

Wireless Short Range Network for the Active Mirror Control of the CTA Experiment

Bachelor Thesis

Mathematisch-naturwissenschaftliche Fakultät
der

Universität Zürich

Roman Gredig



Prof. Dr. Ulrich Straumann
Dr. Achim Vollhardt

Zürich
Juli 2009

Acknowledgments

I thank Prof. Ulrich Straumann for giving me the opportunity to make this bachelor thesis in his group.

A special thank goes to my tutor Dr. Achim Vollhardt who supported me always, took the time to answer all my questions and let me work in his laboratory.

Further I'd like to thank Boris Begemann who gave me some important tips in developing my C# applications.

Contents

1	Introduction	6
1.1	The Cherenkov Telescope Array (CTA) Project	6
1.2	The Goal of this Bachelor Thesis	6
2	Realizing Wireless Communication	8
2.1	Design Parameters	8
2.2	Why Using Wireless Communication?	9
2.3	Comparison of Different Technologies	10
2.3.1	Bluetooth	10
2.3.2	Wireless USB	10
2.3.3	Wi-Fi	11
2.3.4	IEEE 802.15.4	11
3	The RF Data Modules	14
3.1	XBee 802.15.4 OEM RF Modules	14
3.1.1	Set-up a Network with Xbee	14
3.1.2	Short User Manual	15
3.1.3	The API Frame Specifications	19
3.2	The Z-Accel 2.4 GHz ZigBee Processor CC2480	20
3.2.1	Set-up a Network with the CC2480	21
3.2.2	Application Profiles and Binding in ZigBee	21
3.2.3	Controlling the Modules	22
3.2.4	Short User Manual	23
4	The Printed Circuit Board	26
4.1	Definition	26
4.2	The Development of the PCB	26
4.3	Requirements of the PCB	26
4.4	The Components on the PCB	26
4.4.1	The Power Supply	26
4.4.2	The Jumpers	27
4.4.3	Interface	28
4.4.4	The Xbee-Module	28
4.4.5	The CC2480-Module	29
5	The Module-Control-Application	30
5.1	Programming Language	30
5.2	Purpose of the Application	30
5.3	Controlling the Xbee Module	31

5.3.1	Example of the Usage of the Application	32
5.4	Controlling the CC2480 Module	34
5.4.1	Example of the Usage of the Application	35
6	Tests and Evaluation	37
6.1	Test Scenarios with the Xbee Modules	38
6.1.1	Broadcasting	38
6.1.2	Individual Addressing	38
6.2	Test Scenarios with the CC2480 Modules	41
6.2.1	Broadcasting	41
6.2.2	Individual Addressing	41
6.3	Summarized Results	42
6.4	Conclusion and Open Questions	43
	References	45
	A Details of the PCB design	46
	B Listings	48
B.1	Xbee Application	48
B.2	CC2480 Application	58

1 Introduction

1.1 The Cherenkov Telescope Array (CTA) Project

CTA is a project to build a ground-based gamma-ray telescope array to observe cosmic gamma-rays from about 10 GeV to more than 100 TeV. It is intended to analyze gamma-ray sources coming from our or other galaxies. It is planned to have 5–10 times higher sensitivity than current telescopes. [1] [2]

The observed cherenkov light is produced by high energy gamma quants. If they reach atmospheric molecules they can undergo pair production by producing an electron and positron. Those leptons can produce again photons by bremsstrahlung and compton scattering. This photons begin again to produce electrons and positrons by pair production. Leptons produced this way have so high velocities¹ that they generate cherenkov light. This can be detected with the telescopes (see fig. 1).

1.2 The Goal of this Bachelor Thesis

This bachelor thesis is a concept study for the realization of a wireless short range data network for the active mirror control of the CTA experiment. In the first part it has been evaluated what kind of network is most qualified to fit the requirements.

In the second part two different modules, working with the IEEE 802.15.4 standard, have been set up to perform first tests. This included a development of a printed circuit board to operate the modules and an application written in C# to control them.

¹higher than the speed of light in the corresponding medium

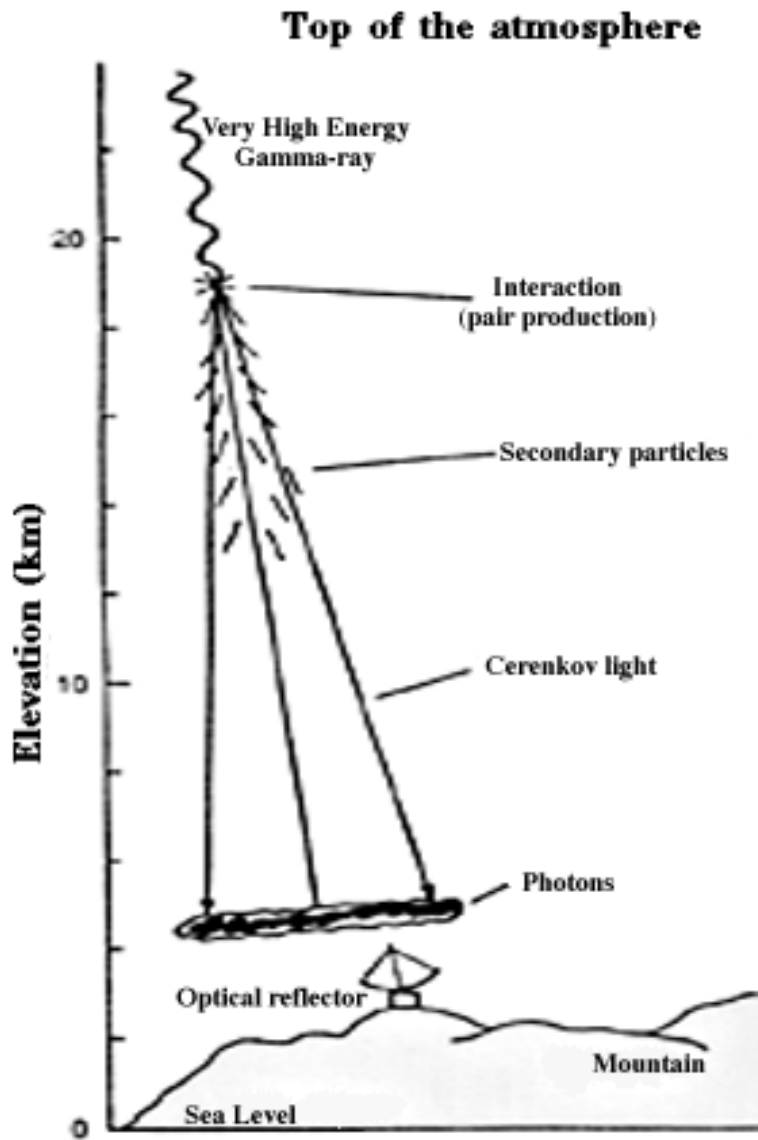


Figure 1: Schematic of a cherenkov light cascade [3]

2 Realizing Wireless Communication

There are many different possibilities to realize a wireless network. For each field of application there is another solution. It has to be chosen a system that fulfills the qualifications.

2.1 Design Parameters

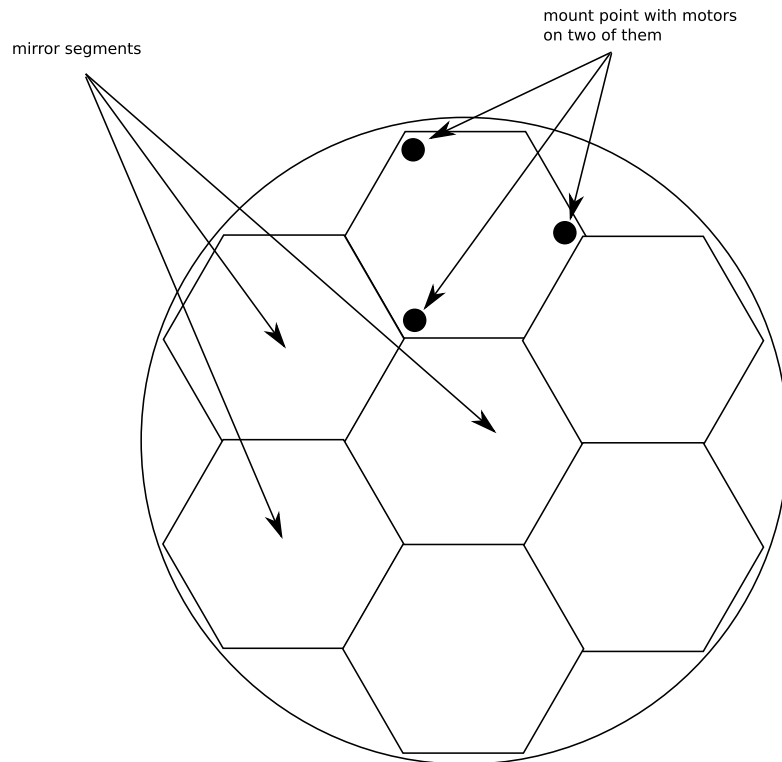


Figure 2: Schematic top view of a telescope dish with the mirrorsegments

To decide what kind of wireless network would be the best, the design parameters have to be set. The CTA consists of many telescopes with different sizes. The biggest ones will have a diameter of about 20 meters. Each mirror of a telescope will be segmented to small hexagonal elements (up to ~ 1000) (see fig. 2). Each segment will be mounted on three points. On two points there will be a motor that allows to change the mirror-segments' orientation. Every element has to be adjusted in the right way to form together the desired shape of the telescope. Additionally they have to compensate deformations when the whole telescope changes its elevation angle. The telescope will change its orientation about every 5 minutes.

This means that for each telescope there has to be formed a network that allows the controlling of all mirror-segments from a central point. The network has to be able to send and receive commands from every motor. Sending the command to adjust the mirrors should be fast because of the 5 minute reset period. But the bandwidth of the communication between the coordinator and the motor has not to be high, since a positioning command just contains several bytes.

It should be easy to integrate the system into the other electronics and to control it with common microprocessors. Because there is a huge amount of devices to be equipped (hundreds of telescopes with up to 1000 motors), it should be cheap to equip the motors with wireless data communication to hold down the costs. Also the energy consumption has to be kept in sight.

Finally it would be the best, when the system uses a common open standard for not being bounded to a specific manufacturer. Summarized, the system that will be used has to respect the following parameters:

- low energy consumption
- cheap
- handle distances about 10–20 meters
- able to control up to 2000 nodes
- fast (not in bandwidth but by sending commands up to 2000 nodes)
- small (it has to fit into the AMC case)
- easy to integrate into the other electronics
- bidirectional communication
- protection of unwanted network access
- open standard

2.2 Why Using Wireless Communication?

Before different wireless technologies are compared, it has to be discussed why a wireless solution should be considered. As explained above there are thousands of mirrors that have to be controlled individually. Wireless communication is not the only way to do this. In existing projects they are usually controlled by a wired data connection. One of the biggest advantages of a wireless solution is the price. By using wireless techniques it isn't necessary that the motors have to be connected to a network via cables. This means that it isn't necessary to lay a data cable for every motor. Because of this huge amount of mirrors one can save money by economize manpower and by using much less

cables. The only cables that are needed are the ones for the power supply which are also much less delicate.

Another point is the connection to the motors. The AMC case is outdoors and therefore it has to be waterproof to protect the inner electronics. Every connection involves the risk of a leak in the AMC case. Having cables also means that there is the risk to have broken ones. Replacing a broken cable needs much more time than only replacing a broken network module because a cable had potentially to be rerouted over the whole telescope.

2.3 Comparison of Different Technologies

2.3.1 Bluetooth

Bluetooth [4] is a wireless protocol designed for data exchange over short distances based on the IEEE 802.15.1 standard. It's most known for data transfer between mobile phones and/or head-sets. Bluetooth is working the ISM² license free 2.4 GHz band. During communication the frequency hops between 79 different frequencies to minimize interference with other devices. In a Bluetooth network exists one master and up to 7 active slaves (piconet). Additionally there can be up to 255 devices in such a piconet. These devices stay synchronized with the others but not active. Theoretically it's possible to combine such piconets to bigger networks by having devices that are registered in more than one piconets. Such extended networks are called *scatternet*. In a scatternet exist devices that are part of two piconets. They can transfer data from one piconet to another. But this is very complicated: the different piconets are not synchronized and the mentioned slave has to negotiate with more then one master when it is active in which piconet [5]. This leads to high latencies when a packet has to travel through the network by crossing different piconets. In the Bluetooth stack exist no routing protocols so it had to be realized in the application layer. The realizing of bigger network by using scatternets is rather a theoretical approach.

Scatternets are not an effective way to realize bigger networks. Therefore, Bluetooth is not an option for the CTA.

2.3.2 Wireless USB

Wireless USB is an extension of the existing *Universal Serial Bus*. It is designed to connect peripheral devices to the computer. Wireless USB works on frequencies between 3.1 and 10.6 GHz. It uses a star topology. The network consists of one host and (up to) 127 devices connected to it. Those devices can only communicate with the host. Data rates are up to 480 Mbit/s (distances lower than 3 meters).

Wireless USB is not an option because the network size is limited to 127 devices.

²Industrial, Scientific, and Medical Band

2.3.3 Wi-Fi

Wi-Fi, also known as Wireless LAN (WLAN), is based on the IEEE 802.11a/b/g/n standard. It's most known to connect computers to the internet with broadband speed. Wi-Fi allows network sizes up to 2007 devices. This would be close enough to control the biggest telescopes. Wi-Fi is designed for having high data rates (~ 54 Mbit/s) and ranges about 100 meters. Wi-Fi doesn't meet the design parameters because it has too high power consumption (10 times higher than IEEE 802.15.4 for example). Because of the high range of Wi-Fi there is also the problem that different telescopes would interfere between each other. Only three different Wi-Fi networks can operate in an overlapping region without interfering.

2.3.4 IEEE 802.15.4

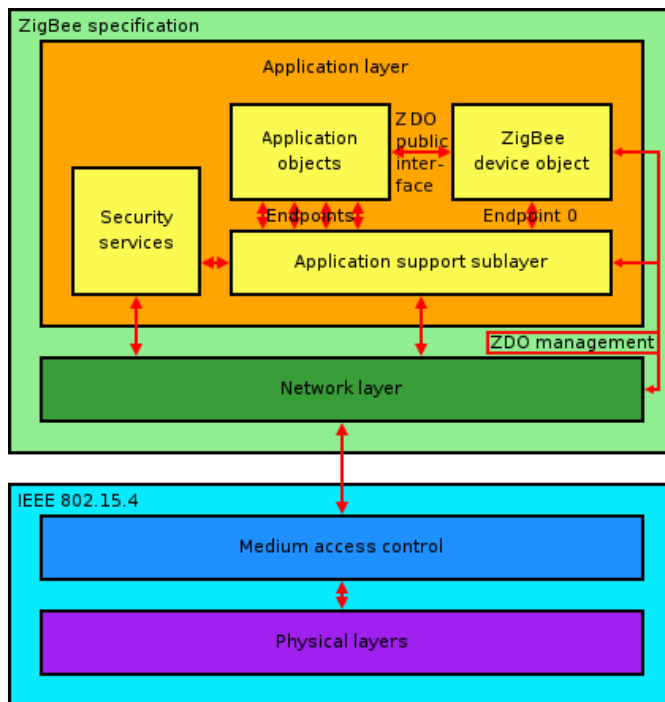


Figure 3: the layers of the OSI model [6]

IEEE 802.15.4 is a standard that specifies the physical (PHY) and the media access control (MAC) layer for personal area networks (see fig. 3). It focuses on low-cost, low-speed and low-energy consumption. Some of the main characteristics are:

- data rate up to 250 kb/s (@ 2.45 GHz)

- star or peer-to-peer operation
- 16 or 64 bit addressing of the devices
- acknowledged protocol for transfer reliability
- low power consumption
- 16 channels in the 2.45 GHz band, 30 channels in the 915 MHz band, 3 channels in the 868 MHz band

This standard defines two different types of devices that can join the network. A full-function device (FFD) has to support all features of the standard. This means that it can operate as a personal area network coordinator, a coordinator or an end-device. FFD can start and manage networks. Each network needs one Personal Area Network (PAN) coordinator. FFDs can talk to other FFDs or RFDs. Reduced-function devices (RFD) can only join to a existing network. They are only able to talk to FFDs. In order to run a network, at least one FFD is required. RFDs are designed for simple applications (like a light switch).

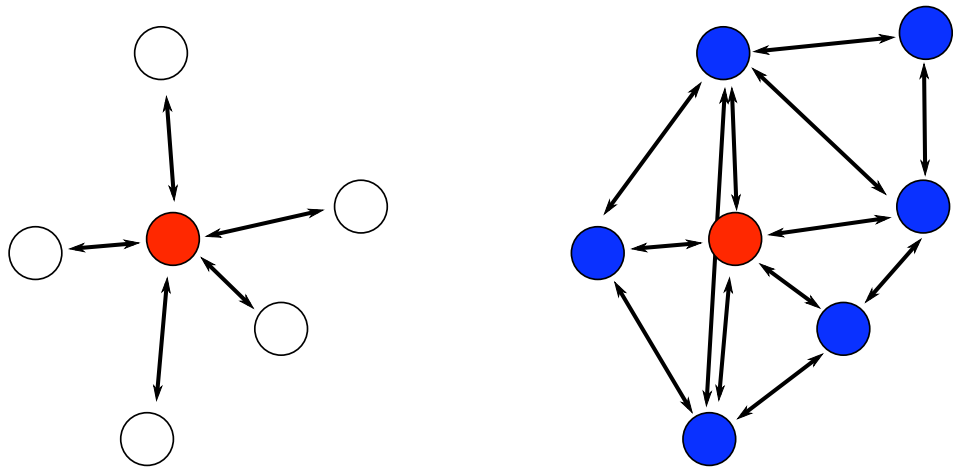


Figure 4: The different IEEE 802.15.4 topologies. Left picture: star topology. The PAN coordinator is in red, the end-devices are white. Right picture: peer-to-peer topology. The PAN Coordinator is in red, the other joined devices are FFDs (in blue).

Network Topologies IEEE 802.15.4 supports two different network topologies. It can operate in a star topology or a peer-to-peer topology. In star topology (see fig. 4) the communication is always between a central point (the PAN coordinator) and the end devices. They can only communicate with the controller.

In a peer-to-peer topology there is also a PAN. But here, every device can communicate to others that are in its range.

ZigBee ZigBee specifies a protocol stack that is based on IEEE 802.15.4. ZigBee specifies three different devices. There are end-devices with a simplified stack (i.e. RFD in IEEE 802.15.4). Additionally there is a coordinator that manages the network and a router. Routers are FFDs that allow to route data. This is necessary for a network with tree topologies. With routers it's possible to extend networks. Star, mesh and cluster tree networks can be implemented.

Conclusion Networks build with IEEE 802.15.4, with or without ZigBee, seems to be the best choice. They fulfill all requirements defined in paragraph 2.1. They are cheap and the only ones that support such a big number of devices in a network. As it is an open standard there is no danger to be dependent to a specific manufacturer. Because the area that has to be covered is relatively small, there is no routing or network extension needed. All motors of the telescope are near enough to contact a central point directly. This means that a simple star topology can already realized with IEEE 802.15.4 without the ZigBee stack. This helps to keep the implementation of a network as simple as possible.

3 The RF Data Modules

For this thesis two different modules coming from different manufacturers have been tested. Both are modules working on the IEEE 802.15.4 standard. They use the same technique to build a network and to communicate among each other.

3.1 XBee 802.15.4 OEM RF Modules

The first module that is tested is the XBee 802.15.4 OEM RF Module. This module is produced by Digi (formerly known as MaxStream). It is engineered to meet the IEEE 802.15.4 standard.

This modules are also available with other firmwares. There is a firmware with a ZigBee stack (see chapter 3.2) and one with DigiMesh. DigiMesh is a proprietary peer-to-peer topology without specific routers or coordinators. In a DigiMesh network all devices can route data. They form dynamically a network in which all devices are reachable. If one device breaks down, the routing will be changed for having still all other devices connected to the network.

In our application the area of one network is small. Therefore there is no data routing between two modules needed.

To find out which firmware is the best, the company *reselec ag*³ has been contacted. *reselec ag* is a swiss company that sells Xbee Modules and supports customers in developing networks. For our application the XBee 802.15.4 firmware is the best. It's simple and can support such a big network. There are two different modes to operate the modules:

Peer-to-Peer: This is the default mode. In this mode the modules work in a peer-to-peer topology. There are no Master/Slave relationships.

NonBeacon (with Coordinator): In this mode, one module has to be configured as a coordinator. The coordinator manages the network. All other devices are so called *end-devices*. They have to associate with the network via the coordinator.

3.1.1 Set-up a Network with Xbee

A network consists of one coordinator and one or more end-devices. Such a network forms a PAN (Personal Area Network). Each PAN is identified via a unique number (the PAN ID). Every device belonging to a specific PAN has to have set this PAN ID. This prevents miscommunication between different networks. This is very important when there are different networks that are close together (e.g. two different telescopes).

³<http://www.reselec.ch>

Starting up the Coordinator The starting of a coordinator is controlled by the A2 command (for all commands refer to [7]). There can be made different modifications. It is possible to perform active scans to detect existing PANs. The second setting is to perform an energy scan during start up. Energy scan means that the coordinator scans for energy on every channel for a given time. This procedure helps to find a free channel where no other networks disturb the communication. It's also possible to start the coordinator neither performing an active scan nor an energy scan. Then it retains the settings that are already programmed.

Starting up the End-Devices There exist also different startup configurations for the end-devices. The end-devices can be configured to search for every coordinator that is reachable. It can also be configured to associate only to a coordinator with a specific PAN ID and/or specific channel.

In our application the network environment is well known and defined. Therefore it would be the best to preset PAN ID and channel for every telescope. This renders a lot of the dynamic configuring and association features unnecessary.

3.1.2 Short User Manual

General For setting up a network with the XBee Modules like described in 3.1.1, several commands have to be sent. By factory setting they are in the so called *transparent mode*. This means that all data coming in via the UART⁴ interface will be sent. Received data will be output directly to the UART. For configuring the module, it has to be set in the AT command mode. This can be done by sending the following sequence (consider flowchart in figure 5):

```
-> wait 1 sec
+++
-> wait 1 sec
-> the device returns "OK"
```

Now it's possible to configure the device via the AT commands. The AT command mode can be left by sending

```
ATCN <CR>
```

After leaving the AT command mode, the device returns to the transparent mode. The waiting time before and after sending the +++ can be adjusted with the ATGT command. A complete sequence would be the following:

⁴UART: Universal Asynchronous Receiver Transmitter. This is a interface that allows serial communication between the module and other devices. Data can be sent or received by a bit-serial data stream.

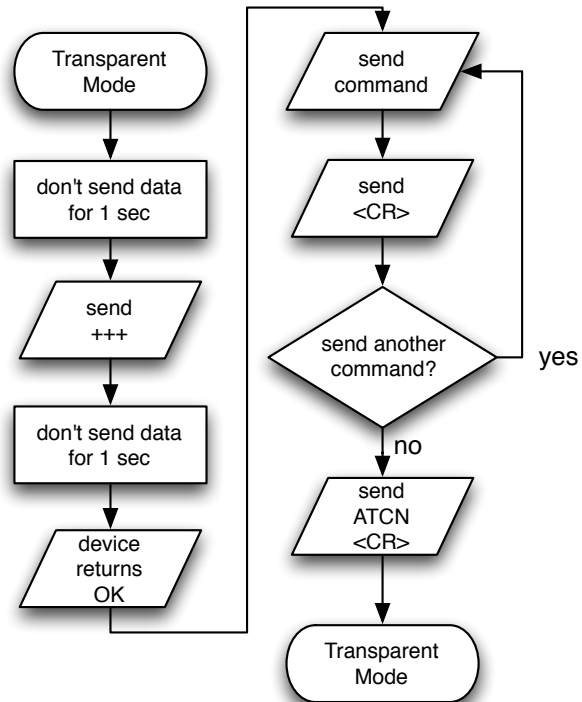


Figure 5: flowchart for configuring the XBee Module by using AT commands while working in transparent mode

```

-> wait 1 sec
+++
-> wait 1 sec
-> response: OK <CR>
ATCE1 <CR>
-> response: OK <CR>
ATMY20 <CR>
-> response: OK <CR>
ATCN <CR>
-> response: OK <CR>
    
```

Multiple commands can also be combined in one line, separated by commas, like in the following example:

```

-> wait 1 sec
+++
-> wait 1 sec
    
```



```
-> response: OK <CR>
ATDL1A0,WR,CN <CR>
-> response: OK, OK, OK <CR>
```

For this user manual we consider the following scenario: We have 3 devices forming a network. Device *A* is the coordinator that starts a network. The coordinator will have a 16-bit short address: 0x0000. Devices *B* and *C* work as end-devices which want to associate with the network initialized by the coordinator. They will have the 16-bit addresses 0x0001 and 0x0002.

Configuring the Coordinator To configure a module to operate as a coordinator, the CE (Coordinator Enabled) parameter has to be set to 1. The coordinator can undergo different start up sequences.

The A2 parameter controls how the coordinator starts. If the bit 0 of the A2 bitfield is set, the coordinator sends on every channel a BeaconRequest command to the broadcast address (0xFFFF) and broadcast PAN ID (0xFFFF). It listens then for beacons from any coordinator. This procedure will be repeated for every channel. Finally, the coordinator will either retain its PAN ID when no other coordinator has been found with this or it will update its PAN ID to a new, unused ID. If the A2 bit 0 is set to 0, no active scan is performed. The coordinator will retain its setting.

The bit 1 of the A2 bitfield controls whether the coordinator issues an energy scan or not. If it is set to 1, the coordinator scans for energy on all channels. This helps to find out on which channel exists a minimum of energy produced by other networks. After scanning, the coordinator sets the channel setting to the channel with the least energy. If the bit 1 is set to 0, the coordinator will not perform an energy scan and retain its channel setting. The bit 2 of the A2 bitfield controls whether other end-devices are allowed to join the network or not.

If addressing with the 16-bit short addresses is desired, every module needs a unique address. This address can be set with the ATMY command. Here the coordinator will have the 16-bit short address 0x0000.

This example shows how to configure the module to work as a coordinator that will perform an energy scan during startup:

```
-> wait 1 sec
+++
-> wait 1 sec
ATCE1 <CR>
ATA26 <CR>
ATMY0 <CR>
ATWR <CR>
```

The last command ATWR saves the settings to the non volatile memory. After a reset of the coordinator, it will retain this settings. After every restart, the module will perform a energy scan and search for a free channel.

Configuring the End-Devices For this example the end-devices will be configured to search for a coordinator with any PAN ID on any channel. This can be controlled with the A1 (End-Device Association Command). The A1 requires also a bitfield. Bit 2 controls whether the device tries to associate with a coordinator or not. If the bit 1 is set to 1, the device can associate with a PAN with any channel value. The bit 0 controls, whether the device can associate with a PAN with any PAN ID value or only with a PAN with the same PAN ID like the end-device currently has. Furthermore we only want that the end-devices communicate with the coordinator. Hence, it follows that the destination where data has to be sent can be programmed once. By using short addresses, two parameters have to be set. The destination high address has always to be 0x0000. (ATDHO). The destination low address has to fit the MY address of the destination device. For the end-devices it's always the coordinator, that uses the MY address 0x0000. For our example the setting is:

```
-> wait 1 sec
+++
-> wait 1 sec
ATA1F <CR>
ATMYx <CR> (with x > 0)
ATDHO <CR>
ATDLO <CR>
ATWR <CR>
```

Usage of the Network When the initial settings are made like above, the network starts up automatically as desired. After the association procedure, all devices are ready to receive and send data from/to the coordinator. As mentioned above, the modules work by default configuration in the transparent mode. For the end-devices, this would be the preferred modus. After an initial configuration there are no additional commands to be sent. It's not necessary to change the destination address for transmitting data due to the fact that in our application they only have to talk to the coordinator. Principally an initial configuration could be made before the module will be integrated to the AMC unit. The microprocessor on the AMC has not to send any configuration commands to the module itself. It has only to interpret commands coming from the coordinator and send answers to it.

For the coordinator itself, the situation is more complex. It has to send commands to different nodes (i.e. every motor). That's the reason why it's necessary to change the destination low address before (the ATDL command). If an operation in transparent mode

is desired, it has always to be changed into the AT command mode for changing the destination address. This costs a lot of time. Another disadvantage of the transparent mode is, that in point to multipoint systems, it is not clear from which remote module incoming data comes, except it is implemented in the application that the remote address is sent too.

Since there is also the possibility to use a module in the API mode⁵, this would be the preferred mode for the coordinator. In the API mode there is directly a command to send data with the destination address as a part of the command. By using this API command mode, it has not to be switched first to the AT command mode, changed the destination address and then switched back to the transparent mode, before data can be sent. Another advantage of the API command mode is, that every data send command immediately returns a answer that indicates whether the transmission was successfully or not.

The next subsection will explain the principles of the mentioned API command mode.

3.1.3 The API Frame Specifications

Start Delimiter (1 Byte)	Length (Bytes 2-3)		Frame Data (Bytes 4-n)	Chksum (Byte n+1)
0x7E	MSB	LSB	API-specific Structure	1 Byte

Table 1: UART data frame in API operation

The API operation requires that the communication with the module is done in a structured interface. In the API Frame Specifications [7] it is defined how the commands [responses] are sent [received] from the module. For using the API mode, the module has to be configured appropriate. While being in the AT command mode it can be done with the ATAP command:

ATAP0 sets the module to transparent operation

ATAP1 activates API Operation

ATAP2 activates API Operation with escaped characters

In the API operation a UART data frame is defined as seen in table 1. The first byte that has to be transmitted is 0x7E. This indicates the start of a data frame. The next two bytes indicate the number of bytes that includes the Frame Data. Now the Frame Data follows. This is the actual API command with additional data in it. The last byte that finishes a data frame is the checksum. The checksum ensures data integrity. The calculation of the checksum is: $0xFF - (\text{sum over all Frame Data bytes})$.

⁵API: Application Programming Interface

3.2 The Z-Accel 2.4 GHz ZigBee Processor CC2480

The second tested module is the CC2480 Z-Accel 2.4 GHz ZigBee Processor. This module supports the ZigBee 2006 stack. Unlike the XBee 802.15.4 OEM RF Modules it can build a real ZigBee network with coordinators, routers and end-devices. The CC2480 is produced by Texas Instruments. It comes in a small 7 x 7 mm QFN-48⁶ package that is a surface-mount device.

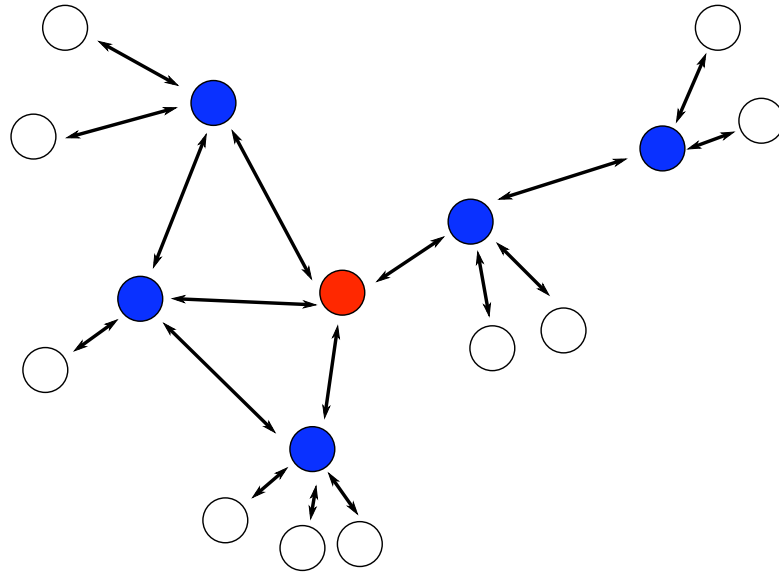


Figure 6: Example of a ZigBee network. The coordinator is in red, routers in blue and end-devices are white.

A ZigBee network consists of one coordinator and one or more end-devices and routers (see fig. 6). The coordinator is the device that starts and manages the network. Since ZigBee is based on IEEE 802.15.4, the coordinator has to specify some network parameters like the PAN ID and the channel.

When the coordinator started up, other devices can join this network. They can be end-devices and routers. If there are routers in a network, other end-devices and routers can join the network via these routers. In this case they are called *childs* of the router. The possibility to have routers (that are childs of the coordinator or of other routers) enables the network to have devices communicating between each other that are out of their RF range. Messages between such two devices are then routed via the routers in between. Depending on the ZigBee specifications and the limitations of the CC2480 module the network can have the following structure [8]:

⁶QFN: Quad Flat No leads

- The maximum network depth (i.e. the hops between a device and the coordinator) is limited to 6.
- Each router or coordinator can have up to 20 child devices. Of these, up to 6 can be routers and the others end-devices.

3.2.1 Set-up a Network with the CC2480

The CC2480 control commands include the so called *Simple API Interface* (SAPI). The SAPI allows a developer to build up a ZigBee network with the knowledge of only 10 commands. This is an easy way to begin developing ZigBee networks. All tests made for this thesis with the CC2480 are made with the SAPI.

Starting up the Coordinator The device type of the module can be controlled with the `ZCD_NV_LOGICAL_TYPE` parameter. The value `0x00` stands for the coordinator. The PAN ID for the network can be controlled by the `ZCD_NV_PANID` parameter. For the channel on which the network has to work exists the `ZCD_NV_CHANLIST` parameter. This is a bit mask of channels. If more than one channel is selected, the device will perform an energy scan. Those channels with the highest energy will be discarded. After the energy scan the device determines the number of existing (other) networks. Finally it will choose the channel with the fewest networks on it.

After having set the network parameters, the application has to be registered. This is done with the `ZB_APP_REGISTER_REQUEST`. The coordinator is now ready to work. The ZigBee stack can be started with the `ZB_START_REQUEST`. Joining the network by other devices is controlled with the `ZB_PERMIT_JOINING_REQUEST`. This command allows to control the joining for individual devices or a group of specific devices (i.e. routers, end-devices).

Set-up end-devices The modules that have to work as an end-device can be set in the corresponding modus with the `ZCD_NV_LOGICAL_TYPE`. Again the application has to be registered with the `ZB_APP_REGISTER_REQUEST` command. The end-devices can now be started with the `ZB_START_REQUEST`. After this, they try to connect to a network (via a coordinator or a router that allows joining).

3.2.2 Application Profiles and Binding in ZigBee

As the CC2480 is a device using ZigBee, there is feature called *Application Profile* [8]. An application profile defines different devices (like a light on/off switch or the lamp), a set of commands exchanged by the different devices and a format how the data is exchanged between the devices.

Suchlike profiles can be either *public* or *private*. The sense of public profiles is interoperability. If one buys a lamp from manufacturer A with ZigBee modules from brand

It's possible to buy a on/off switch from manufacturer B with a ZigBee module in it from brand D. Because public profiles are well defined by the ZigBee Alliance those two products are able to work together. If such a feature is not required, it's possible to use private profiles. Application profiles play an important role, when the *binding* feature is used.

With binding it's possible to generate logical links between devices at the application layer. This means that an application can send data to another device without knowing its destination address. The sending module can then look up the corresponding destination address from its binding table. During the registration of the application it will be specified which module generates data or receives data.

For the test made in this thesis, the destination addresses were known (they can be retrieved with the test application). Therefore the binding feature has not been used. For setting up a network an application profile has to be registered. For this tests and examples a simple light switch profile has been chosen. Because data transmission has been performed with direct 16-bit addressing, it didn't matter what kind of profile has been chosen because the binding feature wasn't used.

3.2.3 Controlling the Modules

The CC2480 can only be controlled by using a well defined application command interface (similar to the API mode of the XBee modules). Because the CC2480 can't be controlled by using AT commands, it is suggested using an application (or a microprocessor) that sends the commands in the right way.

General UART Communication and the Application Interface Communication between an application and the module has to happen in a well defined UART transport frame format [9]. Every data frame has to begin with the *start of frame indicator* SOF byte. The value is (in HEX) 0xFE. Next, the *general format frame* follows. This is actually the specific command or message that has to be sent. The data frame has to be closed with a checksum to ensure correct transmission between the module and the application. The checksum is computed as an XOR of all the bytes from the general format frame. Table 2 shows an overview of the data format. The general frame format

Start indicator (1 Byte)	General Frame Format (Bytes 3 – 256)	Checksum (1 Byte)
0xFE		XOR of all Bytes in the General Frame Format

Table 2: data frame format for the CC2480 API [9]

is well defined too. It contains three parts (see table 3). The first byte defines the length

of the data field. The next two bytes represent the command and the last part includes the data. The content and size of the data field depends on the command and can be empty (depending on the command).

Length (1 Byte)	Command (2 Bytes)	Data (0-253 Bytes)
--------------------	----------------------	-----------------------

Table 3: general frame format for the CC2480 API [9]

3.2.4 Short User Manual

Considering a scenario with one coordinator and two end-devices the modules have to be programmed appropriate.

The Coordinator The coordinator manages the whole network and has to be set up as a first step. The flowchart in figure 7 shows the commands that have to be sent to configure and start a module as a coordinator. The first commands reset the device completely. This helps to bring it to a known state. After resetting, the device will be configured working as a coordinator. As a next step, the application has to be registered. Finally the ZigBee stack can be started. Now the coordinator is up and running. With the last command the coordinator allows other devices to join the network. This command includes a timeout byte. This byte specifies how long other devices can join the network.

The End-Devices To configure the other modules working as end-devices, the commands in the flowchart of fig. 8 have to be sent. The first steps are again to reset the module and bring it in a known state. The next command sets the module to work as an end-device. After that, the application can be registered and the ZigBee stack started. The device will then try to connect to a coordinator or router to join a network. A successfully joining will be confirmed with a status response after starting the stack.

Usage of the Network Having formed a network, all modules are ready to send or receive data. All devices will get a 16-bit short address after having joined a network successfully.

Sending data can be achieved by using the `ZB_SEND_DATA_REQUEST`. With this command it's possible to send data by using the 16-bit short address of the destination. The short addresses `0xFFFC`, `0xFFFFD`, `0xFFFFE` and `0xFFFF` are reserved for using broadcasting or the binding feature.

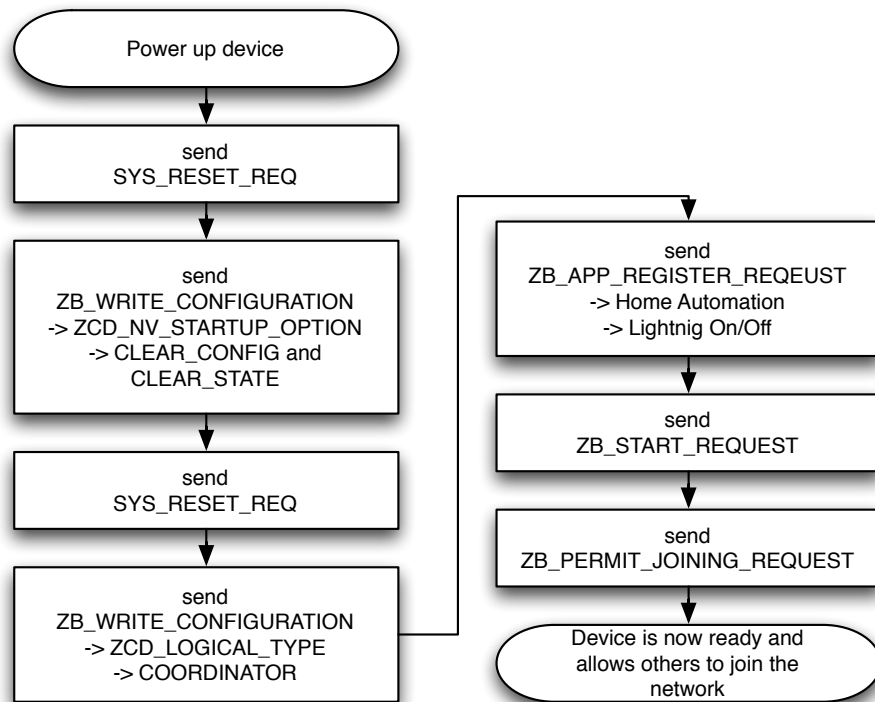


Figure 7: Flowchart for configuring a CC2480 Module working as a coordinator and allowing other devices to join the network.

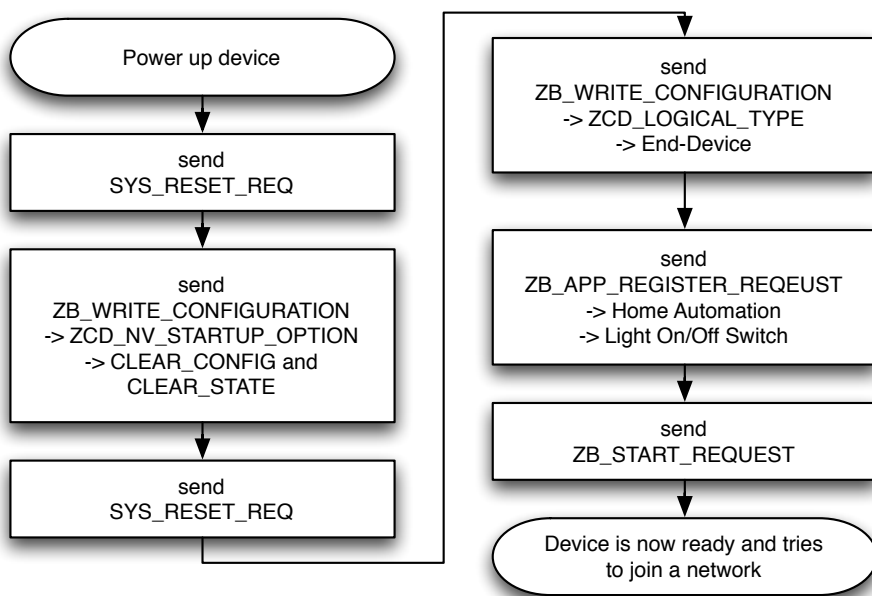


Figure 8: Flowchart for configuring a CC2480 Module working as a end-device.

4 The Printed Circuit Board

4.1 Definition

A printed circuit board (PCB) is used to electrically connect and support electronic components. The electronic components are connected by conductive pathways. Those pathways are etched from a copper sheet laminated onto a non-conductive substrate.

4.2 The Development of the PCB

The PCB has been designed on Eagle [10], which is a powerful tool to draw schematic diagrams and design PCBs. For non-profit or evaluation purposes it's free. The first step in developing a PCB is to draw a schematic diagram. Such a diagram defines which electronic components are needed and how they have to be connected. The schematic is an abstract design where components have neither the right position nor the actual size.

Having developed a diagram, the next step is the design of the PCB. In this work step it has to be determined where the components have to be and where the pathways are. Only now the position and size of the components matters. One has to group components that belong together and has to take care to not have crossing pathways (as long as they carry different signals). Depending on the complexity of the diagram it has to be decided how many layers are necessary. For this work a bilayer PCB was designed.

PCB designing usually results in a CAD-file that shows the final board (see figure 9). This file can be taken as a pattern for etching. The etching has been commissioned to a commercial company that produced ten boards. Figure 18 on page 47 shows a picture of the produced PCB without the parts soldered on it.

4.3 Requirements of the PCB

In our case the function of the PCB is to power up the modules and to provide an interface. The easiest way to test the modules is to do it with a personal computer. Applications can be written on the pc and one can test the modules with maximum control. There has to be programmed an additional microprocessor like a PIC.

Using a computer for testing simplifies the PCB. In addition to the powering there is only an interface needed. Furthermore it should be easy to select the module that is going to be tested. The layout of the PCB is shown in figure 9.

4.4 The Components on the PCB

4.4.1 The Power Supply

To power all the electronic components, there is the LM2937-3.3 voltage regulator [11]. This regulator can be used with a maximal input voltage of 26V. On the output pin it

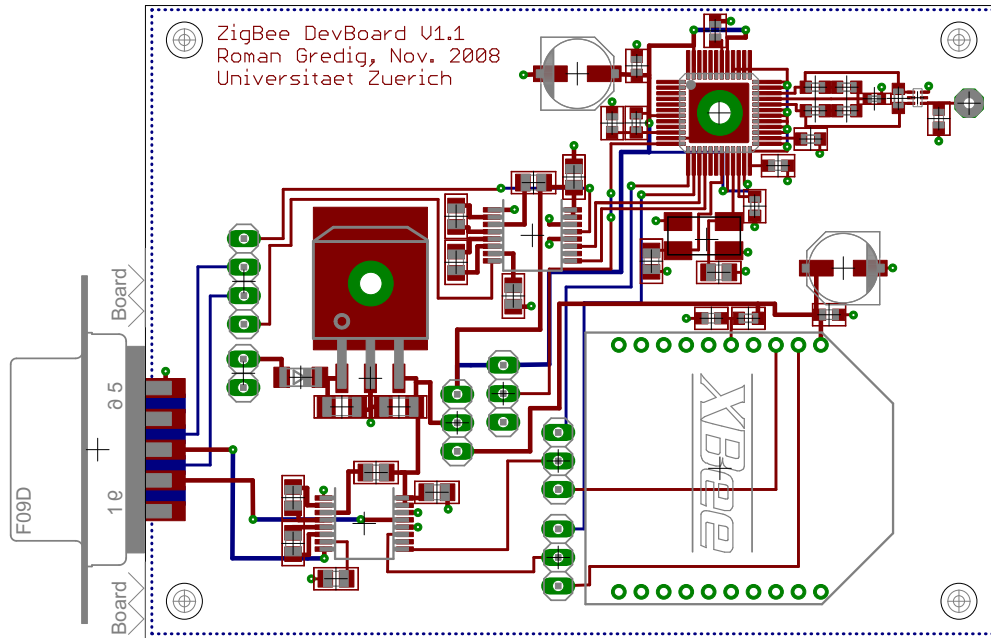


Figure 9: PCB for testing the two modules. Red wires are on the top side, blue wires on the back. The serial interface is at the left side, the CC2480 on top right and the XBee module on the lower right side. The labels of all components are visible in fig. 17 in appendix A.

provides 3.3V. A common diode has been placed in front of the input pin for reverse polarity protection.

4.4.2 The Jumpers

To select a specific module, some jumpers have to be set correctly. Jumper *JP2* selects which module has to be connected to the power supply. The jumpers *JP3* and *JP4* redirect the serial data lines from the computer either to the Xbee or the CC2480 module (refer to chapter 4.4.3). Jumper *JP5* activates the hardware handshake lines needed for the CC2480.

4.4.3 Interface

Both modules come with a built-in UART-Interface. Data transmission with UART is almost the same like the RS232-Interface known from personal computers. Both interfaces use a (bit-)serial procedure for data transmission. Two lines are responsible for the data transmission from and to the pc. Two other lines are used for the hardware handshake. The difference between UART and RS232 lies in the voltage level. RS232 defines the logic zero between 3 to 15 volts. The logic one is between -3 and -15 volts. UART uses transistor-transistor logic (TTL). A logic one is ≥ 2 volts and a logic zero ≤ 0.8 volts.

As the computer uses RS232 and the modules use UART, the electrical signals have to be converted. In this PCB a ICL3221ECAZ is used (see data-sheet [12]). Each ICL3221 can convert two signal lines between the TTL- and RS232-level. The first ICL3221 (*IC3* on the PCB) is used for converting the serial data lines TxD and RxD. The second one converts the hardware-handshake lines RTS and CTS for the CC2480. The pins are shown in figure 10. For 3.3V operation the ICL3221 needs $0.1\mu\text{F}$ capacitors for the charge pumps.

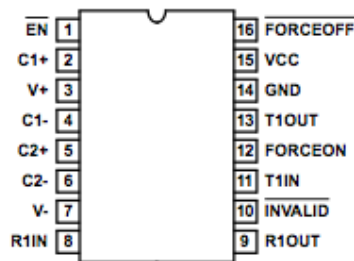


Figure 10: ICL3221ECAZ serial level converter [12]

For connecting the pc to the test board there is a DE-9 female connector at the side of the board.

4.4.4 The Xbee-Module

The integration of the Xbee-Module is very easy. It's connected to the 3.3V power supply coming from the jumper *JP2*. First, there is a voltage stabilization made with two capacitors near the VCC pin of the Module to the ground. The Xbee-Module has a RESET pin. It's active when the input is set low. *C5* and *R15* ensure that the module is in reset condition during powering up. After charging the capacitor, the reset input switches to high. The Pins DIN and DOUT are connected to the serial data line jumpers.

4.4.5 The CC2480-Module

The CC2480 is more complicated to get working. There is also a power stabilization like the Xbee-module has. The CC2480 has no antenna integrated. The upper left part of the PCB (left of the CC2480 module) shows the additional elements used for connecting an antenna. The CFG1-pin is connected to the jumper *JP6*. This jumper allows to set the SPI or UART transport. The CC2480 has also a reset pin. It's active when the input is set low. There is the same circuit to reset the module like above for the Xbee (realized with *C23* and *R2*).

5 The Module-Control-Application

To perform first tests on the modules, an application has been written to control them from a computer. Controlling the modules with a computer (rather than a microchip) has many advantages. Programming is easy and fast and it is possible to display the answers of the modules immediately.

5.1 Programming Language

The decision of the programming language has been made with focus on the following points:

- developing and runnable under *Microsoft Windows XP*⁷
- Graphical User Interface (GUI)
- accessing of the serial port
- easy and fast developing
- fast accessible support (online help and forums)
- object oriented programming
- free of charge

Respecting this points it has been decided to develop the applications in C# using the *Microsoft Visual C# 2008 Express Edition*⁸ which is available free of charge.

5.2 Purpose of the Application

The application allows the full control of the modules via the GUI. It's possible to control whether the connected device is a coordinator, a router or an end-device. Furthermore it is possible to send all commands to form a network (like selecting PAN ID, allow joining, addressing etc.).

When a network is formed, the application can send data and record transmission statistics. Data sending can be achieved with either broadcasting the data to all modules in a network or by sending it to a specific module (see chapter 6 for the different scenarios). The application visualizes all (status) messages that have been sent by the modules. This helps to debug the application and to analyze network issues.

⁷some of the drivers of the module test boards are only available for Windows XP

⁸<http://www.microsoft.com/Express/>

5.3 Controlling the Xbee Module

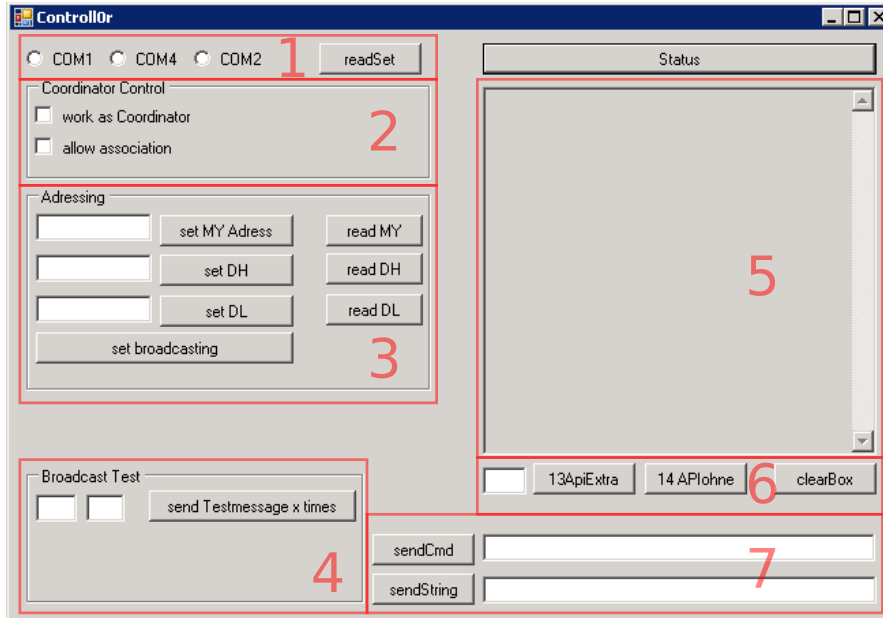


Figure 11: Screenshot of the Xbee Module control application running on Windows XP

The first application that has been written is the one to control the Xbee Module. Fig. 11 shows a screenshot of the GUI. In region (1) it can be selected on which COM Port the module is connected to. Region (2) is the *Coordinator Control*. It allows to run a module as a coordinator and (if it's a coordinator) whether other modules can join its network or not.

Region (3) includes everything for addressing. The buttons `set MY Address` and `read MY` are for setting and reading the attached module's own 16-bit short address. The buttons `set DH`, `set DL`, `read DH` and `read DL` are for setting the destination addresses where data packets have to be sent to. The `set broadcasting` button sets the DL and the DH addresses for broadcasting. This buttons work only correct if the module is in the *transparent mode*.

Region (4) is for testing the broadcast feature. If the destination addresses are set correctly (e.g. by pushing `set broadcasting`), the `send Testmessage x times` button can be used to send x testmessages every y second. This feature has been used to test how many broadcast messages receive the end-devices (refer to chapter 6).

The text-field in region (5) displays all data incoming through the serial interface. This includes received data from another device as well as the output generated from the module itself. The input is readable only if the module works in the transparent

mode. In most cases there is an event handler in the application that reads all incoming data from the serial interface directly and displays it in the mentioned text-field. If commands are executed in the API command mode, the input is read byteswise and converted for the text-filed. This happens for example by using the buttons in region (6).

Region (6) has two commands which send data via the API command mode. As explained in 3.1.2 the API command mode allows the application to send a data send command that includes the destination address and the data directly. This is used here. The buttons `13APIExtra` and `14 APIohne` will send a test message to three different modules z times. This is to simulate sending data to all joined modules individually. In a telescope there can be up to 2000 end-devices. Because we had only three end-devices, the application will send the data z times to the three devices repeatedly. The difference between the two buttons is the following: `13APIExtra` will send the data without waiting for the acknowledge response. An extra thread in the application will receive and write the incoming data as soon as it comes. The button `14 APIohne` sends data to the first device and waits then for the acknowledge. Not until after receiving it, it will continue to send data to the next module. The reason why this different behaviors are implemented is explained in the next chapter. Before this buttons can be used, the module has to be set in the API command mode manually (for example via the `sendCMD` button).

The button `sendCMD` in region (7) is used to send a AT command manually while using the transparent mode of the module. With the button `sendString` an arbitrary string can sent to the module with the address configured in region (3) (also while using the transparent mode).

If there are more than one module connected to a computer with multi-COM Ports, it's possible to start more than one instance of the application.

5.3.1 Example of the Usage of the Application

The diagram in fig. 12 shows how to use the application to setup a module working as an end-device or a coordinator. After the steps shown in the diagram the module is ready for usage. If it's an end-device, it will try to connect to a coordinator. Only after these steps have been executed, the module is ready to send or receive messages from other network participants.

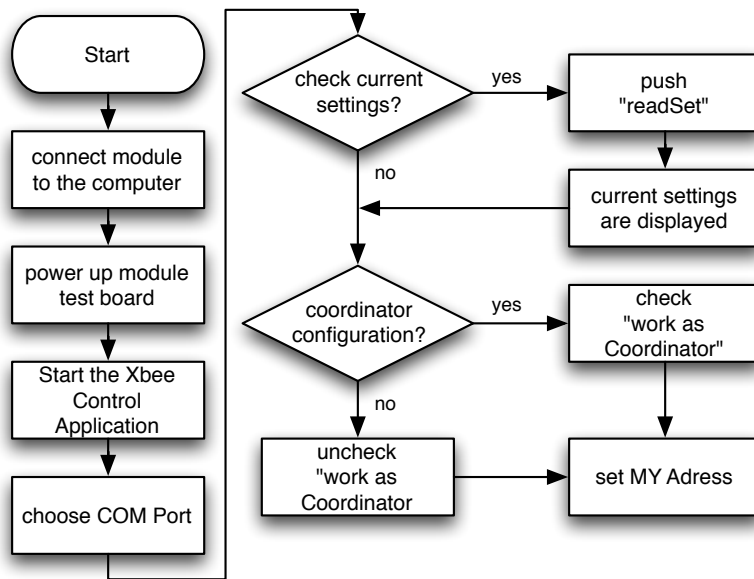


Figure 12: flowchart for using the application to control a Xbee module

5.4 Controlling the CC2480 Module

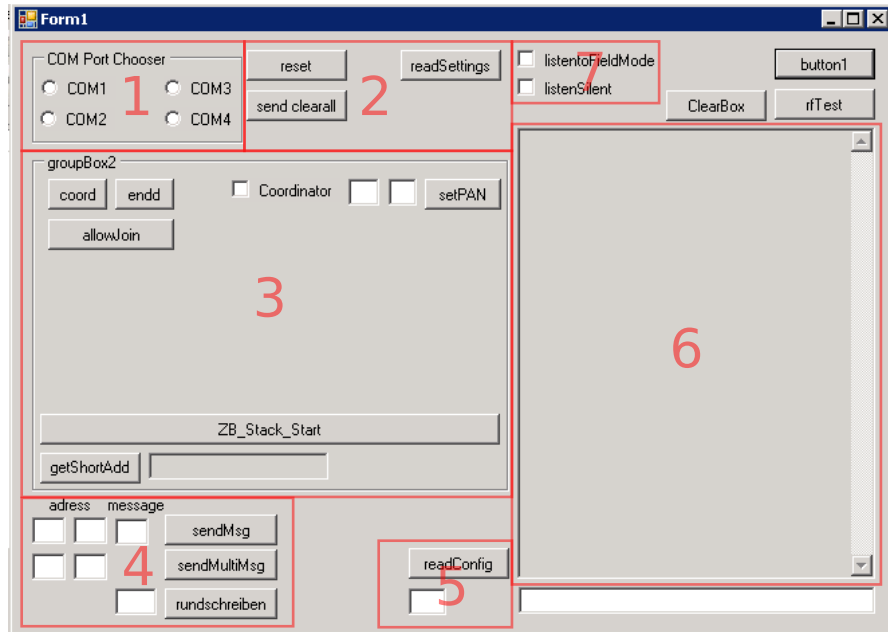


Figure 13: Screenshot of the CC2480 module control application running on Windows XP

The second application has been developed to control and test the CC2480 module. The big difference to the other module is that this module can only be controlled with serial commands that are similar to the API command mode from the former module. There is no transparent mode in these modules. Fig. 13 shows a screenshot of the application.

Again it can be chosen which COM Port has to be used to control the module (region (1)). Region (2) is primary used for initial configuration. The button `reset` sends a `SYS_RESET_REQ` command to the device. This step causes a reset of the device. The button `send clearall` sets a startup option of the device, so that the device will clear any configuration parameters on the next reset. The combination of those two buttons is a way to bring the module in a known state as it will start with the default values. `readSettings` checks whether the device is a coordinator or not and which PAN ID is set.

The buttons `coord` and `endd` in region (3) send a number of commands to the device. They configure the module as a coordinator or end-device and register the application. These are all requirements to start the ZigBee stack. The ZigBee stack can be started with the `ZB_Stack_Start` button. If the device is set to a coordinator, the joining of other devices can be allowed with the `allowJoin` button. With the `getShortAdd` button the short address of a module can be recalled when it is part of

a formed network.

Region (4) includes the commands for testing of the network. With the `sendMsg` button one can send one byte to a device with the address given in the first two text-fields on the left side. In the *message* text-field the hex value of the byte, that has to be sent, can be typed in. The `sendMultMsg` button allows to send multiple messages to a device. The two text-fields on the left are for controlling how many messages in which interval have to be sent. `rundschreiben` simulates again sending messages individually to a lot of end-devices. As there were only three CC2480 devices working, it sends a message repetitive to the two end-devices.

The `readConfig` button in region (5) is for manually reading a configuration value from the module. It executes the ZB_READ_CONFIGURATION command with the ConfigID byte given in the text-field below.

Region (7) controls whether all serial data that is sent from the module to the computer will be shown in the text-field of region (6). If `listenToFieldMode` is activated, all data will be displayed in the text-field. If `listenSilent` is active, there is only an indication that data is coming from the module, but it will not be shown in the text-field.

5.4.1 Example of the Usage of the Application

The flowchart of fig. 14 shows how to use the application to control a CC2480 module. After proceeding these steps, the module is working as a coordinator or an end-device. If the module is working as a coordinator and other devices have to join its network the button `allowJoin` has to be pushed. This button works only after having started the ZigBee stack with `ZB_Stack_Start`.

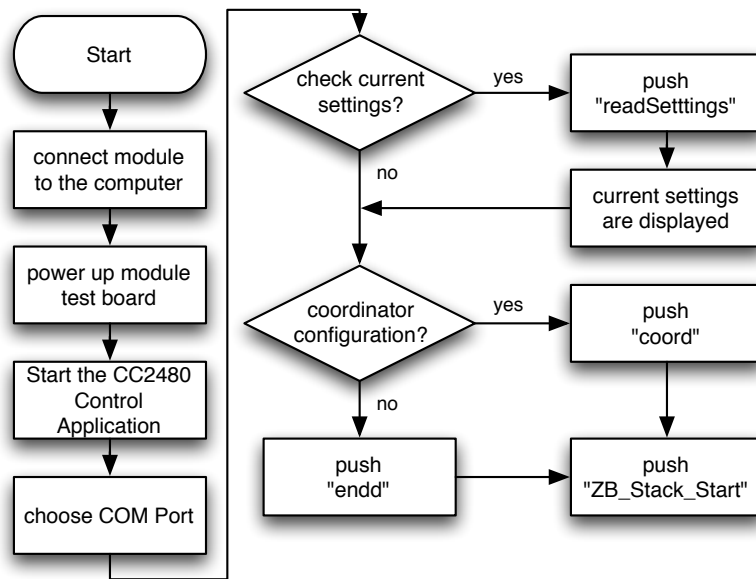


Figure 14: flowchart for using the application to control a CC2480 module

6 Tests and Evaluation

With the prepared PCBs and the applications it was possible to test different scenarios. The final goal of such a network is to control up to 2000 motors from a central point. There are different ways to realize this. One approach is to send to every motor a command (with the target coordinates) individually. The advantage of this is that it's always known whether every motor received the command or not, because of the acknowledging in the transmission protocol. A disadvantage is that there is a heavy load on the network. For every end-device (i.e. motor) the coordinator has to send a individually addressed command which is time-consuming.

Another approach is storing on every motor tables with coordinates in it. For each elevation angle that has to be reached with the telescope there is a record in an individual table for every motor. This way the coordinator has only to send a command like *go to elevation x*. Every motor can lookup the position where it has to go in its own local table. If it's realized this way, broadcasting can be used. When the coordinator broadcasts a message, all end-devices will listen to. The problem is that the coordinator doesn't know whether all devices have received the message or not. Broadcasting uses no acknowledging.

6.1 Test Scenarios with the Xbee Modules

6.1.1 Broadcasting

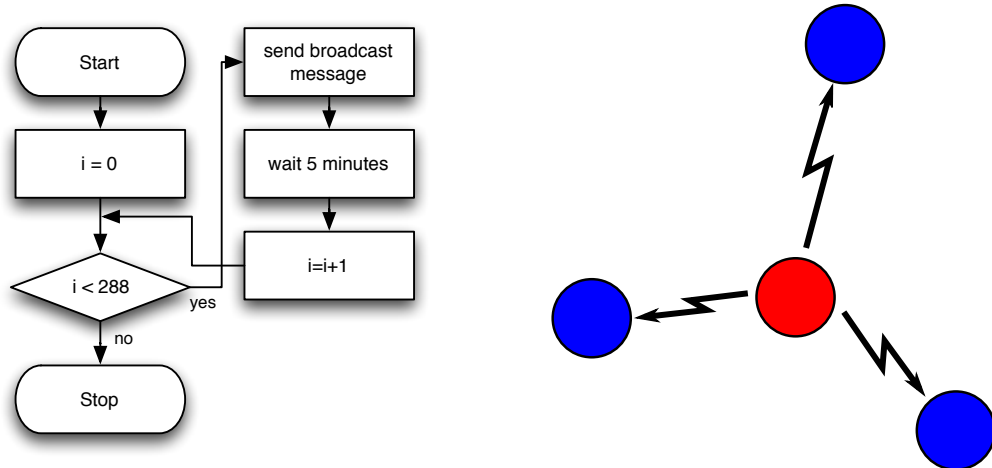


Figure 15: Left: flowchart for sending broadcast messages every 5 minutes (also valid for the CC2480 modules). Right: setup for the broadcast test. The coordinator is in red, the end-devices are in blue. The flashes symbolize a broadcast message that is sent to all end-devices simultaneous.

The first tests with the Xbee modules was made with the following scenario (see flowchart in fig. 15). One module worked as a coordinator and three others joined the network as end-devices. The coordinator sent a test-message every 5 minutes. It sent it via a broadcast command. One series of tests took 24 hours. This means 288 test-messages overall. These series have been repeated 5 times. The results are that almost all messages have been received from all the modules. Only in the first series one module missed one message. The reason why one module missed a message could be that it has not yet joined the network completely when the first broadcast message has been sent.

If controlling of the mirrors is realized this way, it is not directly known, whether all modules received the command for a new position. It has somehow different to be ensured that all mirrors are in the right position. This will be discussed in section 6.3.

6.1.2 Individual Addressing

The second approach controlling the mirrors is to send the positioning command to every motor individually. There are two main advantages of this method. First it is always known whether a module received a command or not. This is because every individual addressed message received a acknowledge answer of the destination module. The second point is, that the motors don't need to have a local table for the positioning. It is

sufficient when the mirror control center (which is connected to the network coordinator) knows where every motor has to go. It can then send the specific coordinates directly to the motor. This is very useful when something on the telescope changed so that internal calibration tables of the motors would be obsolete.

On the other hand, addressing every module individually increases the network load dramatically since there have to be contacted up to 2000 modules.

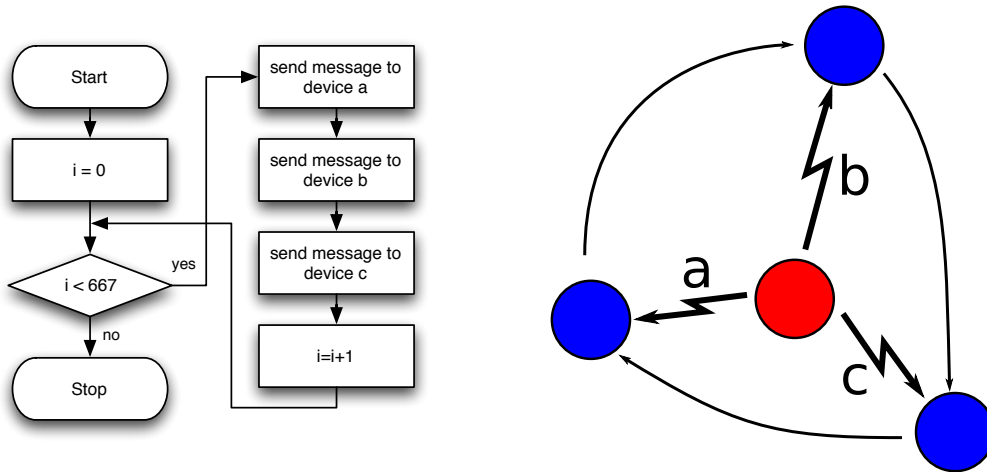


Figure 16: Left: flowchart for sending a message individually to every device. (also valid for the CC2480 modules). Right: setup for the test. The coordinator is in red, the end-devices in blue. The flashes symbolize sending messages successively to every end-device. After one turn, the sequence begins again at the first device until having completed 667 turns.

This situation has been simulated with the following approach (see fig. 16). There were not 2000 modules available. As a workaround, the three end-devices have been contacted one by one 667 times. This results in 2001 individual contacts. It turned out that this procedure is very sensitive to the COM-Port settings. Table 4 shows the results of the time consumption. If *extra thread* is set to *yes*, the application uses an extra thread to read the acknowledge answers from the module. This reading works independent from the sending command. The application doesn't wait for the acknowledge before it continues to send. If it is set to *no*, the applications sends the command to the first module, waits then for the response and not until received it, it will send the command to the next module. The reason for these different modes is that it has been discovered that the time consumption differs widely. This indicates that a lot of improvements can be realized with the application. These tests show already that it's possible to reach times in the region of 15 seconds. It's also evident that the time consumption is reciprocally proportional to the baud rate that has been used to communicate between the module and the computer.

Baud Rate [bps]	extra thread [boolean]	command length [bytes]	duration time [sec]
9600	yes	7	33
9600	yes	7	33
9600	yes	7	34
9600	yes	7	33
9600	yes	7	33
9600	yes	30	81
9600	yes	30	81
9600	yes	30	81
9600	yes	30	81
9600	yes	30	81
9600	yes	30	81
57600	yes	30	13.49
57600	yes	30	13.49
57600	yes	30	13.48
57600	yes	30	13.74
57600	yes	30	13.49
57600	yes	30	13.49
57600	no	30	31.98
57600	no	30	31.95
57600	no	30	31.98
57600	no	30	31.97
57600	no	30	32.02
57600	no	30	31.97

Table 4: Different test-conditions for the Xbee Module

6.2 Test Scenarios with the CC2480 Modules

6.2.1 Broadcasting

Also with the CC2480 modules a network has been formed to perform some tests. Three modules were available to form a network. One of the modules was the coordinator sending the data. The first test consists again to send broadcast messages to all the end-devices. It was the same procedure like for the XBee modules (fig. 15). 288 messages have been sent in a 5 minutes interval. This procedure has been repeated 5 times. All of them have been received by all the end-devices.

6.2.2 Individual Addressing

While trying to perform the test by individual addressing it turned out that there is a problem that increases the time consumption for sending data. First it has been formed a network with one coordinator and two end-devices. While trying to record the time consumption for sending 2000 messages it turned out that there are (relatively) long pauses between the send request and the moment when the packed had been received by the end-devices. This problem lies in an end-device specific way of communication: End-devices in ZigBee are designed to be battery powered⁹. Usually it's thought that end-devices can sleep most of the time. When something happens, they contact the coordinator (like a automatic fire alarm, where the fire sensors are connected to the end-devices). This means there is mainly a data transmission *from* the end-device *to* the coordinator. To enable the end-device to receive data too by saving energy it doesn't listen all the time to the coordinator for receiving data. It actually *polls* the coordinator for existing data. If the computer sends a send-data-request to the coordinator, the coordinator stores the message and waits until the end-device in question polls for data. Only now the data can be sent from the coordinator to the module. Even so the poll rate can be varied and set to a minimal value of 1 millisecond.

In the first series of tests there has been sent 1000 times a message to each of the two end-devices (simulating a contacting of 2000 nodes). The baud rate was set from the beginning to 115200 bps and the poll time mentioned above was changed to the minimum of 1 millisecond. The results are shown in table 5.

Table 5 shows that the time consumption for contacting 2000 nodes is about 2 minutes. This takes too long as the telescope has to set its position every five minutes. This would mean that it can only measure about 3 minutes in each position.

To check whether this long time is caused by the polling problem, the end-devices have changed to work as routers. Routers are designed to be connected to a power supply because they have to route data. Therefore they are always *on* and can receive data immediately. In table 6 can be seen that with this configuration the procedure can

⁹low power consumption is one of the design parameters of ZigBee

command length [bytes]	duration time [sec]
1	112
23	113
30	114

Table 5: individual addressing to end-devices

be accelerated dramatically. Contacting 2000 nodes needs then only about 45 seconds. It has to be kept in mind that this is not a way to realize the network in reality. In the

command length [bytes]	duration time [sec]
1	42
1	42
1	42
1	42
30	46
30	46
30	46
30	46

Table 6: individual addressing to routers

technical specifications of the CC2480 module it's written that every parent device can only have up to 6 routers as child devices.

6.3 Summarized Results

In principle it is possible to reach adequate fast speed to control up to 2000 motors. Both possibilities (i.e. broadcasting or individual addressing) seem to be ways to realize the wireless communication. In the case of using broadcasting there has to be thought about ways to ensure that all modules receive the positioning commands.

One example to solve this problem is the usage of a watchdog timer. Every module has to "know" how often it should receive a command. If the time expires without having received a command, it can contact the coordinator for requesting it. The coordinator will then send the command individually to only those modules that didn't receive the broadcast message.

Another way for having more control by using broadcasting is to send a counter

number with every command. The motors can compare all numbers coming from the coordinator. If the difference of the numbers between two commands is bigger than 1, the motor knows that it missed one command. In this case the motor can send a message to the coordinator and then it's known that one measurement will be not valid or at least incomplete.

By using individual addressing, the best time that can be reached, really depends on how the serial communication is implemented. It's not sure whether the fastest possible time has already been reached yet with the test applications. One crucial point is the implementation of the communication between the computer (or the microprocessor) and the rf data module. It depends not only on trivial settings like baud rate.

Furthermore there are also other ways to reduce the amount of data that has to be sent. One can think about giving the end-devices more "intelligence": a common scenario for example is that the telescope has to go to a specific position and after that, it has only to work against the earth rotation. In this case one could send a command to the end-devices to turn to a specific point. After that, they end-devices have to move the motors autonomously to correct earth rotation. In this case, there is no more data needed to be sent. Only at the beginning there has to be sent data to each motor.

Looking at the tests with the CC2480 module, it's difficult to say how fast a real network can be. In 3.2 it has been showed that according to the *CC2480 Developer's Guide* it's not possible to realize a simple star network with 2000 end-devices. The network has to be subdivided to different branches with routers. With only three devices it could't be figured out how long the propagation of a data packet is taken from the coordinator down to the last end-device (and the acknowledge back to the coordinator). Furthermore there is the polling problem with the end-devices that increases the time to control all nodes in a network. At least for the CC2480 module it has to be assumed that controlling all motors by individual addressing takes longer in such a big network.

6.4 Conclusion and Open Questions

In the authors opinion the Xbee modules seem to be the better choice. With the firmware that has been used, a simple star network can be formed. If individual addressing is preferred, they are faster. The fact that data has not to be routed from the coordinator to the end-device by other devices in the middle makes it easier to estimate how long it takes to reach all nodes. Having no devices in between reduces the network load. Another important point is that in a star network a corrupt device only affects the corresponding mirror. But if a router in a cluster tree network has a malfunction, all its child devices are separated from the coordinator unless dynamical reorganization is implemented.

This thesis is a first clarification whether a wireless network can be a way to realize the active mirror control of a telescope. There are still a lot of questions that have to be answered. For instance no serious tests were performed to test the range of such

networks. According to the data sheets of the two tested modules it should be no problem to handle distances given by the telescopes. With a battery powered test-board for the XBee module it was possible to transmit data over a distance higher than 40 meters. In this test was one module outdoors and one module indoors with a closed window in between. With both modules outdoors it was possible to overcome distances about 60 meters.

Because of a design error in the PCB, the antenna-part of the CC2480 module was not correctly working. With this modules it was only possible to transmit data over a maximum distance about 4 meters. But this result has nothing to do with the real values that could be reached.

It has also not been investigated how the real conditions, like a big metallic space frame for mounting the mirrors, could affect the working range of those modules.

A next important step would be to test the network in more realistic conditions. This means that one should build bigger networks to investigate how reliable the network is when there are hundreds of devices with distances between each other that are comparable to the telescope dimensions. Especially when using the broadcast method one could examine whether all end-devices still receive all messages or how often they miss one. Such a network allows also to test how long it takes until the whole network is ready after an electrical power outage for example.

References

- [1] <http://cta.physik.uzh.ch>.
- [2] <http://www.cta-observatory.org>.
- [3] http://imagine.gsfc.nasa.gov/docs/science/how_12/cerenkov.html.
- [4] Bluetooth SIG, Inc., *Specification of the Bluetooth System, BLUETOOTH SPECIFICATION Version 2.1 + EDR (vol 0)*, July 2007.
- [5] M. Bürge, *Standards drahtloser Übertragung: Von Bluetooth zu IEEE 802.15.4/Zig-Bee*, 2004.
- [6] http://de.wikipedia.org/wiki/IEEE_802.15.4.
- [7] Digi International Inc., *Xbee(R) / XBee-PRO(R) OEM RF Modules*, Sept. 2008. http://ftp1.digi.com/support/documentation/90000982_A.pdf.
- [8] Texas Instruments, *CC2480 Developer's Guide SWRA176*. <http://focus.ti.com/lit/an/swra176/swra176.pdf>.
- [9] Texas Instruments, *CC2480 Interface Specification (Rev. A)*. <http://focus.ti.com/lit/er/swra175a/swra175a.pdf>.
- [10] <http://www.cadsoft.de>.
- [11] National Semiconductors, *LM2937-2.5, LM2937-3.3 400mA and 500mA Voltage Regulators*, Aug. 2005. <http://www.national.com/ds/LM/LM2937-3.3.pdf>.
- [12] intersil, *ICL3221ECAZ Data Sheet*, Sept. 2008. <http://www.intersil.com/data/fn/fn4910.pdf>.

A Details of the PCB design

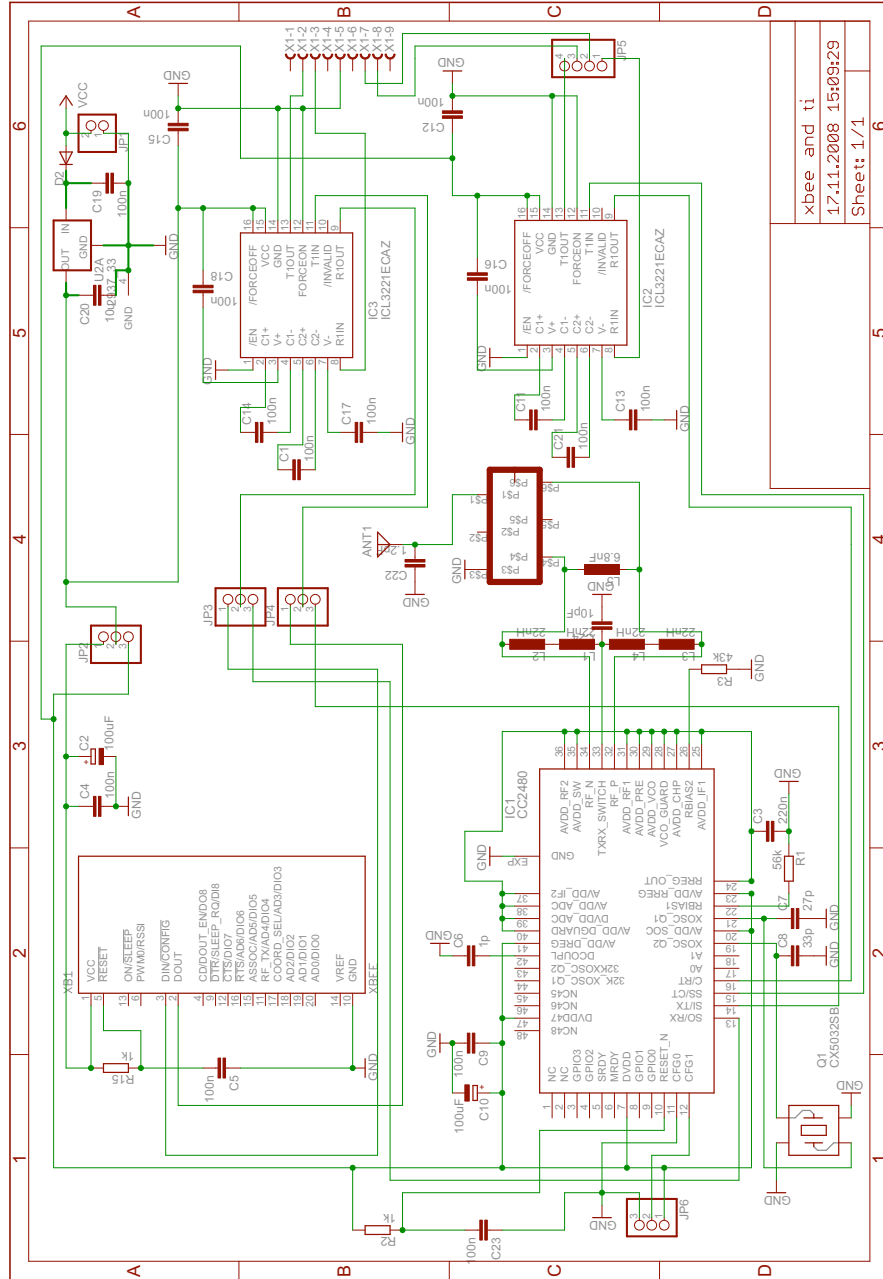


Figure 17: Circuit Diagram of the testboard

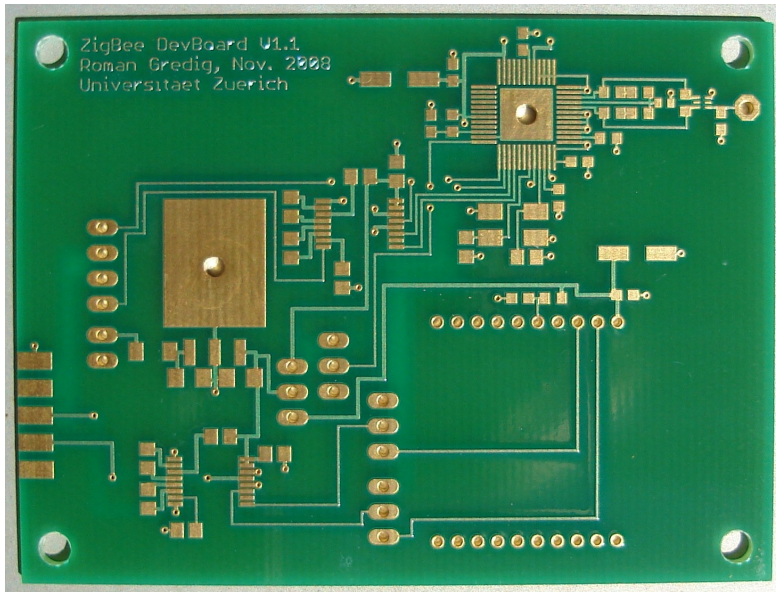


Figure 18: Picture of the unpopulated PCB

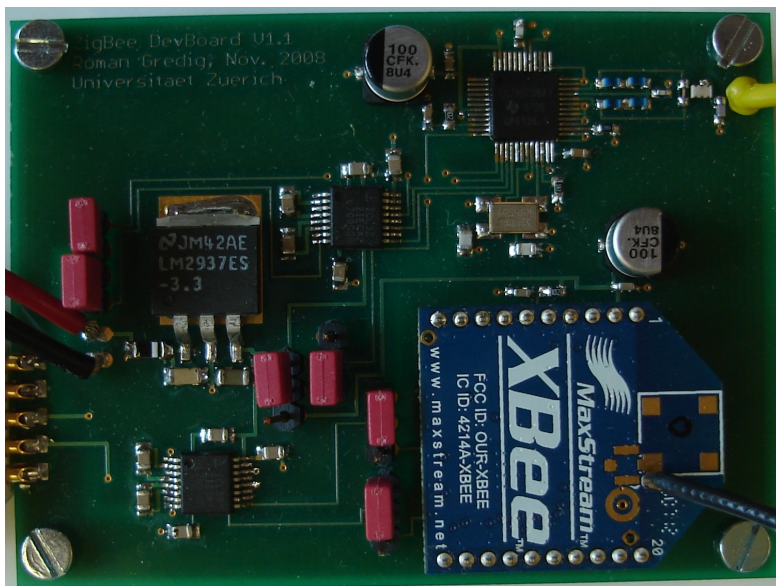


Figure 19: Picture of the populated PCB. The small IC on the upper right side is the CC2480 module, the blue on the lower right side is the XBee module.

B Listings

B.1 Xbee Application

Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
// von roman dazugetan
using System.Threading;

namespace serial_fuer_roman
{
    public partial class Form1 : Form
    {
        // instanzvariablen
        Boolean useActionHandler = true;
        APIControl api;
        String leseBuffer;
        ThreadStart del;
        Thread myFirstThread;
        int anzahlRunden;
        // Constructor
        public Form1()
        {
            // -----
            // Constructor
            // -----
            // GUI stuff (generated by VS)
            InitializeComponent();
            // init serial port
            serialPort1.BaudRate = 57600;
            // serialPort1.PortName = "COM1";
            serialPort1.Parity = System.IO.Ports.Parity.None;
            serialPort1.Handshake = System.IO.Ports.Handshake.None;
            serialPort1.StopBits = System.IO.Ports.StopBits.One;
            serialPort1.DataBits = 8;
            // serialPort1.
            // add EventHandler to serialport (recieve)
            serialPort1.DataReceived += new System.IO.Ports.SerialDataReceivedEventHandler(
                serialPort1_DataReceived);
            // serialPort1.Open();

            this.api = new APIControl(serialPort1, textBox1);

            // roman versucht multithreading

        }

        // EventHandler for DataReceived
        void serialPort1_DataReceived(object sender, System.IO.Ports.
            SerialDataReceivedEventArgs e)
        {
            // -----
            // Event Handler for serialport1 DataRecieved Event
            // -----
            // dirty hack, serial port event runs in different thread which is not allowed to
            // access gui elements; next line enables this
            CheckForIllegalCrossThreadCalls = false;
            // read all available data and append it to the textbox
            String neues = serialPort1.ReadExisting();
            textBox1.Text = textBox1.Text + neues;
            //textBox1.Text = textBox1.Text + serialPort1.ReadExisting();
        }
    }
}
```



```

// read current settings of a device and set the checkboxes
private void readCurrentSettings()
{
    Boolean status;
    String result;
    // normalen eventhandler entfernen
    serialPort1.DataReceived -= new System.IO.Ports.SerialDataReceivedEventHandler(
        serialPort1_DataReceived);
    useActionHandler = false;
    try
    {
        // entering command mode
        System.Threading.Thread.Sleep(1100);
        serialPort1.Write("+++");
        System.Threading.Thread.Sleep(1100);

        serialPort1.DiscardInBuffer();
        serialPort1.Write("ATMY\r");
        textBox3.Text = serialPort1.ReadTo("\r");

        serialPort1.DiscardInBuffer();
        serialPort1.Write("ATDH\r");
        textBox4.Text = serialPort1.ReadTo("\r");

        serialPort1.DiscardInBuffer();
        serialPort1.Write("ATDL\r");
        textBox5.Text = serialPort1.ReadTo("\r");

        serialPort1.DiscardInBuffer();
        serialPort1.Write("ATCE\r");
        result = serialPort1.ReadTo("\r");
        if (result.Contains("0"))
        {
            checkBox1.Checked = false;
        }
        else
        {
            checkBox1.Checked = true;
        }

        serialPort1.DiscardInBuffer();
        serialPort1.Write("ATA2\r");
        result = serialPort1.ReadTo("\r");
        if (result.Contains("2"))
        {
            checkBox2.Checked = false;
        }
        else
        {
            checkBox2.Checked = true;
        }

        // leave command mode
        serialPort1.Write("ATCN\r");
    }
    catch (Exception excep)
    {
        MessageBox.Show("You must first choose an unused COM Port.", "Serial_Interface_
            Error",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
    // normal eventhandler wieder aktivieren
    serialPort1.DataReceived += new System.IO.Ports.SerialDataReceivedEventHandler(
        serialPort1_DataReceived);
    useActionHandler = true;
}

// opens new comport
private void changeComport(String port, RadioButton button)
{
    serialPort1.Close();
    serialPort1.PortName = port;
    try

```

B Listings

```
    {
        serialPort1.Open();
        textBox1.Text = textBox1.Text + "\r\ncomport_changed_to_" + port + "\r\n";
    }
    catch (Exception excep)
    {
        MessageBox.Show("Failed_to_open_" + port + ".\nMaybe_it's_already_in_use.", "
            Serial_Interface_Error",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        button.Checked = false;
    }
}

// sends at commands to the serial port
private void sendSerialCommand(String command)
{
    try
    {
        // new line in Status text field
        textBox1.Text = textBox1.Text + "\r\n";
        // entering command mode
        System.Threading.Thread.Sleep(1100);
        serialPort1.Write("+++");
        System.Threading.Thread.Sleep(1100);

        // send commands
        serialPort1.Write(command + "\r");

        // exit at command mode
        serialPort1.Write("ATCN\r\n");
    }
    catch (Exception e)
    {
        MessageBox.Show("You_must_first_choose_an_unused_COM_Port.", "Serial_Interface_
            Error",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
}

private void sendSerialDataLine(String data)
{
    try
    {
        serialPort1.Write(data + "\r\n");
    }
    catch (Exception e)
    {
        MessageBox.Show("You_must_first_choose_an_unused_COM_Port.", "Serial_Interface_
            Error",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
}

}
private void btnStatus_Click(object sender, EventArgs e)
{
    // -----
    // status button callback
    // -----
    // send command sequence to chip (writeline automatically appends \r\n, eg newline)

    textBox1.Text = textBox1.Text + "\r\n";
    sendSerialCommand("ATSH,SL");
}

// send string to selected comport
private void button1_Click(object sender, EventArgs e)
{
    sendSerialDataLine(textBox2.Text);
}
```

```
}

// The radiobuttons
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton1.Checked)
    {
        changeComport("COM1", radioButton1);
    }
}

private void radioButton2_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton2.Checked)
    {
        changeComport("COM4", radioButton2);
    }
}

private void radioButton3_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton3.Checked)
    {
        changeComport("COM2", radioButton3);
    }
}
// end of the radiobuttons

// clear the textbox
private void button2_Click(object sender, EventArgs e)
{
    textBox1.Text = "";
}

// setting the coordinator modus
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
    if (useActionHandler)
    {
        if (checkBox1.Checked) // enable coordinator mode
        {
            sendSerialCommand("ATCE1");
        }
        else // disable coordinator mode
        {
            sendSerialCommand("ATCE0");
        }
    }
}

// checkbox "allow association"
private void checkBox2_CheckedChanged(object sender, EventArgs e)
{
    if (useActionHandler)
    {
        if (checkBox2.Checked) // allow association
        {
            sendSerialCommand("ATA26");
        }
        else // disable association
        {
            sendSerialCommand("ATA22");
        }
    }
}

// set my adress
private void button3_Click(object sender, EventArgs e)
{
    String command = "ATMY" + textBox3.Text;
    sendSerialCommand(command);
}
```

B Listings

```
// set DH address
private void button4_Click(object sender, EventArgs e)
{
    String command = "ATDH" + textBox4.Text;
    sendSerialCommand(command);
}

// set DL address
private void button5_Click(object sender, EventArgs e)
{
    String command = "ATDL" + textBox5.Text;
    sendSerialCommand(command);
}

// enable broadcasting
private void button6_Click(object sender, EventArgs e)
{
    String command = "ATDLFFFF,DH0";
    sendSerialCommand(command);
}

// read MY Address
private void button7_Click(object sender, EventArgs e)
{
    sendSerialCommand("ATMY");
    /*
    // begin romans test
    string[] fenster;
    fenster = textBox1.Text.Split('\r');
    int groesse;
    groesse = fenster.GetLength(0);
    // end romans test
    */
}

// read DH
private void button8_Click(object sender, EventArgs e)
{
    sendSerialCommand("ATDH");
}

// read DL
private void button9_Click(object sender, EventArgs e)
{
    sendSerialCommand("ATDL");
}

// Broadcast test
private void button10_Click(object sender, EventArgs e)
{
    try
    {
        int zahler = Convert.ToInt32(textBox6.Text);
        int intervall = Convert.ToInt32(textBox8.Text) * 1000;
        sendSerialDataLine("Broadcasting_the_message_" + zahler + "_times");
        for (int i = 0; i < zahler; i++)
        {
            sendSerialDataLine("Broadcast_message_Nr." + (i+1) + "_" + DateTime.Now.
                ToShortTimeString());
            Thread.Sleep(intervall);
        }
    }
    catch (Exception excep)
    {
        MessageBox.Show("you_must_enter_an_integer.", "Parsing_Error",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
}

// manually read current settings
private void button11_Click(object sender, EventArgs e)
{
    readCurrentSettings();
}

// send given command
private void button12_Click(object sender, EventArgs e)
```

```

{
    sendSerialCommand(textBox7.Text);
}

private void button13_Click(object sender, EventArgs e)
{
    del = new ThreadStart(threadrec);
    myFirstThread = new Thread(del);

    // this.sendSerialCommand("ATAP1");

    // normalen eventhandler entfernen machen
    // serialPort1.DataReceived -= new System.IO.Ports.SerialDataReceivedEventHandler(
        serialPort1_DataReceived);

    // debug eventhandler einschalten
    // serialPort1.DataReceived += new System.IO.Ports.SerialDataReceivedEventHandler(
        debug_eventhandler);

    anzahlRunden = Convert.ToInt32(textBox9.Text);

    // textBox1.Text = textBox1.Text + "\r\n Ohne Antwort auslesen";
    DateTime startZeit = DateTime.Now;
    DateTime stopZeit;

    textBox1.Text = textBox1.Text + "\r\nStart-time:_" + startZeit.ToLongTimeString();
    // DateTime.Now.ToLongTimeString();

    // if (!myFirstThread.IsAlive) myFirstThread.Start();
    myFirstThread.Start();
    while (!myFirstThread.IsAlive)
    {
        // warte, bis er laeuft
    }
    leseBuffer = "";
    for (int i = 0; i < anzahlRunden; i++)
    {
        api.transmitData(0x00, 0x01, 0x00, new byte[] { 0x43, 0x48, 0x49, 0x50, 0x30, 0
            x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x30, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0
            x50, 0x30, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x30, 0x5B, 0x5D, 0x0D, 0
            x0A });
        // this.readByte(7, true);

        api.transmitData(0x00, 0x02, 0x00, new byte[] { 0x43, 0x48, 0x49, 0x50, 0x32, 0
            x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x32, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0
            x50, 0x32, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x32, 0x5B, 0x5D, 0x0D, 0
            x0A });
        // this.readByte(7, true);

        api.transmitData(0x00, 0x03, 0x00, new byte[] { 0x43, 0x48, 0x49, 0x50, 0x31, 0
            x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x31, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0
            x50, 0x31, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x31, 0x5B, 0x5D, 0x0D, 0
            x0A });
        // this.readByte(7, true);
    }
    stopZeit = DateTime.Now;
    textBox1.Text = textBox1.Text + "\r\nStop-time:_" + stopZeit.ToLongTimeString();
    textBox1.Text = textBox1.Text + "\r\nDifferenz:_" + stopZeit.Subtract(startZeit)
        .ToString() + "\r\n";

    // debug eventhandler deaktivieren
    // serialPort1.DataReceived -= new System.IO.Ports.SerialDataReceivedEventHandler(
        debug_eventhandler);

    // normal eventhandler wieder aktivieren
    // serialPort1.DataReceived += new System.IO.Ports.SerialDataReceivedEventHandler(
        serialPort1_DataReceived);
    myFirstThread.Join();
    // normal eventhandler wieder aktivieren
    serialPort1.DataReceived += new System.IO.Ports.SerialDataReceivedEventHandler(
        serialPort1_DataReceived);
    // sendSerialCommand("ATAP0");
}

```

B Listings

```
private void readByte(int anzahl, bool anzeigen)
{
    byte[] answer = new byte[anzahl];
    for (int i = 0; i < anzahl; i++)
    {
        answer[i] = (byte)serialPort1.ReadByte();
    }

    if (anzeigen) textBox1.Text = textBox1.Text + "\r\n" + System.BitConverter.ToString(
        answer);
}

private void button14_Click(object sender, EventArgs e)
{
    // normalen eventhandler entfernen machen
    serialPort1.DataReceived -= new System.IO.Ports.SerialDataReceivedEventHandler(
        serialPort1.DataReceived);

    anzahlRunden = Convert.ToInt32(textBox9.Text);

    DateTime startZeit = DateTime.Now;
    DateTime stopZeit;

    textBox1.Text = textBox1.Text + "\r\nStart_time:_ " + startZeit.ToLongTimeString();
    // DateTime.Now.ToLongTimeString();
    leseBuffer = "";
    serialPort1.DiscardInBuffer();
    for (int i = 0; i < anzahlRunden; i++)
    {
        api.transmitData(0x00, 0x01, 0x00, new byte[] { 0x43, 0x48, 0x49, 0x50, 0x30, 0
            x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x30, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0
            x50, 0x30, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x30, 0x5B, 0x5D, 0x0D, 0
            x0A });
        this.readByte(7, false);

        api.transmitData(0x00, 0x02, 0x00, new byte[] { 0x43, 0x48, 0x49, 0x50, 0x32, 0
            x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x32, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0
            x50, 0x32, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x32, 0x5B, 0x5D, 0x0D, 0
            x0A });
        this.readByte(7, false);

        api.transmitData(0x00, 0x03, 0x00, new byte[] { 0x43, 0x48, 0x49, 0x50, 0x31, 0
            x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x31, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0
            x50, 0x31, 0x5B, 0x5D, 0x43, 0x48, 0x49, 0x50, 0x31, 0x5B, 0x5D, 0x0D, 0
            x0A });
        this.readByte(7, false);
    }
    stopZeit = DateTime.Now;
    textBox1.Text = textBox1.Text + "\r\nStop_time:_ " + stopZeit.ToLongTimeString();
    textBox1.Text = textBox1.Text + "\r\nDifferenz:_ " + stopZeit.Subtract(startZeit).
        ToString() + "\r\n";

    // normal eventhandler wieder aktivieren
    serialPort1.DataReceived += new System.IO.Ports.SerialDataReceivedEventHandler(
        serialPort1.DataReceived);
    // sendSerialCommand("ATAP0");
}

void debug_eventhandler(object sender, System.IO.Ports.SerialDataReceivedEventArgs e)
{
    // -----
    // Event Handler for serialport1 DataRecieved Event
    // -----
    // dirty hack, serial port event runs in different thread which is not allowed to
    // access gui elements; next line enables this
    CheckForIllegalCrossThreadCalls = false;
    // textBox1.Text = textBox1.Text + "Actionhandler fired" ;

    //int anzahl = serialPort1.BytesToRead;
    //byte[] answer = new byte[anzahl];
    //for (int i = 0; i < anzahl; i++)
    //{
    //    answer[i] = (byte)serialPort1.ReadByte();
    //    if (answer[i] == 0x7E) textBox1.Text = textBox1.Text + "\r\n";
    //    textBox1.Text = textBox1.Text + "-" + answer[i].ToString("X");
    //}
}
```

```

    //}
    leseBuffer = leseBuffer + serialPort1.ReadExisting();

    // textBox1.Text = textBox1.Text + System.BitConverter.ToString(answer);
}

public void threadrec()
{
    CheckForIllegalCrossThreadCalls = false;
    // normalen eventhandler entfernen machen
    serialPort1.DataReceived -= new System.IO.Ports.SerialDataReceivedEventHandler(
        serialPort1_DataReceived);

    serialPort1.DiscardInBuffer();
    // geruempel ignorieren

    int gotAck = 0;
    int anzahlLeserunden = 3 * anzahlRunden;
    // einlesen aller antworten
    byte[,] answer = new byte[anzahlLeserunden, 7];
    textBox1.Text = textBox1.Text + "\r\n";
    for (int i = 0; i < anzahlLeserunden; i++)
    {
        for (int j = 0; j < 7; j++)
        {
            answer[i, j] = (byte)serialPort1.ReadByte();
        }
    }

    // ueberpruefe statusbyte
    for (int i = 0; i < anzahlLeserunden; i++)
    {
        if (answer[i, 5] == (byte)0x00) gotAck++;
    }

    textBox1.Text = textBox1.Text + "\r\n" + "Anzahl_Antwort_en:_" + gotAck;

    //// myFirstThread.S
    //for (int i = 0; i < anzahlLeserunden; i++)
    //{
    //    for (int j = 0; j < 7; j++)
    //    {
    //        textBox1.Text = textBox1.Text + answer[i, j].ToString("X");
    //    }
    //}
}
}
}

```

APIControl.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace serial_fuer_roman
{
    class APIControl
    {
        // instantzvariabeln
        System.IO.Ports.SerialPort serialPort1;
        System.Windows.Forms.TextBox box;
        byte startDelimiter;

        // konstruktormethode
        public APIControl(System.IO.Ports.SerialPort serialPort1, System.Windows.Forms.TextBox
            box)
        {
            this.serialPort1 = serialPort1;
            this.box = box;
            this.startDelimiter = 0x7E;
        }

        // returns checksum from a frame array
        private byte checkSum(byte[] frame)
        {
            byte summe = 0x00;
            for (int i = 0; i < frame.Length; i++)
            {
                summe += frame[i];
            }

            byte result = (byte)((byte)0xFF - summe);
            return result;
        }

        // send a command to the serial interface
        public void sendCommand(byte[] length, byte[] frame)
        {
            byte[] buffer = new byte[4 + frame.Length];

            buffer[0] = startDelimiter;
            buffer[1] = length[0];
            buffer[2] = length[1];

            frame.CopyTo(buffer, 3);
            buffer[buffer.Length - 1] = this.checkSum(frame);
            serialPort1.Write(buffer, 0, buffer.Length);
        }

        public void transmitData(byte dh, byte dl, byte opt, byte[] data)
        {
            byte[] frame = new byte[5 + data.Length];
            byte[] length = new byte[2];
            length[0] = 0x00;

            frame[0] = 0x01;
            frame[1] = 0x52; // frameid
            frame[2] = dh;
            frame[3] = dl;
            frame[4] = opt; // option bit ??
            data.CopyTo(frame, 5);

            length[1] = (byte)frame.Length;
            this.sendCommand(length, frame);
        }

        public void leaveApiMode()
        {
            byte[] frame = new byte[5];
            frame[0] = 0x08;
            frame[1] = 0x52;
            frame[2] = 0x41; //A
            frame[3] = 0x50; //P
            frame[4] = 0x30; //O null
            byte[] length = {0x00, 0x05};
        }
    }
}
```



```
        this.sendCommand(length, frame);  
    }  
}
```

B.2 CC2480 Application

Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using System.IO;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        // instanzvariablen
        CcComands cccontrol;
        bool useEventHandler;

        int runden;

        public Form1()
        {
            // Constructor
            InitializeComponent();

            // kommandoclassenobjekt machen

            //
            this.cccontrol = new CcComands(serialPort1, textBox1);
            useEventHandler = true;
        }

        // Eventhandler, wenn was gekommen ist.
        void serialPort1_DataReceived(object sender, System.IO.Ports.
            SerialDataReceivedEventArgs e)
        {
            // -----
            // Event Handler for serialport1 DataRecieved Event
            // -----
            // dirty hack, serial port event runs in different thread which is not allowed to
            // access gui elements; next line enables this
            CheckForIllegalCrossThreadCalls = false;
            // textBox1.Text = textBox1.Text + "Actionhandler fired" ;

            int anzahl = serialPort1.BytesToRead;
            byte[] answer = new byte[anzahl];
            for (int i = 0; i < anzahl; i++)
            {
                answer[i] = (byte)serialPort1.ReadByte();
                if (answer[i] == 0xFE) textBox1.Text = textBox1.Text + "\r\n";
            }

            textBox1.Text = textBox1.Text + System.BitConverter.ToString(answer);
        }

        // button1 for debugging
        private void button1_Click(object sender, EventArgs e)
        {
            //byte [] command = {1,2,3};
            byte configID = 0x24;
            byte len = 0x02;
            byte[] value = {0x01, 0x00};
            byte val0 = 0x01;
            byte val1 = 0x00;
            cccontrol.zb_write_configuration(configID, len, value);
        }
    }
}
```

```

// button rf test
private void button2_Click(object sender, EventArgs e)
{
    // get device info
    // byte[] answer = cccontrol.zb_read_configuration(0x03);
    // textBox2.Text = System.BitConverter.ToString(answer);
    textBox1.Text = textBox1.Text + "test1_\r\n";
    cccontrol.test_rf(0x00, 0x00, 0x9F);
    textBox1.Text = textBox1.Text + "test2_\r\n";
    cccontrol.test_rf(0x01, 0x00, 0x9F);

    textBox1.Text = textBox1.Text + "test3_\r\n";
    cccontrol.test_rf(0x02, 0x00, 0x9F);
}

// opens the given comport
private void changeComport(String port, RadioButton button)
{
    serialPort1.Close();
    serialPort1.PortName = port;
    try
    {
        serialPort1.Open();
        textBox1.Text = textBox1.Text + "\r\ncomport_changed_to_" + port + "\r\n";
    }
    catch (Exception excep)
    {
        MessageBox.Show("Failed_to_open_" + port + ".\nMaybe_it's_already_in_use.", "
            Serial_Interface_Error",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
        button.Checked = false;
    }
}

#region ComPort Radio buttons
// RadioButtons
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton1.Checked)
    {
        changeComport("COM1", radioButton1);
    }
}

private void radioButton2_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton2.Checked)
    {
        changeComport("COM2", radioButton2);
    }
}

private void radioButton3_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton3.Checked)
    {
        changeComport("COM3", radioButton3);
    }
}

private void radioButton4_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton4.Checked)
    {
        changeComport("COM4", radioButton4);
    }
}

// end radiobuttons
#endregion

// send SREQ without data
private void sendSREQ(byte[] command, int answerLength)
{

```

B Listings

```
// calculate command
byte sof = (byte)0xfe;
byte length = 0x00;
byte cmd0 = command[1];
byte cmd1 = command[2];

byte fcs;
fcs = (byte)(length ^ cmd0 ^ cmd1);
byte[] buffer = { sof, length, cmd0, cmd1, fcs };
serialPort1.DiscardInBuffer();
serialPort1.Write(buffer, 0, 5);
this.getResponse(answerLength);
}

// get response
private string getResponse(int answerLength)
{
    try
    {
        byte[] answer = new byte[answerLength];
        for (int i = 0; i < answerLength; i++)
        {
            answer[i] = (byte)serialPort1.ReadByte();
        }
        return System.BitConverter.ToString(answer);
    }
    catch (Exception ex)
    {
        return "error"; // errorhandling
    }
}

// Coordinator Checkbox
private void checkBox1.CheckedChanged(object sender, EventArgs e)
{
}

// read in configurations
private void button3_Click(object sender, EventArgs e)
{
    // first disable eventhandlers
    useEventHandler = false;

    // coordinator check:
    byte[] answer;
    answer = cccontrol.zb_read_configuration(0x87);

    // debug info
    // textBox2.Text = System.BitConverter.ToString(answer);
    if (answer[7].Equals(0x00))
    {
        checkBox1.Checked = true;
    }
    else
    {
        checkBox1.Checked = false;
    }

    answer = cccontrol.zb_read_configuration(0x83);
    byte[] panID = new byte[2];
    panID[0] = answer[7];
    panID[1] = answer[8];
    textBox3.Text = panID[0].ToString("X");
    textBox4.Text = panID[1].ToString("X");

    // finally, enable eventhandlers
    useEventHandler = true;
}
}
```

```

// setting the PAN ID
private void button4_Click(object sender, EventArgs e)
{
    byte panID0 = byte.Parse(textBox3.Text, System.Globalization.NumberStyles.HexNumber);
    byte PANID1 = byte.Parse(textBox4.Text, System.Globalization.NumberStyles.HexNumber);
    cccontrol.zb_write_configuration(0x83, 0x02, new byte[] { panID0, PANID1 });
}

// listen mode eventhandler control
private void checkBox2_CheckedChanged(object sender, EventArgs e)
{
    if (checkBox2.Checked)
    {
        serialPort1.DataReceived += new System.IO.Ports.SerialDataReceivedEventHandler(
            serialPort1.DataReceived);
    }else
    {
        serialPort1.DataReceived -= new System.IO.Ports.SerialDataReceivedEventHandler(
            serialPort1.DataReceived);
    }
}

// zigbee stack start
private void button5_Click(object sender, EventArgs e)
{
    checkBox2.Checked = true;
    // cccontrol.zb_app_register_request();
    cccontrol.zb_start_request();
}

// sends a test message to the given short address
private void button6_Click(object sender, EventArgs e)
{
    if (!checkBox2.Checked) checkBox2.Checked = true;
    byte message = byte.Parse(textBox8.Text, System.Globalization.NumberStyles.HexNumber);
    byte short0 = byte.Parse(textBox5.Text, System.Globalization.NumberStyles.HexNumber);
    byte short1 = byte.Parse(textBox6.Text, System.Globalization.NumberStyles.HexNumber);
    cccontrol.zb_send_data_request(new byte[] { short0, short1 }, message);
}

// get short address
private void button7_Click(object sender, EventArgs e)
{
    if (checkBox2.Checked) checkBox2.Checked = false;
    cccontrol.zb_get_device_info();
    byte[] answer = new byte[14];
    for (int i = 0; i < 14; i++)
    {
        answer[i] = (byte)serialPort1.ReadByte();
    }

    byte[] address = new byte[] { answer[5], answer[6], answer[7], answer[8], answer[9], answer[10], answer[11], answer[12]};
    textBox1.Text = textBox1.Text + "\r\n" + System.BitConverter.ToString(answer);
    textBox7.Text = System.BitConverter.ToString(address);
}

// reset button
private void button8_Click(object sender, EventArgs e)
{
    if (!checkBox2.Checked) checkBox2.Checked = true;
    textBox1.Text = textBox1.Text + "Resetting Device\r\n";
    cccontrol.sys_reset_req();
}

// clear configs (needs reset)

```

```

private void button9_Click(object sender, EventArgs e)
{
    if (checkBox2.Checked) checkBox2.Checked = false;
    textBox1.Text = textBox1.Text + "schicke_alles_loeschen\r\n";
    cccontrol.zb_write_configuration(0x03, 0x01, new byte[] {0x03});
}

// clears the main text fiels
private void button10_Click(object sender, EventArgs e)
{
    textBox1.Text = "";
}

// button sendMultimesg
private void button11_Click(object sender, EventArgs e)
{
    if (!checkBox2.Checked) checkBox2.Checked = true;
    byte message = byte.Parse(textBox8.Text, System.Globalization.NumberStyles.
        HexNumber);
    byte short0 = byte.Parse(textBox5.Text, System.Globalization.NumberStyles.HexNumber
    );
    byte short1 = byte.Parse(textBox6.Text, System.Globalization.NumberStyles.HexNumber
    );

    int zahler = Convert.ToInt32(textBox9.Text);
    int intervall = Convert.ToInt32(textBox10.Text) * 1000;

    textBox1.Text = textBox1.Text + "\r\nsending_the_message_" + zahler + "_times";
    for (int i = 0; i < zahler; i++)
    {
        //cccontrol.zb_send_data_request(new byte[] { short0, short1 }, message);
        cccontrol.zb_send_data_request(new byte[] { short0, short1 }, new byte[] { 0xAA,
            0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0
            xAA, 0xAA, 0xAA, 0xAA, 0xAA });
        Thread.Sleep(intervall);
    }
}

// button rundschreiben (destination addresses hardcoded
private void button12_Click(object sender, EventArgs e)
{
    if (checkBox2.Checked) checkBox2.Checked = false;

    runden = Convert.ToInt32(textBox12.Text);

    string[] antwort = new string[runden];

    textBox1.Text = "Start_time:_" + DateTime.Now.ToLongTimeString();
    for (int j = 0; j < runden; j++)
    {
        // dieser chip ging kaputt
        //cccontrol.zb_send_data_request(new byte[] { 0x3E, 0x14 }, 0xBB); // 1 byte
        message
        // cccontrol.zb_send_data_request(new byte[] { 0x3E, 0x14 }, new byte[] { 0xAA,
        0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0
        xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0
        xAA, 0xAA, 0xAA, 0xAA }); // multi byte message
        //this.getResponse(5); // just ack of the sending command
        //antwort[0, j] = this.getResponse(7); // packet transmission status

        //cccontrol.zb_send_data_request(new byte[] { 0x01, 0x00 }, 0xCC);
        cccontrol.zb_send_data_request(new byte[] { 0x01, 0x00 }, new byte[] { 0xDD, 0
        xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0
        xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0xDD, 0
        xDD, 0xDD, 0xDD, 0xDD });

        this.getResponse(5); // just ack of the sending command
        antwort[j] = this.getResponse(7); // packet transmission status
    }

    textBox1.Text = textBox1.Text + "\r\n" + "Stop_time:_" + DateTime.Now.

```

```

        ToLongTimeString();

// check the answers
int erfolge = 0;
for (int i = 0; i < runden; i++)
{
    if (antwort[i] == "FE-02-46-83-01-00-C6") erfolge++;
}

textBox1.Text = textBox1.Text + "\r\n" + "Erfolge:_" + erfolge;

// export answers to txt file

//using (StreamWriter sw = File.CreateText("ausgabe1.txt"))
//{
//    for (int i = 0; i < runden; i++)
//    {
//        sw.WriteLine(antwort[1, i]);
//    }
//}
//    sw.Close();
//}

}

// reads the config with the params from the textfields
private void button13_Click(object sender, EventArgs e)
{
    byte configID = byte.Parse(textBox1.Text, System.Globalization.NumberStyles.
        HexNumber);
    cccontrol.zb_read_configuration(configID);
    textBox1.Text = textBox1.Text + "\r\n" + System.BitConverter.ToString(cccontrol.
        zb_read_configuration(configID));
}

#region silent listening
private void checkBox4_CheckedChanged(object sender, EventArgs e)
{
    if (checkBox4.Checked)
    {
        serialPort1.DataReceived += new System.IO.Ports.SerialDataReceivedEventHandler(
            serialPort1_DataReceived_silent);
    }
    else
    {
        serialPort1.DataReceived -= new System.IO.Ports.SerialDataReceivedEventHandler(
            serialPort1_DataReceived_silent);
    }
}

// eventhandler for silent listening
void serialPort1_DataReceived_silent(object sender, System.IO.Ports.
    SerialDataReceivedEventArgs e)
{
    CheckForIllegalCrossThreadCalls = false;
    serialPort1.ReadExisting();
    // textBox1.Text = textBox1.Text + "=";
    // if (textBox1.Text.Length > 20) textBox1.Text = "";
}
#endregion

// send commands to set to coordinator mode
private void button14_Click(object sender, EventArgs e)
{
    checkBox2.Checked = false;

    // set the chip to coordinator mode
    byte[] value = { 0x00 };
    cccontrol.zb_write_configuration(0x87, 0x01, value);

    // set poll time
    byte configID = 0x24;
    byte len = 0x02;
    byte[] value2 = { 0x01, 0x00 };

```

B Listings

```
        cccontrol.zb_write_configuration(configID, len, value2);

        // set the application for coordinator
        cccontrol.zb_app_register_request(true);

    }

    // sets the chip to router mode
    private void button15_Click(object sender, EventArgs e)
    {
        checkBox2.Checked = false;

        // set the chip to router mode
        byte[] value = { 0x01 };
        cccontrol.zb_write_configuration(0x87, 0x01, value);

        // set poll time
        byte configID = 0x24;
        byte len = 0x02;
        byte[] value2 = { 0x01, 0x00 };
        cccontrol.zb_write_configuration(configID, len, value2);

        // set the application for coordinator
        cccontrol.zb_app_register_request(false);
    }

    // turn on joining.
    private void button16_Click(object sender, EventArgs e)
    {
        cccontrol.zb_permit_joining_request(true);
    }
}
}
```


CCComands.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    class CCComands
    {
        //instantzvariabeln
        System.IO.Ports.SerialPort serialPort1;
        System.Windows.Forms.TextBox box;
        byte sof;

        // Konstruktor
        public CCComands(System.IO.Ports.SerialPort serialPort1, System.Windows.Forms.TextBox
            box)
        {
            this.serialPort1 = serialPort1;
            this.box = box;
            this.sof = (byte)0xfe;
        }

        public void sys_version()
        {
            // calculate command

            byte length = 0x00;
            byte cmd0 = 0x21;
            byte cmd1 = 0x02;

            byte fcs;
            fcs = (byte)(length ^ cmd0 ^ cmd1);
            byte[] buffer = { sof, length, cmd0, cmd1, fcs };

            //serialPort1.DiscardInBuffer();
            //serialPort1.Write(buffer, 0, 5);
            this.schreibeBuffer(buffer, 0, 5, true);
            readResult(10);
        }

        public void sys_reset_req()
        {
            box.Text = box.Text + "Resetting_device\r\n";
            byte length = 0x01;
            byte cmd0 = 0x41;
            byte cmd1 = 0x00;
            byte data = 0x00;

            byte fcs;
            fcs = (byte)(length ^ cmd0 ^ cmd1 ^ data);
            byte[] buffer = { sof, length, cmd0, cmd1, data, fcs };
            //serialPort1.DiscardInBuffer();
            //serialPort1.Write(buffer, 0, 6);
            this.schreibeBuffer(buffer, 0, 6, false);
            // readResult(11);
        }

        public byte[] zb_read_configuration(byte configID)
        {
            try
            {
                box.Text = box.Text + "Reading_configuration\r\n" + configID;
                byte length = 0x01;
                byte cmd0 = 0x26;
                byte cmd1 = 0x04;
                byte data = configID; // 1 byte: ist configid

                byte fcs;
                fcs = (byte)(length ^ cmd0 ^ cmd1 ^ data);
                byte[] buffer = { sof, length, cmd0, cmd1, data, fcs };
                serialPort1.DiscardInBuffer();
            }
        }
    }
}

```

B Listings

```
        serialPort1.Write(buffer, 0, 6);

        sof = (byte)serialPort1.ReadByte();
        length = (byte)serialPort1.ReadByte();

        int lengthInt = length;
        byte[] answer = new byte[lengthInt + 5];
        answer[0] = sof;
        answer[1] = length;
        for (int i = 2; i < (answer.Length); i++)
        {
            answer[i] = (byte)serialPort1.ReadByte();
        }

        return answer;
    }
    catch (Exception e)
    {
        // error handling
        return new byte[] { 1 };
    }
    // box.Text = box.Text + System.BitConverter.ToString(answer) + "\r\n";
}

public void zb_write_configuration(byte configID, byte len, byte[] value)
{
    byte length = (byte)(2 + value.Length);
    byte cmd0 = 0x26;
    byte cmd1 = 0x05;
    // data ist jetzt ConfigID Len Value

    byte fcs;
    fcs = (byte)(length ^ cmd0 ^ cmd1 ^ configID ^ len);
    for (int i = 0; i < value.Length; i++)
    {
        fcs ^= value[i];
    }

    byte[] buffer = new byte[7 + value.Length];
    buffer[0] = sof;
    buffer[1] = length;
    buffer[2] = cmd0;
    buffer[3] = cmd1;
    buffer[4] = configID;
    buffer[5] = len;

    value.CopyTo(buffer, 6);
    buffer[buffer.Length - 1] = fcs;
    // serialPort1.DiscardInBuffer();
    this.schreibeBuffer(buffer, 0, buffer.Length, false);
    // serialPort1.Write(buffer, 0, buffer.Length);
    // box.Text = System.BitConverter.ToString(buffer) + "\r\n";
    // debuginfo

    this.readResult(6);
    // end debuginfo
}

// just writes the given buffer and flushes the inbuffer first
private void schreibeBuffer(byte[] buffer, int start, int stop, bool flushInBuffer)
{
    if (flushInBuffer) serialPort1.DiscardInBuffer();

    try
    {
        serialPort1.Write(buffer, start, stop);
    }
    catch (Exception e)
    {
        MessageBox.Show("You must first choose an unused COM Port.", "Serial_Interface_Error",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }
}
}
```

```

// writes result to textbox
private void readResult(int answerLength)
{
    try
    {
        byte[] answer = new byte[answerLength];
        for (int i = 0; i < answerLength; i++)
        {
            answer[i] = (byte)serialPort1.ReadByte();
        }
        box.Text = box.Text + System.BitConverter.ToString(answer) + "\r\n";
    }
    catch (Exception excec)
    {
        // errorhandling
    }
}

public void zb_get_device_info()
{
    byte sof = 0xFE;
    byte length = 0x01;
    byte cmd0 = 0x26;
    byte cmd1 = 0x06;
    byte data = 0x02;
    byte fcs;
    fcs = (byte)(length ^ cmd0 ^ cmd1 ^ data);
    byte[] buffer = { sof, length, cmd0, cmd1, data, fcs };
    serialPort1.DiscardInBuffer();
    serialPort1.Write(buffer, 0, 6);
    // this.readResult(14);
}

public void zb_app_register_request(bool isCoordinator)
{
    byte[] buffer;
    if (isCoordinator)
    {
        byte length = 0x0D;
        byte cmd0 = 0x26;
        byte cmd1 = 0x0A;
        // begin data
        //byte appEndPoint = 0x01;
        //byte appProfileId0 = 0x01;
        //byte appProfileId1 = 0x01;
        byte[] data = { 0x01, 0x04, 0x01, 0x00, 0x01, 0x01, 0x00, 0x01, 0x01, 0x00, 0
            x01, 0x00, 0x01 };
        byte fcs = 0x25;
        buffer = new byte[] { sof, length, cmd0, cmd1, 0x01, 0x04, 0x01, 0x00, 0x01, 0
            x01, 0x00, 0x01, 0x00, 0x01, 0x01, 0x00, fcs };
        //      00    01    FE    0D    26    0A    01    04    01    00    01    01
    }
    else
    {
        byte length = 0x0D;
        byte cmd0 = 0x26;
        byte cmd1 = 0x0A;
        // begin data
        //byte appEndPoint = 0x01;
        //byte appProfileId0 = 0x01;
        //byte appProfileId1 = 0x01;
        byte[] data = { 0x01, 0x04, 0x01, 0x00, 0x01, 0x01, 0x00, 0x01, 0x01, 0x00, 0
            x01, 0x00, 0x01 };
        byte fcs = 0x27;
        buffer = new byte[] { sof, length, cmd0, cmd1, 0x01, 0x04, 0x01, 0x03, 0x01, 0
            x01, 0x00, 0x01, 0x00, 0x01, 0x01, 0x00, fcs };
        //      00    01    FE    0D    26    0A    01    04    01    03    01    01
    }
    this.schreibeBuffer(buffer, 0, buffer.Length, true);
    this.readResult(6);
}

public void zb_start_request()
{

```

B Listings

```
byte length = 0x00;
byte cmd0 = 0x26;
byte cmd1 = 0x00;
byte fcs = (byte)(length ^ cmd0 ^ cmd1);

byte[] buffer = { sof, length, cmd0, cmd1, fcs };

this.schreibeBuffer(buffer, 0, 5, false);
// this.readResult(5);
// this.readResult(6);
}

#region daten senden
// send test data to the given short address only one byte
public void zb_send_data_request(byte[] shortAddress, byte message)
{
    byte length = 0x09;
    byte cmd0 = 0x26;
    byte cmd1 = 0x03;
    // comid hadl ack rad
    // len
    byte[] data = { shortAddress[0], shortAddress[1], 0x01, 0x01, 0x01, 0xFF, 0x05, 0
        x01, message };
    byte fcs;

    fcs = (byte)(length ^ cmd0 ^ cmd1);
    for (int i = 0; i < data.Length; i++)
    {
        fcs ^= data[i];
    }
    byte[] buffer = { sof, length, cmd0, cmd1, data[0], data[1], data[2], data[3], data
        [4], data[5], data[6], data[7], data[8], fcs };

    this.schreibeBuffer(buffer, 0, buffer.Length, false);
    // this.readResult(5);
}

// hier tu ich ueberladen, damit ich mehrere bytes schicken kann
public void zb_send_data_request(byte[] shortAddress, byte[] message)
{
    byte length = (byte)(8 + message.Length);
    byte cmd0 = 0x26;
    byte cmd1 = 0x03;
    byte len = (byte)(message.Length);
    // box.Text = length.ToString("X");
    byte[] data = new byte[8 + message.Length];
    byte[] data1 = { shortAddress[0], shortAddress[1], 0x01, 0x01, 0x01, 0xFF, 0x05,
        len };
    byte fcs;

    data1.CopyTo(data, 0);
    message.CopyTo(data, 8);

    fcs = (byte)(length ^ cmd0 ^ cmd1);
    for (int i = 0; i < data.Length; i++)
    {
        fcs ^= data[i];
    }
    byte[] buffer = new byte[4 + data.Length + 1];
    buffer[0] = sof;
    buffer[1] = length;
    buffer[2] = cmd0;
    buffer[3] = cmd1;

    data.CopyTo(buffer, 4);
    buffer[buffer.Length - 1] = fcs;
    //byte[] buffer = { sof, length, cmd0, cmd1, data[0], data[1], data[2], data[3],
        data[4], data[5], data[6], data[7], data[8], fcs };

    this.schreibeBuffer(buffer, 0, buffer.Length, false);
    // this.readResult(5);
}
#endregion

public void test_rf(byte mode, byte frq, byte tx)
```

```

{
    byte length = 0x04;
    byte cmd0 = 0x41;
    byte cmd1 = 0x40;
    //byte mode = 0x00;
    //byte frq = 0x00;
    //byte tx = 0xFF;
    byte fcs = (byte)(length ^ cmd0 ^ cmd1 ^ mode ^ frq ^ tx);
    byte[] buffer = new byte[] { sof, length, cmd0, cmd1, mode, frq, tx, fcs };
    this.schreibeBuffer(buffer, 0, 6, false);
}
// enables joinig for everybody ?
public void zb_permit_joining_request(bool allow)
{
    if (allow)
    {
        byte length = 0x03;
        byte cmd0 = 0x26;
        byte cmd1 = 0x08;
        byte dest0 = 0xFC;
        byte dest1 = 0xFF;
        byte timeout = 0xFF; // means indefinitely
        byte fcs = (byte)(length ^ cmd0 ^ cmd1 ^ dest0 ^ dest1 ^ timeout);

        byte[] buffer = new byte[] { sof, length, cmd0, cmd1, dest0, dest1, timeout,
            fcs };
        this.schreibeBuffer(buffer, 0, buffer.Length, false);
    }
    else
    {
        byte length = 0x03;
        byte cmd0 = 0x26;
        byte cmd1 = 0x08;
        byte dest0 = 0xFF;
        byte dest1 = 0xFF;
        byte timeout = 0x00; // means indefinitely
        byte fcs = (byte)(length ^ cmd0 ^ cmd1 ^ dest0 ^ dest1 ^ timeout);

        byte[] buffer = new byte[] { sof, length, cmd0, cmd1, dest0, dest1, timeout,
            fcs };
        this.schreibeBuffer(buffer, 0, buffer.Length, true);
    }
}
}
}

```