
Data { get; set; } : byte[]
PacketLengths { get; set; } : byte[]
TimeStamps { get; set; } : string
UARTPacket
Class
Fields
data : byte[]
Properties
Data { get; set; } : byte[]

Figure 12 - Class Diagram of the Web Service Interface

To demonstrate the simplicity of WCF for defining web services, consider Code-fragment 2 where the complete definition of the service contract is shown. As an example for defining a data type in a so called *Data Contract*, see Code-fragment 3 – the definition of the UARTPacket type, which basically consists of a single byte array.

```
[ServiceContract]
public interface ISerialExtension
{
    [OperationContract]
    void SendDataViaUART(UARTPacket packet);

    [OperationContract]
    UARTResponse ReadNextResponse();

    [OperationContract]
    UARTResponseList ReadAllResponses();
}
```

Code-fragment 2 - Service Contract Definition for UARTExtensionService

The service contract then gets “automatically” translated into the corresponding WSDL-file, which can be fetched from the service host. The WSDL actually consists of a root-file, including several XML schema definitions (XSD-files). The definitions seen above (class diagram and code) will be translated into the schema seen in Figure 13.

```
[DataContract]
public class UARTPacket
{
    byte[] data;

    [DataMember]
    public byte[] Data
    {
        get { return data; }
        set { data = value; }
    }
}
```

Code-fragment 3 - Definition of Data Contract for UARTPacket class

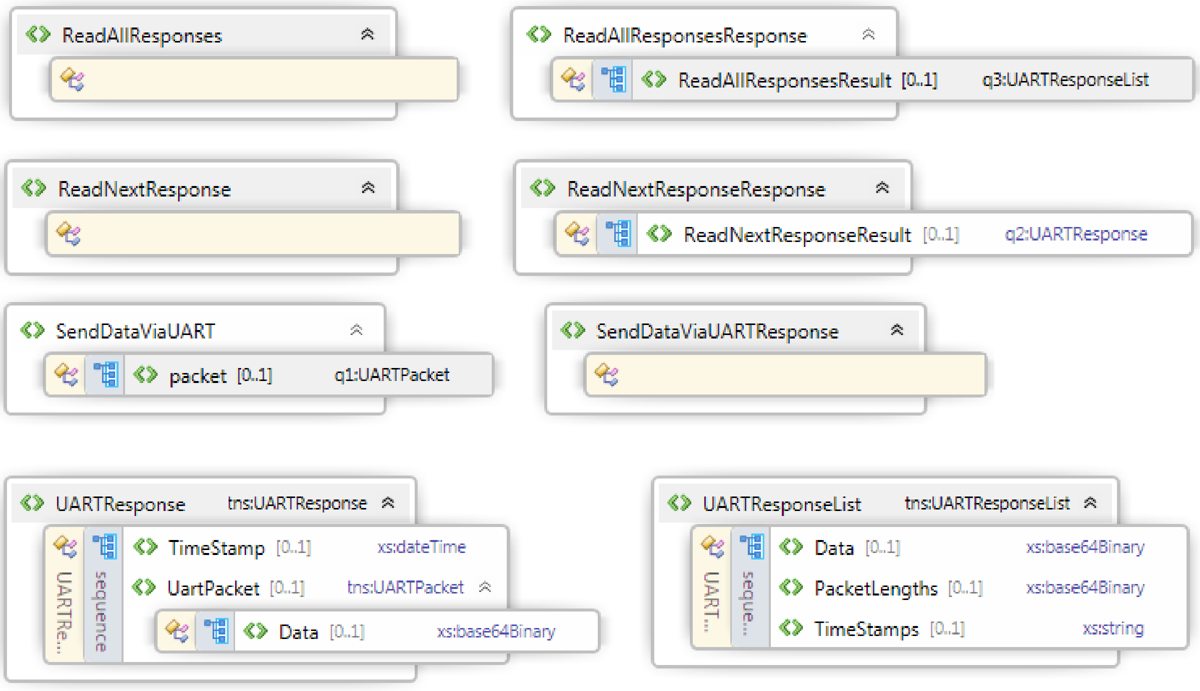

ReadAllResponses
ReadAllResponsesResponse
ReadAllResponsesResult [0..1] q3:UARTResponseList
ReadNextResponse
ReadNextResponseResponse
ReadNextResponseResult [0..1] q2:UARTResponse
SendDataViaUART
packet [0..1] q1:UARTPacket
SendDataViaUARTResponse
UARTResponse tns:UARTResponse
UARTRe...
sequence
TimeStamp [0..1] xs:dateTime
UartPacket [0..1] tns:UARTPacket
Data [0..1] xs:base64Binary
UARTResponseList tns:UARTResponseList
UART...
seque...
Data [0..1] xs:base64Binary
PacketLengths [0..1] xs:base64Binary
TimeStamps [0..1] xs:string

Figure 13 - UARTExtensionService XSD

Service Operations

To read a single message from the `receptionQueue`, call `ReadNextResponse` without any parameters. If the call succeeds, it will return a `UARTResponse` object that basically consists of a timestamp (time of reception on the HCU) and a byte array (the actually received message in its raw form). In most cases it will then be reasonable to translate that raw message back into an `MTGenericPacket` to read its command-field and determine the exact message type.

To reduce the number of service calls, the `UARTExtensionService` also provides an operation for reading all pending messages on the queue at once – call `ReadAllResponses` without any parameters. The most reasonable return-value would have been an array of `UARTResponse` objects. Unfortunately, there are some bugs for generating the service code for a Micro Framework device. Especially for *nested arrays*, it is often impossible to get the generated code to work, even when manipulating the generated code afterwards (which is very undesirable anyway).

Therefore, the call will return a `UARTResponseList` object, where all M&T packets are appended to a single byte array. To be able to separate the messages again, the individual packet lengths are also transmitted in a separate array. Since `DateTime` arrays are also problematic, the timestamps are packed into a single string, where each timestamp is separated by a newline. For more details, see the in-code documentation of the service implementation.

The current service-implementation only allows sending single commands with the `SendDataViaUART` operation, which takes a `UARTPacket` as a parameter and has no return value.

Additional Details

To generate the service code for the Micro Framework, a tool called *"MfSvcUtil.exe"* can be used. The tool comes with the .NET Micro Framework SDK and basically takes a WSDL-file or an URL to the location of it and generates wrapper-classes for the service operations and the involved data types (XML packing/unpacking). More information on the tool can be found in the Micro Framework SDK documentation, on MSDN or on various tutorials. The MFZStackApp-project contains a batch file (*generate.bat*) which will automatically re-generate the service code.

> *The address of the web service (URN) will also be randomly regenerated when generating the service code. Make sure to either overwrite it with the old address or update the address in the GUI applications.*

To produce the code for service clients, one can simply use a so called *Service Reference* – for an example see one of the two GUI-projects. Alternatively, a tool called "*SvcUtil.exe"* can be used.

Transmitting byte arrays in SOAP messages is not straightforward, since a SOAP message is essentially an XML document where the "data" may not contain XML-characters. For a byte-array, one cannot ensure that it does not contain such characters (in fact, the byte array could be an XML document itself). Therefore, SOAP uses a mechanism called *Base64-Encoding* where the byte-array gets encoded into valid character sequences (64 different characters: letters, numbers and a few special characters). This encoding introduces an inherent overhead of one third of the array's size – three bytes are being translated into four characters.

This mechanism is unsuitable for transmitting large byte arrays – for this purpose there is another kind of message type – MTOM encoded messages. MTOM allows the definition of messages with multiple body parts – a body part can also contain any kind of binary data. The first body part is usually the original SOAP message, but instead of adding byte arrays directly, references to other body parts are used.

The web service stack on the Micro Framework seems to have problems with parsing large SOAP-messages, but not with MTOM-encoded messages.

3.2.5 ZigBee Control Center

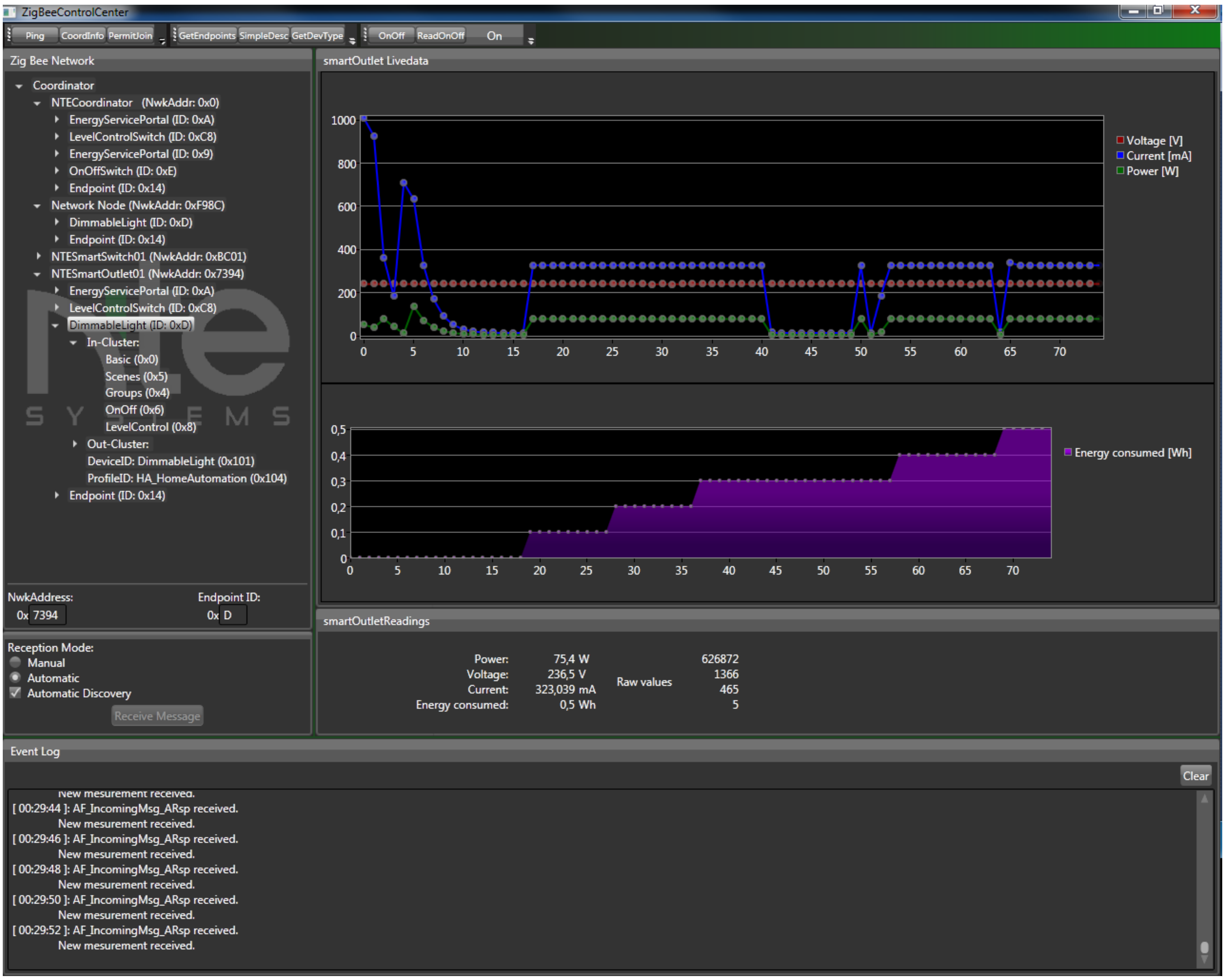
The ZigBee Control Center is the GUI application and provides an interface for interaction with the ZigBee network. A screenshot of the GUI can be seen in Figure 14 – for more details on the displayed data and the possibilities for interaction see the GUI handbook in A.3.

The application basically polls the HCU for received messages from the coordinator by repeatedly calling `ReadAllResponses` via the web service interface. If the call returns any messages, they will be processed sequentially. If the user triggers a command (e.g. by using one of the buttons), it will immediately be sent to the HCU via the web service interface – no queuing of messages to send.

Also worth noticing is the fact, that the application heavily uses the `MTCommands` namespace, which is defined in the ZStackCOM project – a project targeted at the .NET Micro Framework. Even though the GUI uses the "full" .NET framework, there have been no compatibility issues throughout the whole project – code written for the Micro Framework will also work on the full framework and vice versa (of course one has to avoid classes that are not supported "on the other side").

In the current scenario, the GUI also keeps a simple model of the network. Due to the problems with the web service interface it seemed more reasonable to transmit the raw messages to the GUI application, instead of having a network representation on the Micro Framework device and transmitting this representation (because the involved web service operations would have certainly required more complex data types – which are still problematic for the code generation). Another reason is, that the GUI has to keep a model of the network anyway, since it is being displayed in a *TreeView* (can be seen at the left hand side of the GUI). According to the *Model-View-Controller (MVC)* pattern, the data should be represented in a separate model (the view-model).

> *The GUI as well as the web service interface can be seen as an example use of the ZStack-COMLibrary and a demonstration of functionality. Both parts will be replaced by more sophisticated solutions in later development stages of the HCU system. However, they do complete this POC project and provide all the required functionality and while having some known technical weaknesses the implementations do more than just serving their purpose.*



Zig Bee Network

- Coordinator
  - NTECoordinator (NwkAddr: 0x0)
    - EnergyServicePortal (ID: 0xA)
    - LevelControlSwitch (ID: 0xC8)
    - EnergyServicePortal (ID: 0x9)
    - OnOffSwitch (ID: 0xE)
    - Endpoint (ID: 0x14)
  - Network Node (NwkAddr: 0xF98C)
    - DimmableLight (ID: 0xD)
    - Endpoint (ID: 0x14)
  - NTESmartSwitch01 (NwkAddr: 0xBC01)
  - NTESmartOutlet01 (NwkAddr: 0x7394)
    - EnergyServicePortal (ID: 0xA)
    - LevelControlSwitch (ID: 0xC8)
    - DimmableLight (ID: 0xD)
      - In-Cluster:
        - Basic (0x0)
        - Scenes (0x5)
        - Groups (0x4)
        - OnOff (0x6)
        - LevelControl (0x8)
      - Out-Cluster:
      - DeviceID: DimmableLight (0x101)
      - ProfileID: HA_HomeAutomation (0x104)
    - Endpoint (ID: 0x14)

NwkAddress: 0x 7394
Endpoint ID: 0x D

Reception Mode:
Manual
[✓ Automatic]
[✓ Automatic Discovery]

smartOutletReadings

Power:	75,4 W	Raw values	626872
Voltage:	236,5 V		1366
Current:	323,039 mA		465
Energy consumed:	0,5 Wh		5

Event Log

```
        New mesurement received.
[ 00:29:44 ]: AF_IncomingMsg_ARsp received.
        New mesurement received.
[ 00:29:46 ]: AF_IncomingMsg_ARsp received.
        New mesurement received.
[ 00:29:48 ]: AF_IncomingMsg_ARsp received.
        New mesurement received.
[ 00:29:50 ]: AF_IncomingMsg_ARsp received.
        New mesurement received.
[ 00:29:52 ]: AF_IncomingMsg_ARsp received.
        New mesurement received.
```

Figure 14 - Screenshot of the Zig Bee Control Center

4 Conclusions

4.1 Summary

The overall aim of this project was "*getting to know ZigBee*". As NTE plans to integrate the technology in their future products, it was of special interest to explore the ZigBee domain with a project that covers some of the **future use cases** and provides a **realistic insight** to the advantages but also the problems when working with this technology. To cover the aspects of hardware-development, a parallel project has been conducted. The results of that project are four device-prototypes: the smartLight, the smartSwitch, the smartOutlet and a coordinator device (see 2.1.2 for details on the devices and [3] for the project's documentation) as well as a lot of know-how on integrating ZigBee hardware.

These hardware-prototypes have been completed with this software project that ultimately provides a graphical user interface for wirelessly monitoring and controlling the ZigBee network nodes.

The most obvious outcome, of course, is a functional prototype system that fulfills all the defined requirements for this POC (see 3.1.2). This gives a perspective on the possible "look and feel" of a future implementation. It helps estimating aspects like the response time of nodes under real conditions, the process of monitoring the network state or the amount of information that can be retrieved from a network node. It is also well suited for demonstrating an interactive *technological preview*.

A second achievement is the implementation of a comprehensive library for encapsulating the ZigBee communication. This library provides a solid basis for future integration into related projects and can easily be extended to fit the needs that have not yet been covered. This is somewhat unusual for a proof of concept, since a POC usually denotes a "disposable" solution, whose only purpose is to proof a certain concept (or proof it wrong). On the other hand one usually speaks of a *prototype* when referring to an early project that is used as a basis for further developments. However, the boundaries between the two definitions are blurred; the biggest danger is "keeping a project that was supposed to be disposable, because it works so well".

In the current project the web service interface and the ZigBee Control Center do "work quite well" but they have never been intended to be reused – therefore they lack the corresponding design and carefulness during implementation. The `ZStackCOM` library has always had the focus to be the basis for future extensions and thus has a sophisticated design and a well thought implementation. Another issue is the fact that the involved data structures and mechanisms are very well defined in the M&T API and the ZigBee specifications. Most parts of the library are simply an implementation of these definitions. So the `ZStackCOM` library can be seen as a prototype whereas the other software-parts have a proof of concept character.

Other findings of the project are more subtle but nonetheless important – they concern the revelation of strengths and weaknesses of certain approaches, the compatibility of certain paradigms or design principles with the ZigBee domain, ... - in a word: experience. Also, the results of this project can help new developers to set foot in the technological domain faster and steepen their learning curve.

4.1.1 Known Issues

The biggest weakness of the current POC system is probably the web service interface. One reason is that the code generation is buggy and therefore the web service operation `ReadAllResponses` has a (to say the least) non-intuitive design (see. 3.2.4 for more details). The other problem is that the implemented web service stack on the Micro Framework shows some strange behavior. Requests are typically processed within a few hundred milliseconds (90ms –

250ms). Occasionally, a request will take a few seconds (!) to be processed. Since service calls are blocking, this is quite inconvenient. Solving these issues is part of ongoing work at NTE Systems; however, it requires modifications within the source code of the Micro Framework and is therefore non-trivial.

Another issue with the web service interface is that it will remove a message from the queue of received messages during a service call. If that service call fails for some reason, the message is lost. A safer approach would involve a *second* service call for actually removing the message from the queue *after* it has been received successfully.

Another drawback of the current implementation is the lack of a test bed for the `ZStackCOMLibrary`. Even though the library has been used quite a lot and proven to be stable, this can never replace a test bed. Before extending the library or using it in a non-prototype project, it has to be extensively tested. Since the library has a lot of classes with properties that read or write on the underlying byte-array, it is cumbersome to create an individual test case for every property. One might even consider, writing a simple test case generator to automatically generate these tests.

But also the `COMHandler` has to be tested thoroughly – under no circumstances must it drop UART messages or parts of it. Especially with the handling of message stubs (3.2.2) all possible cases must be tested, since they involve precise timing and are therefore hard to produce under real usage conditions.

In this project, a single library for handling M&T- and ZigBee-communication has been implemented. The ZigBee commands are the payload of a special kind of M&T messages. One therefore might think about pulling these separate parts into separate libraries. In the process of doing so, it might also be reasonable to further encapsulate the M&T message-infrastructure within a manufacturer independent layer (M&T is specific to Texas Instruments' products). However, since every manufacturer has a slightly different API, this might involve a significant amount of work. Therefore, one first has to put some thought into manufacturer-dependency, because also the hardware-transceivers cannot be exchanged that easily. If the choice is, to stick with one manufacturer, then the workload for adding a manufacturer-independent layer to the library is hard to justify.

4.1.2 Lessons Learned

ZigBee is very powerful and flexible – it fits to a broad range of application scenarios. It can be adapted to many paradigms and supports many different design principles. For instance, the reporting-mechanism can be used in push- or pull-architectures (polling or event-driven) and even a mixture of both cases can be reasonable. ZigBee has been well-designed and has undergone a long process of specification development (ZigBee standards have been proposed before any hard- or software products were available and not the other way round).

This flexibility comes with a certain cost when having to reproduce the same flexibility in your product. For a hardware-product this usually means using a third-party transceiver chip. For software projects the flexibility has to be encapsulated by a clever and sophisticated architecture and design.

On the other hand, ZigBee can also be very specific. The public profile definitions are vast and exhaustive. To ensure compatibility with a full profile, a significant amount of workload is needed. This concerns mostly software-products, since a hardware device will usually only use a very small subset of a profile.

A nice side-result is that sharing code or even a complete library between a project for the .NET Micro Framework and a "full" .NET Framework project was really unproblematic. At least as long as basic types and classes (and compositions of these basic types) were used, there were no problems at all. Namespaces and naming conventions have been migrated carefully to the

Micro Framework. Of course, one has to take care, not to "accidentally" include or use any types that are only available one of both platforms.

4.2 Possible Next Steps

As already mentioned the implementation of a test bed for the `ZStackCOMLibrary` is the most obvious next step and cannot be omitted, if the library shall be used in a product. The separation of the library into multiple projects and the addition of a manufacturer-independent layer might also be among the immediate next steps.

The Micro Framework device should keep an internal representation of the ZigBee network up to date. In the current project this implementation has been moved to the GUI-client, due to issues with the web service interface (see 3.2.5). The structure of such a model is more or less given by the ZigBee specifications. An example (extracted from the model of the GUI) is shown in Figure 15.

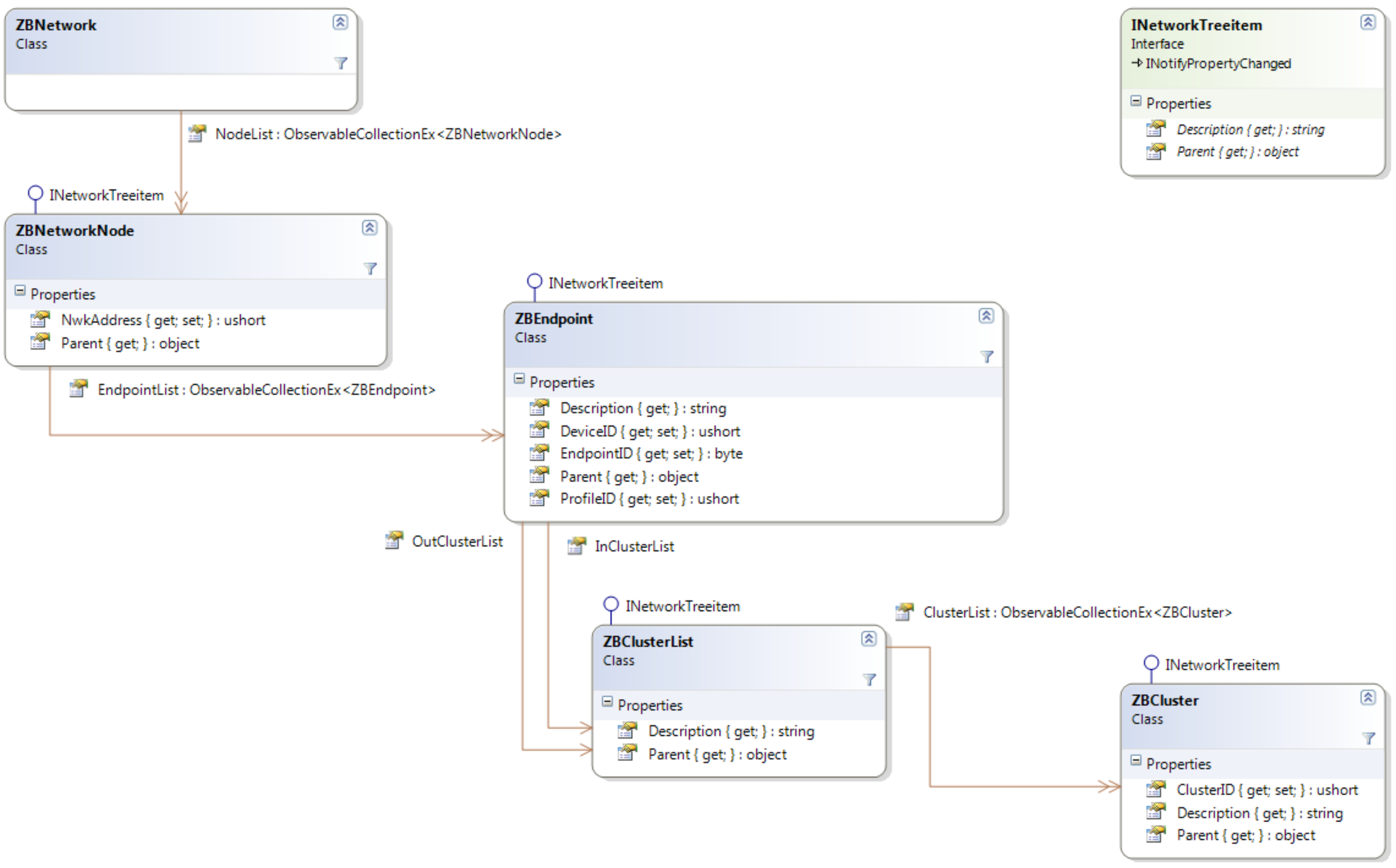
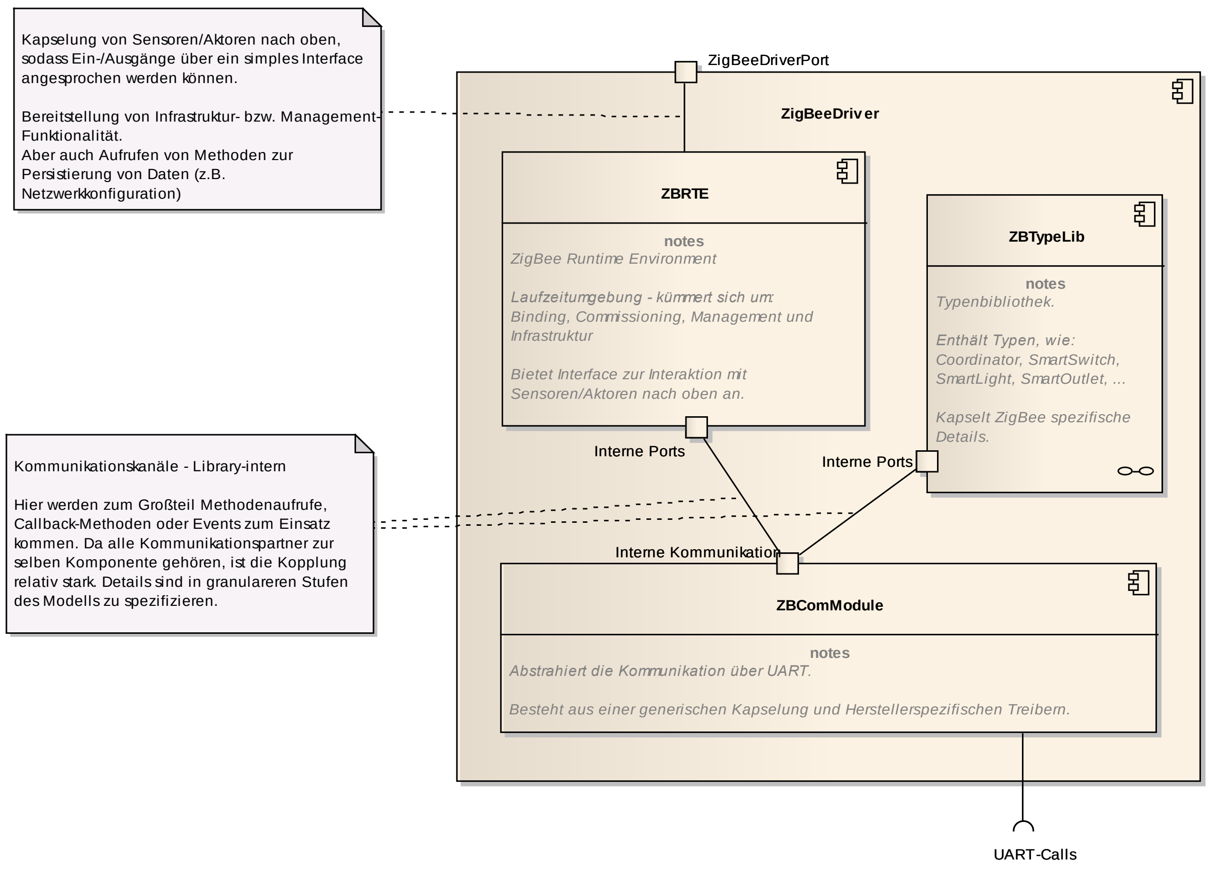

ZBNetwork
Class
NodeList : ObservableCollectionEx<ZBNetworkNode>
INetworkTreeitem
ZBNetworkNode
Class
Properties
NwkAddress { get; set; } : ushort
Parent { get; } : object
EndpointList : ObservableCollectionEx<ZBEndpoint>
INetworkTreeitem
ZBEndpoint
Class
Properties
Description { get; } : string
DeviceID { get; set; } : ushort
EndpointID { get; set; } : byte
Parent { get; } : object
ProfileID { get; set; } : ushort
OutClusterList
InClusterList
INetworkTreeitem
ZBClusterList
Class
Properties
Description { get; } : string
Parent { get; } : object
ClusterList : ObservableCollectionEx<ZBCluster>
INetworkTreeitem
ZBCluster
Class
Properties
ClusterID { get; set; } : ushort
Description { get; } : string
Parent { get; } : object
INetworkTreeitem
Interface
→ INotifyPropertyChanged
Properties
Description { get; } : string
Parent { get; } : object

Figure 15 - ZigBee Network Model Structure

The network model consists of nested lists with a depth of three. The ZigBee network has a list of network nodes, where each node has a list of endpoints. Each endpoint holds an In- and an Out-Cluster list – both of them are lists of cluster objects. The `INetworkTreeItem` interface shown in Figure 15 is part of the view model, but it also shows the general property of all ZigBee classes *to have a single parent object* (except for the network).

On a larger scale it might be desirable to encapsulate functional entities like the smartLight, the smartSwitch, the smartOutlet and the coordinator. The current implementation of the library does not distinguish between different types of network nodes. As far as a later system integration is concerned, a further layer of abstraction has to be introduced. E.g. have a smartLight-object with a property of the switching state (that can also be set to change the state). The implemented library provides the basis for such types with the M&T- and ZigBee commands. However, on a higher level of abstraction, these commands must be of no concern.

In the course of this project, a few architectural considerations for further development and parts of the system integration have been made. When integrating ZigBee into the existing ECU/HCU firmware, the transceiver must be transparent and ZigBee devices must be encapsulated up to a level of abstraction where they “behave (almost) like directly connected I/O’s”. Similar to an I/O driver (for instance a PWM driver), the ZigBee functionality should be packaged into a *ZigBee Driver* component. A proposal for the structure of such a component can be seen in Figure 16 (in German).


Kapselung von Sensoren/Aktoren nach oben, sodass Ein-/Ausgänge über ein simples Interface angesprochen werden können.

Bereitstellung von Infrastruktur- bzw. Management-Funktionalität.
Aber auch Aufrufen von Methoden zur Persistierung von Daten (z.B. Netzwerkkonfiguration)

ZBRTE notes: ZigBee Runtime Environment

Laufzeitumgebung - kümmert sich um: Binding, Commissioning, Management und Infrastruktur

Bietet Interface zur Interaktion mit Sensoren/Aktoren nach oben an.

ZBTypeLib notes: Typenbibliothek.

Enthält Typen, wie: Coordinator, SmartSwitch, SmartLight, SmartOutlet, ...

Kapselt ZigBee spezifische Details.

Kommunikationskanäle - Library-intern

Hier werden zum Großteil Methodenaufrufe, Callback-Methoden oder Events zum Einsatz kommen. Da alle Kommunikationspartner zur selben Komponente gehören, ist die Kopplung relativ stark. Details sind in granulareren Stufen des Modells zu spezifizieren.

ZBComModule notes: Abstrahiert die Kommunikation über UART.

Besteht aus einer generischen Kapselung und Herstellerspezifischen Treibern.

Figure 16 - ZigBee Driver Architecture (in German)

The Driver basically consists of three components:

- ZBComModule: Encapsulates the communication via UART and provides means to send and receive ZigBee commands. It mainly consists of a communication handler component and (manufacturer-specific) command structures, wrapped by a manufacturer-independent layer.
- ZBTypeLib: A type library, providing higher-level abstractions such as classes for the smartLight, the smartOutlet, the smartSwitch and the coordinator, but also for the general ZigBee network types (node, endpoint, cluster ...). The level of abstraction of these types should already hide the underlying message/command exchange.
- ZBRTE: A runtime environment, which is the “*active*” component of the driver. It is mainly concerned with keeping the internal network representation up to date, but it also has to manage bindings, coordinate the commissioning, etc.

These architectural considerations can be found in the project's architectural model (created with the Enterprise Architect™). However, they are still very coarse and have to be refined in consecutive design stages.

4.3 Résumé

Personally, I've had several factors of motivation for this project. On one hand, my work at NTE System has always been fruitful and it has allowed me to get to know a professional industrial working environment, which certainly is a contrast to my education at the university. Another reason is that the ZigBee technology has sparked my interest, as it allows for exciting technological visions. I personally believe that ZigBee is among the *enabling technologies* for upcoming technological developments that will diffuse into our everyday lives. A third, more practical aspect was my interest in the future of embedded development. With the .NET Micro Framework one can at least get a glimpse on a modern way to write embedded applications.

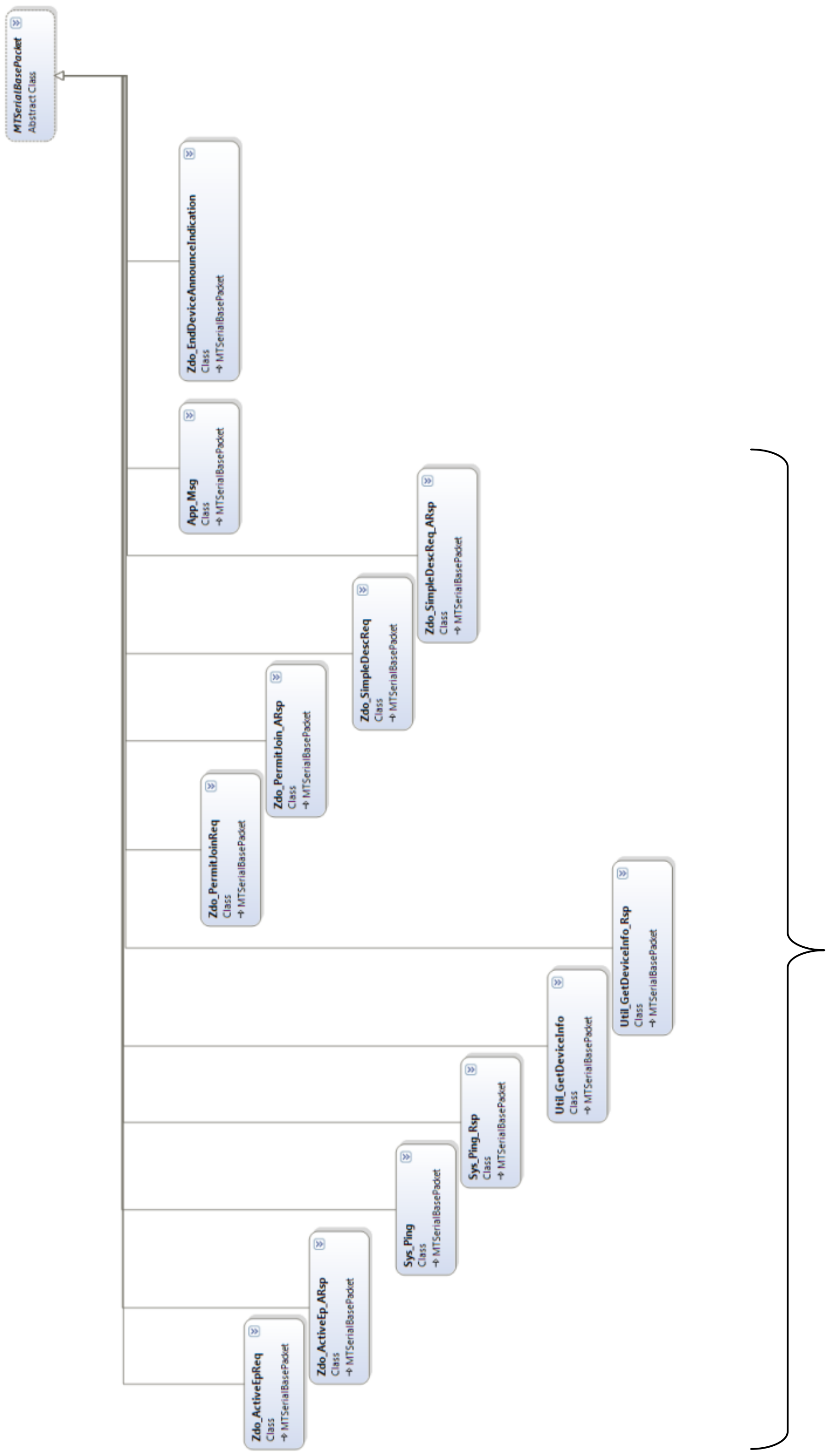
In my master's programme, I have set a focus on robotics. Due to that, I was also curious about the impact of ZigBee for that domain. Since ZigBee has a low-power design principle and requires only little computational resources, it seems suitable for an adoption into robotic scenarios. Furthermore, ZigBee can provide an enriched environment for an autonomous agent. Imagine a house, packed with ZigBee devices. A (household-) robot could easily draw a rich context out of the information that he can retrieve of these devices. For instance, by using information about electrical lighting and running devices (such as a TV or a stereo), a vacuum-cleaner robot could prefer rooms where no person is in right now. One could also equip all potted plants with wireless watering indicators and have a household robot taking care of the watering.
The technology becomes even more interesting when considering the information that can be inferred from a smart home. The field of *Ambient Assisted Living (AAL)* aims at assisting elderly people in their homes with the use of state of the art technology. One important goal would be the detection of "irregularities" in the daily routines and contacting one of the family members or even calling an ambulance. ZigBee devices do perfectly fit into such scenarios, because they are inexpensive and seamlessly ***blend*** into our everyday life (having a surveillance camera in every room is less acceptable than simply having ZigBee switches, lights and other devices that simply enrich existing household appliances).

The project was conducted over a period of six months with a weekly working time varying from one to three days. In retrospect this created the illusion of a quite slow pace of work – even though the net-pace was quite good, it seemed as if progress was kind of slow (considering only the net-working time, the project could have been conducted in two months of full-time work).

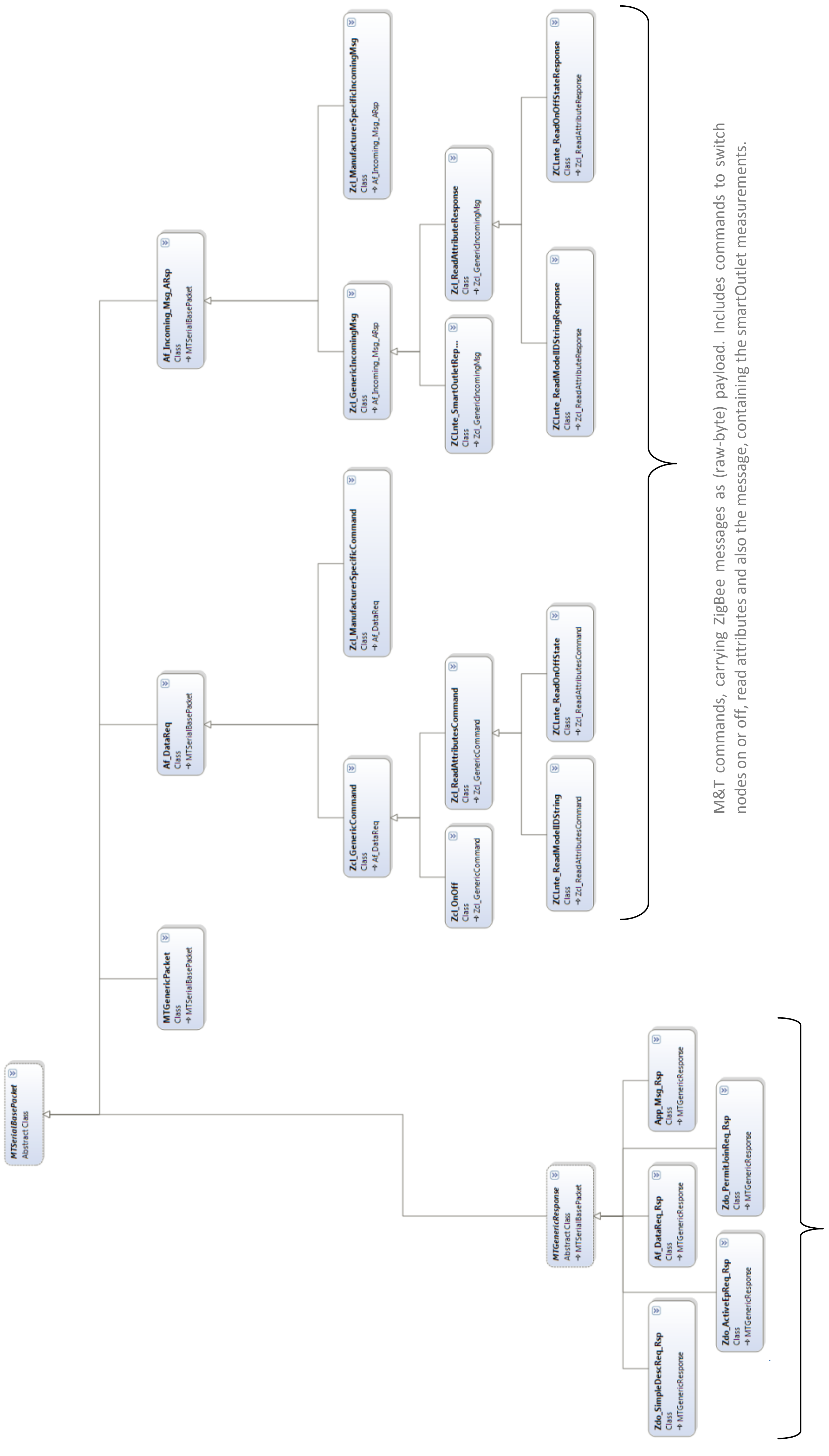
In the course of this project, I was able to achieve the goals, stated above – I have gained a lot of practical experience in a professional environment; I have been able to improve my .NET Micro Framework skills and I have been gained quite some insight on the ZigBee technology.

A. Appendix

A.1 The MTCommands namespace



M&T commands – including requests and (asynchronous) responses for the device info, the active endpoints on a node, the simple descriptor of an endpoint and the command to open the join-window.



M&T commands, carrying ZigBee messages as (raw-byte) payload. Includes commands to switch nodes on or off, read attributes and also the message, containing the smartOutlet measurements.

Synchronous M&T responses, containing information on whether the request has been processed successfully or not.

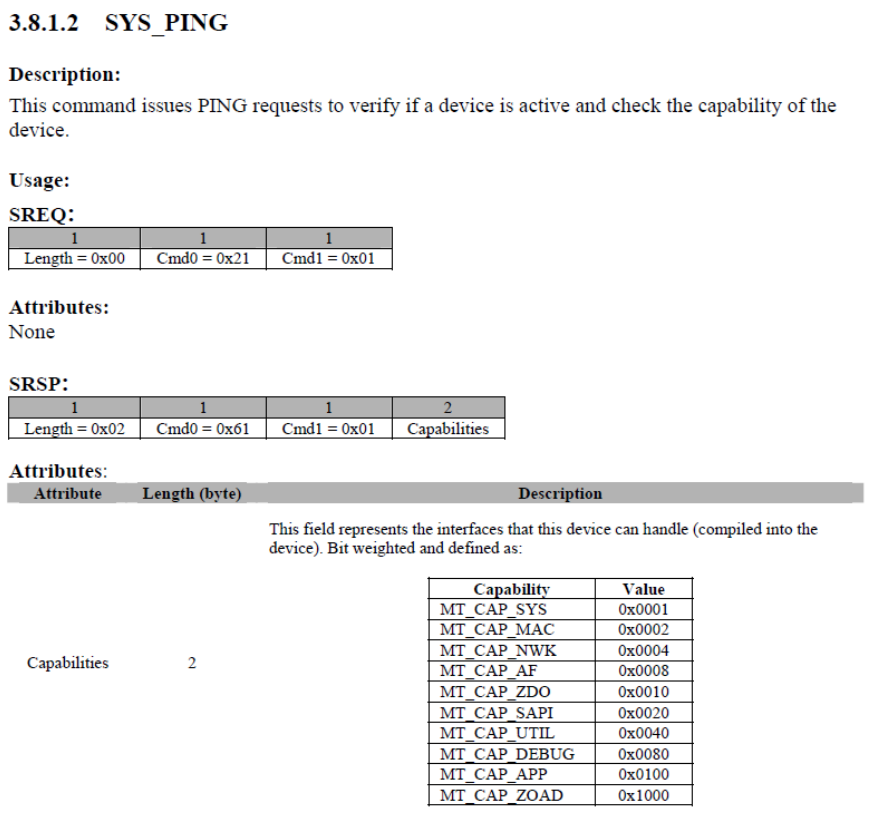
A.2 Extending the command infrastructure

This is a brief tutorial on extending the command infrastructure, i.e. the `MTCommands` namespace. Before actually writing any code, **make yourself familiar** with the `MTCommands` namespace. An overview of all its classes and the derivation relationships can be seen in A.1. A discussion of the most important classes can be found in 3.2.3. Make sure to find an appropriate base class for your new command.

If you are writing a new ZigBee command, you should probably derive from `Zcl_GenericCommand` or `Zcl_GenericIncomingMessage`. To see a usage example, you can then take a look at one of the derived classes. Your main source of information for that new command will then be the ZCL specification [5] or one of the public profile definitions, like [6] or [7].

If you are implementing a new M&T command, you should at least derive from `MTSerialBasePacket`, perhaps you can even find a more suitable base class. Your main source of information in that case will be the M&T API [8].

Let's take a look at the implementation of a very simple command – the Ping command. Since it is an M&T command, we will take a look at the M&T API. On page 44 we can find the specifications for the command and the response – see Figure 17.



3.8.1.2 SYS_PING

Description:

This command issues PING requests to verify if a device is active and check the capability of the device.

Usage:

SREQ:

1	1	1
Length = 0x00	Cmd0 = 0x21	Cmd1 = 0x01

Attributes:
None

SRSP:

1	1	1	2
Length = 0x02	Cmd0 = 0x61	Cmd1 = 0x01	Capabilities

Attributes:

Attribute	Length (byte)	Description
Capabilities	2	This field represents the interfaces that this device can handle (compiled into the device). Bit weighted and defined as:

Capability	Value
MT_CAP_SYS	0x0001
MT_CAP_MAC	0x0002
MT_CAP_NWK	0x0004
MT_CAP_AF	0x0008
MT_CAP_ZDO	0x0010
MT_CAP_SAPI	0x0020
MT_CAP_UTIL	0x0040
MT_CAP_DEBUG	0x0080
MT_CAP_APP	0x0100
MT_CAP_ZOAD	0x1000

Figure 17 - Definition of SYS_PING in M&T API ([8])

Taking a closer look, we see that the command consists of a synchronous request (SREQ) with a data length of 0x00 and the command 0x2101. A data length of zero means that we have to specify no parameters. Therefore, it seems appropriate to derive directly from the `MTSerialBasePacket`. We do this with the following line:

```
public class Sys_Ping : MTSerialBasePacket
```

Next, we have to make sure to specify the correct command. This is done by overriding a property of the (abstract) base class.

```
        protected override CMDEnums.CMDEnum CommandEnum
        {
            get { return CMDEnums.CMDEnum.SYS_Ping; }
        }
```

The `CMDEnum` already has a definition for `SYS_Ping` which is 0x2101. If your command is not yet defined in that enumerator, make sure to **add it** there.

Now, all that's left are the constructors for our new class. See the complete code in Code-fragment 4. We have one parameter-less constructor, that tells the constructor of the base-class that the data-length is 0x00 bytes. And we have a constructor that takes a raw byte-array and calls the corresponding constructor of the base class.

```
namespace ZStackCOM.MTCommands.SYSCommands
{
    /// <summary>
    /// Implementation of SYS_PING command
    /// </summary>
    public class Sys_Ping : MTSerialBasePacket
    {
        protected override CMDEnums.CMDEnum CommandEnum
        {
            get { return CMDEnums.CMDEnum.SYS_Ping; }
        }

        public Sys_Ping()
            : base(0x00)
        {
        }


        public Sys_Ping(byte[] rawData)
            : base(rawData)
        {
        }

    }
}
```

Code-fragment 4 - Definition of the SYS_Ping Command

In the second step, we want to implement the response to SYS_Ping. The M&T API states that we will get a synchronous response (SRSP) with a data length of 0x02 bytes and a command identifier of 0x6101 (see Figure 17).

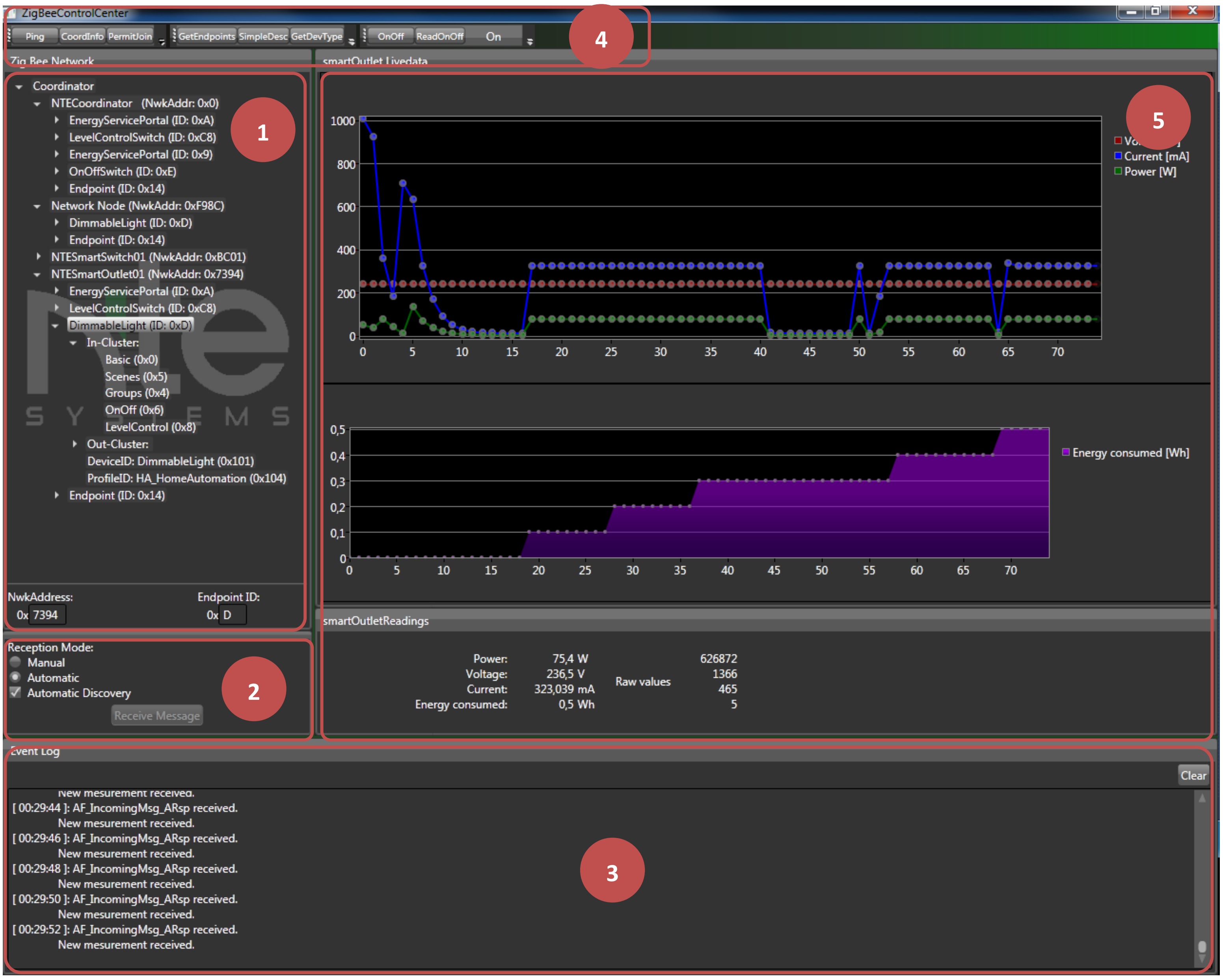
As before, we have to make sure that the corresponding command is defined in the `CMDEnum`. Then we derive a new class (again `MTSerialBasePacket` is the most appropriate base class). We need to provide more or less the same two constructors, but this time we have to tell the base class to allocate two bytes (0x02) for the data. The API definition also tells us something about the content of the data – it is more or less a mapping of Boolean variables to the individual data-bits. For our implementation, we have defined a simple type (`MTCapabilities`) that contains all these Boolean properties. To access the results of the response, we write a method which returns such an `MTCapabilities` object where the data-bits have been mapped back to the corresponding Boolean properties. The complete source code can be found in Code-fragment 5.

```
namespace ZStackCOM.MTCommands.SYSCommands
{
    /// <summary>
    /// Implementation of SYS_PING RESPONSE
    /// </summary>
    public class Sys_Ping_Rsp : MTSerialBasePacket
    {

        protected override CMDEnums.CMDEnum CommandEnum
        {
            get { return CMDEnums.CMDEnum.SYS_Ping_Response; }
        }


        #region constructors
        public Sys_Ping_Rsp()
            : base(0x02)
        {
        }

        public Sys_Ping_Rsp(byte[] rawData)
            : base(rawData)
        {
        }
        #endregion

        #region methods
        /// <summary>
        /// Returns an object, describing the enabled interfaces of the MT-device.
        /// </summary>
        /// <returns>Object, describing status of the MT-interfaces on the device.</returns>
        public MTCapabilities GetMTCapabilities()
        {
            MTCapabilities cap = new MTCapabilities();
            cap.SYSInterfaceEnabled = (Data[0] & 0x01) > 0;
            cap.MACInterfaceEnabled = (Data[0] & 0x02) > 0;
            cap.NWKInterfaceEnabled = (Data[0] & 0x04) > 0;
            cap.AFInterfaceEnabled = (Data[0] & 0x08) > 0;
            cap.ZDOInterfaceEnabled = (Data[0] & 0x10) > 0;
            cap.SAPIInterfaceEnabled = (Data[0] & 0x20) > 0;
            cap.UTILInterfaceEnabled = (Data[0] & 0x40) > 0;
            cap.DEBUGInterfaceEnabled = (Data[0] & 0x80) > 0;
            cap.APPInterfaceEnabled = (Data[1] & 0x01) > 0;
            cap.ZOADInterfaceEnabled = (Data[1] & 0x10) > 0;

            return cap;
        }
        #endregion

    }
}
```

Code-fragment 5 - Definition of the SYS_Ping Response

This example has demonstrated the implementation of a very simple command. All the source code involved can be found in the `ZStackCOM` project in the `MTCommands.SYSCommands` namespace. More sophisticated commands are based on the same principles – mapping data-bytes to properties or objects returned by methods. It is suggested to take your time to familiarize yourself with the implemented commands, since they cover a broad range of different cases.

A.3 Handbook for the ZigBee Control Center GUI



- Coordinator
  - NTECoordinator (NwkAddr: 0x0)
    - EnergyServicePortal (ID: 0xA)
    - LevelControlSwitch (ID: 0xC8)
    - EnergyServicePortal (ID: 0x9)
    - OnOffSwitch (ID: 0xE)
    - Endpoint (ID: 0x14)
  - Network Node (NwkAddr: 0xF98C)
    - DimmableLight (ID: 0xD)
    - Endpoint (ID: 0x14)
  - NTESmartSwitch01 (NwkAddr: 0xBC01)
  - NTESmartOutlet01 (NwkAddr: 0x7394)
    - EnergyServicePortal (ID: 0xA)
    - LevelControlSwitch (ID: 0xC8)
    - DimmableLight (ID: 0xD)
      - In-Cluster:
        - Basic (0x0)
        - Scenes (0x5)
        - Groups (0x4)
        - OnOff (0x6)
        - LevelControl (0x8)
      - Out-Cluster:
      - DeviceID: DimmableLight (0x101)
      - ProfileID: HA_HomeAutomation (0x104)
    - Endpoint (ID: 0x14)

NwkAddress: 0x 7394 Endpoint ID: 0x D

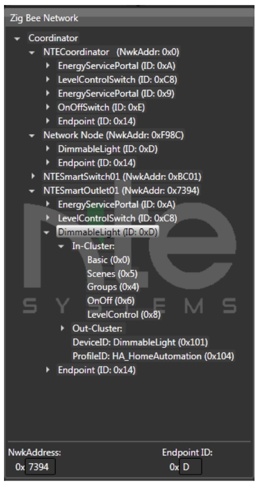
Reception Mode:
- Manual
- Automatic
- [✓ Automatic Discovery]

Power:	75,4 W	Raw values	626872
Voltage:	236,5 V		1366
Current:	323,039 mA		465
Energy consumed:	0,5 Wh		5

```
        New mesurement received.
[ 00:29:44 ]: AF_IncomingMsg_ARsp received.
        New mesurement received.
[ 00:29:46 ]: AF_IncomingMsg_ARsp received.
        New mesurement received.
[ 00:29:48 ]: AF_IncomingMsg_ARsp received.
        New mesurement received.
[ 00:29:50 ]: AF_IncomingMsg_ARsp received.
        New mesurement received.
[ 00:29:52 ]: AF_IncomingMsg_ARsp received.
        New mesurement received.
```

Figure 18 - Functional parts of the Control Center GUI

1

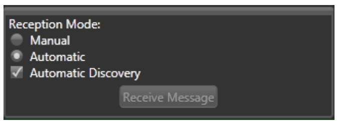


```
Zig Bee Network
▾ Coordinator
  ▾ NTECoordinator  (NwkAddr: 0x0)
    ▸ EnergyServicePortal (ID: 0xA)
    ▸ LevelControlSwitch (ID: 0xC8)
    ▸ EnergyServicePortal (ID: 0x9)
    ▸ OnOffSwitch (ID: 0xE)
    ▸ Endpoint (ID: 0x14)
  ▾ Network Node (NwkAddr: 0xF98C)
    ▸ DimmableLight (ID: 0xD)
    ▸ Endpoint (ID: 0x14)
  ▸ NTESmartSwitch01 (NwkAddr: 0xBC01)
  ▾ NTESmartOutlet01 (NwkAddr: 0x7394)
    ▸ EnergyServicePortal (ID: 0xA)
    ▸ LevelControlSwitch (ID: 0xC8)
    ▾ DimmableLight (ID: 0xD)
      ▾ In-Cluster:
          Basic (0x0)
          Scenes (0x5)
          Groups (0x4)
          OnOff (0x6)
          LevelControl (0x8)
      ▸ Out-Cluster:
        DeviceID: DimmableLight (0x101)
        ProfileID: HA_HomeAutomation (0x104)
    ▸ Endpoint (ID: 0x14)

NwkAddress:            Endpoint ID:
 0x 7394                0x D
```

ZigBee network tree view:

This visual control will display information about the network nodes. It shows a hierarchical tree with the nodes (corresponding to a physical device) on the first layer. Every node can have a list of endpoints, shown on the second layer. For each endpoint, the tree shows the profile- and device-ID as well as the in- and out-cluster lists on the third layer.
The network address of a node can be seen in brackets behind the node's name. Endpoint- and cluster-IDs will be resolved into a descriptive string if possible, with the raw value shown in brackets.
Note that this information may only be partially available, if the corresponding readouts have not yet been executed.
At the bottom of the control, the currently selected node-address and endpoint-ID can be seen. These values will be used when executing commands. Both fields are editable, to make it possible to enter custom values.

2

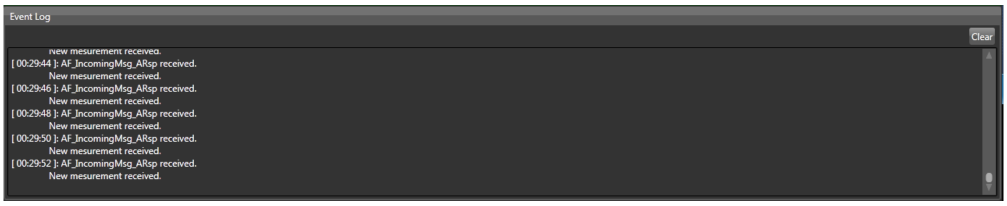


Reception Mode:
- ○ Manual
- ● Automatic
- ☑ Automatic Discovery

Reception modes:

When starting up the Control Center, the reception mode will be set to manual, which means that messages received at the coordinator will not automatically be fetched. Instead, the button "Receive Message" can be used to fetch a single message (using a web service call). When the automatic mode is selected, all pending messages will be fetched periodically.
If CheckBox "Automatic Discovery" is checked, all information of a new network node will automatically be read out. Otherwise, one has to manually read the active endpoints and every endpoints' simple descriptor.

3

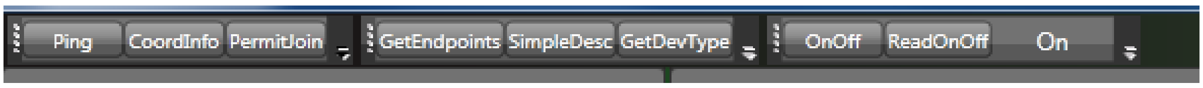


```
          New mesurement received.
[ 00:29:44 ]: AF_IncomingMsg_ARsp received.
          New mesurement received.
[ 00:29:46 ]: AF_IncomingMsg_ARsp received.
          New mesurement received.
[ 00:29:48 ]: AF_IncomingMsg_ARsp received.
          New mesurement received.
[ 00:29:50 ]: AF_IncomingMsg_ARsp received.
          New mesurement received.
[ 00:29:52 ]: AF_IncomingMsg_ARsp received.
          New mesurement received.
```

Event Log:

The bottom part of the GUI shows a rolling log, that will display information about ongoing events such as the sending or reception of messages and possibly additional information about certain received messages (such as the simple descriptor). The log will also show errors that occurred during operation.

4

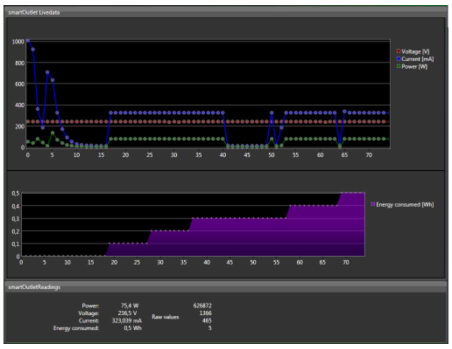


Menu Bar:

The menu-bar provides controls for interacting with the network nodes. The following table describes the function of every button.

Button	Description
Ping	Will send a ping-command to the coordinator to check if basic communication works. The coordinator will respond with a message, containing information about the enabled M&T interfaces. This information will be shown in the event log.
CoordInfo	Will retrieve the device information of the connected device (usually the connected device is the coordinator). If a coordinator device answers, it will be added to the tree view and if the automatic discovery is selected, the commands to read out more information will be triggered (same as manually clicking GetEndpoints and SimpleDesc for every endpoint). Besides the Ping-command, this should usually be the first action, before using any of the other buttons.
PermitJoin	This will open a window of (default) 10 seconds, where new devices are allowed to join the network. If a new device joins, a message box with information about it will pop up and if the automatic discovery is selected, the devices endpoints and simple descriptors will be read out. Using the collapse button to the right of the PermitJoin button, one can modify the join-time window. Valid values are zero to 255 (which means that the window will be kept open permanently).
GetEndpoints	This command will retrieve all active endpoints of the currently selected node. If the endpoints are not already displayed in the tree view, they will be added. When using automatic discovery, this button is obsolete as long as no service call fails.
SimpleDesc	This button will trigger the retrieval of the simple descriptor for the currently selected endpoint on the selected node. The simple descriptor contains a profile- and device-ID as well as the In- and Out-cluster lists. When using automatic discovery, this button is obsolete as long as no service call fails.
GetDevType	This command will read the manufacturer specific model string of a device and write that string into the tree view (as the node's name). This command only works when an endpoint with a "Basic" In-cluster is selected.
OnOff	This button can be used to toggle the switching state of a "switchable" device. The command can be sent to any endpoint, but will only have an effect if it is being sent to an endpoint with an "OnOff"-In-cluster. Use this button to switch the smartLight and smartOutlet on and off.
ReadOnOff	This button will read the current switching state of an OnOff device. The command will only work if it is being sent to an endpoint that has an "OnOff"-In-cluster (or also Out-cluster). The result of the operation will be displayed on the label right next to the button.

5



Power:	75,4 W		626872
Voltage:	236,5 V	Raw values	1366
Current:	323,039 mA		465
Energy consumed:	0,5 Wh		5

smartOutlet Measurements:

As soon as the smart Outlet has joined the network and started its measurements (usually after the first switch-on), it will periodically send its measurement values. These values will be displayed in the two charts (the upper one shows mains-voltage, load-current and the power consumption, the lower one shows the cumulated energy consumption over time).

Underneath the chart-area, the last received measurement value will also be displayed – showing the raw values of the ADC (analog digital converter) for debug-purposes as well as the converted values with its corresponding unit of measure.

Standard Workflow for the ZigBee Control Center:

1. Make sure that the right application has been deployed to the Micro Framework device and the device is powered up. Also make sure that the IP-Address of the device matches the one given in the source code (!) of the Control Center Application.
2. Make sure that the coordinator is connected via serial port to the MF device and is powered up as well.
3. Send a Ping command. If it succeeds change the mode of reception to automatic and see if the coordinator has responded to the ping.
4. Make sure the automatic discovery mode is checked.
5. Use CoordInfo to trigger the discovery of the coordinator node and its endpoints. The tree view should now (automatically) update the received information.
6. If one of the calls fails, you can use GetEndpoints, SimpleDesc or GetDevType to manually retrieve the information. Make sure to select (highlight) the right endpoint before.
7. Use PermitJoin to allow new devices to join the network. The join-window will be open for 10 seconds by default.
8. Switch on other ZigBee devices, if they are not running already.
9. You should observe a message box popping up for every device that has joined the network. If this fails for one or more devices, make sure that the permit-window is open and restart the devices.
10. If a router-device (mains-powered) has already joined the network, new network nodes will join via these device(s). In the current setup they are configured to permanently allow a join – even if the permit-join window of the coordinator is currently closed.
11. The control center has automatically read out the information on the new devices' endpoints and clusters. If this has partially failed, you can use the GetEndpoints, SimpleDesc and GetDevType buttons to manually retrieve the information. Make sure to select the right endpoint before invoking the operation.
12. You should now see your devices in the tree view. Select the right endpoints to interact with them (switching them on or off, reading their switching state).

B. Bibliography

[1] Drew Gislason, *Zigbee Wireless Networking*. Oxford, USA: Newnes Publications, 2008.

[2] Gerald Kupris and Axel Sikora, *Zigbee*.: Franzis, 2007.

[3] Raggl Thomas, *smartHomeControl - a proof of concept using ZigBee*.: not published, 2011.

[4] Fred Eady, *Hands-On Zigbee*.: Butterworth Heinemann, 2007.

[5] ZigBee Alliance, *ZigBee Cluster Library Specification*., 2008.

[6] ZigBee Alliance, *ZigBee Home Automation Public Application Profile*., 2010.

[7] ZigBee Alliance, *ZigBee Smart Energy Profile Specification*., 2008.

[8] Texas Instruments, *Z-Stack Monitor & Test API*., 2010.

[9] Jens Kühner, *Expert.NET Micro Framework*.: Apress, 2009.

[10] Donald Thompson and Rob S. Miles, *Embedded Programming with the Microsoft.NET Micro Framework*.: Microsoft Press, 2007.

[11] W3C Schools. (2011, March) Introduction to Web Services. [Online]. http://www.w3schools.com/webservices/ws_intro.asp

[12] OASIS. (2011, March) Devices Profile for Web Services (DPWS). [Online]. http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01