Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

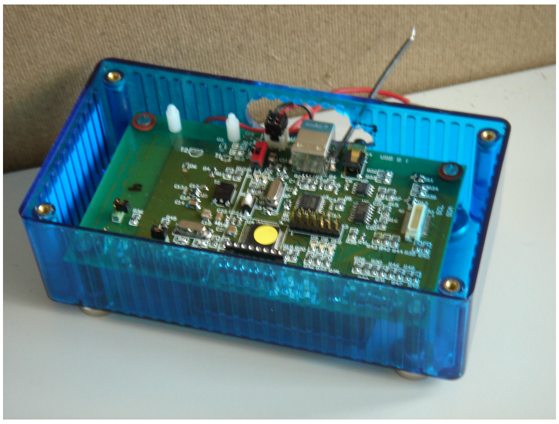
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

SIEMENS

MASTER THESIS

for the degree of
Master of Science ETH in Electrical Engineering and Information Technology

Ethernet-based Deployment Support Network



September 2007 until March 2008

Author:
Florian Betschart

Tutors:
Dr. Simon Künzli
Kevin Martin
Dr. Jan Beutel

Supervisor:
Prof. Dr. Lothar Thiele

Abstract

Embedded in the area of wireless sensor networks an Ethernet-based Deployment Support Network (DSN) has been developed within the work presented in this thesis. This system allows us to monitor the sensor network nodes by inspecting captured status messages. In addition, the user can reprogram and configure the sensor network e.g. reset the sensor nodes using a remote procedure call protocol. Wireless sensor networks are hardly observable due to the distributed nature of the system as well as limited energy recources, memory and computing power. As a previous system approach based on Bluetooth technology did not provide the reliabiliy and data throughput as intended, a wired architecture should deal with above limitations. After having evaluated different design approaches finally a hybrid system consisting of USB-interfaced DSN nodes and an Ethernet platform has been built.

The work included DSN node hardware design to connect the target nodes, implementation of the DSN node's firmware and development of the required gateway software running on the Ethernet platform. The user can access the DSN over the network in the backbone.

The evaluation of the system showed that the Ethernet-based DSN achieves good reliability concerning the implemented features. The DSN nodes support high data throughput and configuring of the sensor nodes is possible within a few seconds.

Zusammenfassung

Drahtlose Sensornetze erfordern aufgrund ihrer Systemkomplexität während der Entwicklung, der Testphase wie auch während der Wartung ein hohes Mass an Beobachtbarkeit. Begrenzte Ressourcen wie Energie, Speicher und Rechnerkapazität verlangen nach einer Lösung, welche möglichst wenig zusätzliche Hard- und Software der Netzwerk Knoten selbst dafür verwendet.
In dieser Masterarbeit entstand während 26 Wochen ein neues so genanntes Deployment Support Network (DSN), welches dem Benutzer erlaubt das eigentliche Sensornetz zu beobachten und zu konfigurieren. Das neue DSN soll ein bereits bestehendes System ablösen, welches über Bluetooth kommuniziert. Um gegenüber dem Bluetooth-Netz Zuverlässigkeit und Datendurchsatz zu erhöhen und auch grössere Netze (64 oder 128 Knoten) bedienen zu können, nutzt das neue System Ethernet und USB und stellt im Wesentlichen drei Dienste zur Verfügung:

- Software upload zu den DSN Knoten als Zwischenspeicher und automatische Programmierung der Sensor Knoten.
- Empfangen von Statusmeldungen, so genannte log messages, von den Sensor Knoten.
- Kontrolle des Netzes durch Senden von Befehlen an DSN- und Sensor Knoten unter Verwendung eines Remote Procedure Call Protokolls.

Ein zentraler DSN Server bildet die Schnittstelle zum DSN über Ethernet. Auf den Server können Benutzer mit Client Tools zugreifen um beispielsweise die Knoten mit neuer Software zu bestücken. Die Arbeit bestand darin, eine Evaluierung von möglichen Systemarchitekturen durchzuführen und danach die gewählte Variante umzusetzen inklusive Hard- und Software Entwicklung der DSN Knoten sowie DSN spezifischer Software Implementation für einen bestehenden Ethernet zu USB Gateway. Abschliessende Tests mit acht DSN- bzw. Sensor Knoten haben gezeigt, dass die DSN Knoten eine hohe Performance aufweisen und zuverlässig funktionieren.

Preface

With the present master thesis I finish my graduate study at the Department of Information Technology and Electrical Engineering (D-ITET) at the Swiss Federal Institute of Technology (ETH) in Zurich. First, I would like to thank to Prof. Dr. Lothar Thiele and Dr. Jan Beutel from the Computer Engineering and Networks Laboratory (TIK) for supporting and attending my work. I would like to thank to Siemens Building Technologies (SBT) to make it possible to do my thesis in the interesting field of fire detection systems in the Communication & Wireless Group (CWL Group) as part of R&D department in Zug.

I would like to thank to my advisor Dr. Simon Künzli, Senior Engineer at SBT, for his guidance and helpful feedback during the project. Furthermore, I would like to address my sincere thanks to my co-advisor Kevin Martin and Severin Hafner, both R&D Engineers at SBT, who helped me especially with the DSN tools. Thanks also to Manuel Lussi, Apprentice Electronics at SBT, who supported the soldering of the DSN adapterboard prototypes. My gratitudes also go to the CWL Group, namely Alex, Erich, Urs, Pascal, Christian, Matthias B., Philipp and Matthias N. for their helpfulness and the kindly atmosphere during the work.

In addition, special thanks to Chris Liechti, Software Developer at SBT, for the inspiring discussions related to my work or technical stuff in general.

Many thanks and love to Carola who attended me on my way the last four and half a year during my academic education.

Zug, March 2008

Florian Betschart

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Wireless Sensor Networks

This thesis is embedded in the area of *Wireless Sensor Networks (WSNs)*. A WSN consists of distributed sensor nodes that run autonomously. Such devices normally are equipped with a microprocessor, a power module, a radio chip and antenna for communication. This system's objective is to monitor the environment and gather information in various application areas such as sports, human health care, building technologies or vehicle systems. Optical, thermal, acoustic, or acceleration sensor units are therefore integrated. WSNs can for example be installed as alarm transmitter in fire detection systems or earthquake warning systems as described in [1]. There are often different types of WSN nodes in a system. One type records data periodically like air temperature for example. Another one acts as data storage unit or as data gateway to a wired backbone network. With wireless communication, sensor nodes can be placed and deliver data from impassable areas, which is a big advantage compared to wired systems. Also radio reprogramming is possible without touching the system at all.

1.2 WSN observability limitions

Devices in a WSN are highly energy- and resource-constrained due to the distributed nature, which makes the design and implementation a rather difficult task. The development requires a detailed adaptation of the devices to the operation environment. The nodes are built for a special purpose and the required hard- and software is therefore highly optimised because of limited processor memory, speed and energy. A network observing glacier movement is typically different from one used to monitor human health state as described in [3] for example. The architecture named *CodeBlue* provides wireless monitoring of patients and alerting first responders. Those devices have to allocate more computing power and

should be small, light and unobtrusive to wear, whereas the first example with the glacier monitoring would probably care less about equipment size or velocity of data transmission. On the other hand, this type is optimised to resist the harsh environmental conditions.
How and where to proceed captured data? Should the device send the data to a gateway, which needs a lot of battery power, or do some first analysis itself? What kind of media acces control protocol shall be taken? Does the device need any ciphering for data communication and how to realise it with low computing power? These are questions, WSN-Engineers have to deal with. Naturally, verification during development, testing, commissioning and maintenance of a WSN described are fundamental tasks. Due to above described boundaries, observability is very limited but highly needed in WSNs. Therefore, a system to monitor a WSN has to take care on these limitations.

1.3 Deployment of WSNs

1.3.1 Debug code running on target nodes

There are several ways to observe a WSN's behaviour. Blinking LEDs[1] or message outputs to indicate the device state lead to a solution with additional hardware or debug-software. Since all this more or less influences the system actions by instrumenting the devices memory and wasting energy, this is not satisfying.

1.3.2 Bluetooth-based Deployment Support Network

To overcome the problem described above, the Swiss Federal Institute of Technology Zurich (*ETH Zurich*) has developed a so called *Deployment Support Network (DSN)* [4]. This second autonomous multi-hop network is sublayered to the WSN using *Bluetooth Nodes (BTnodes)* for communication and gathering data from the WSN nodes. More information about Bluetooth can be found in [5] and the datasheet of the BTnodes is available from [6]. The basic idea is to monitor the WSN without using the same radio frequency range. No interfering would ever disturb the WSN nodes message traffic. Each node of the WSN therefore is connected physically with a DSN device. Power for the DSN nodes is usually provided over cables or from batteries alternatively. Programming of the WSN nodes over the DSN is possible.

[1] Light Emitting Diodes

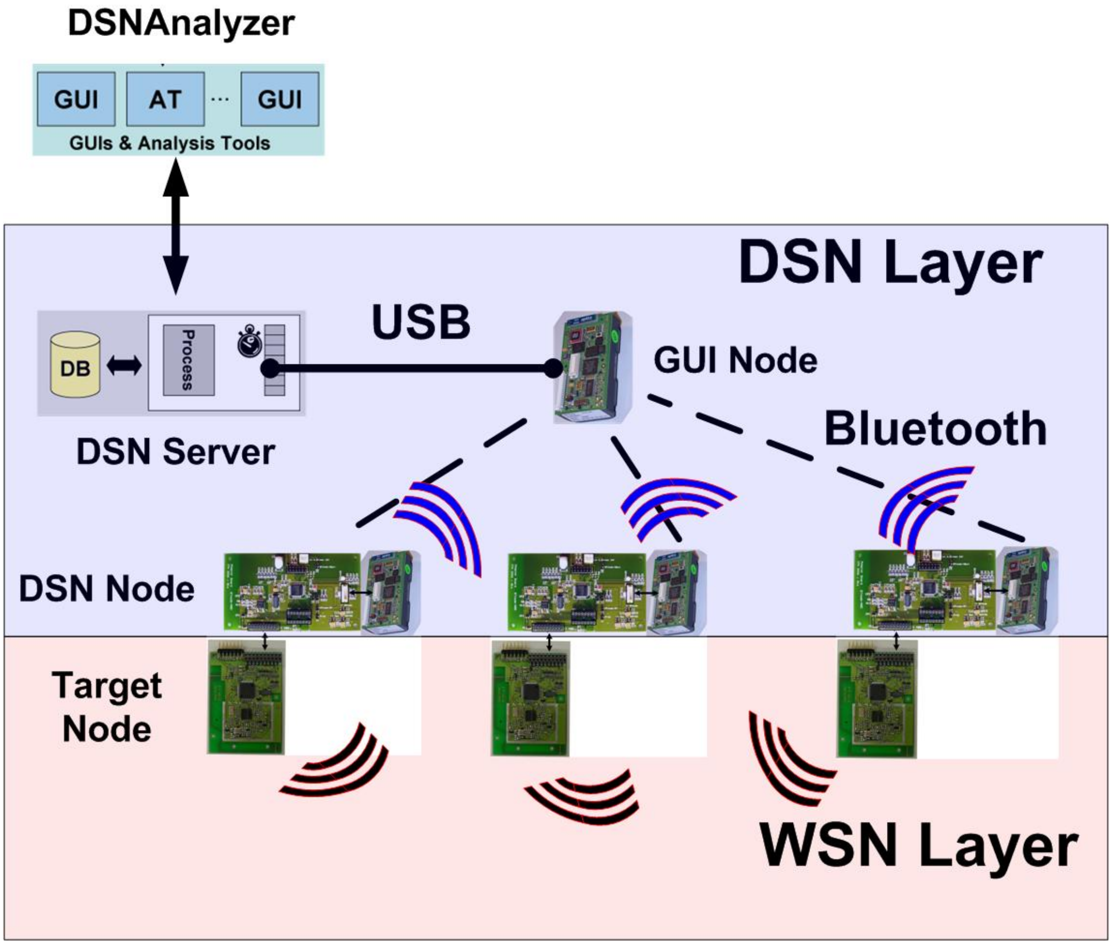


Figure 1.1: The target nodes use radio communication to build a wireless network monitoring the environment. Over an adapterboard, they transfer log messages to and receive commands from the DSN nodes, here BTnodes. The BTnodes communicate with Bluetooth over the GUI node to the DSN Server. The DSN is a sublayered wireless network. The DSNAnalyzer is a client GUI-tool to reconstruct the WSN and configure the target nodes.

1.3.2.1 DSN Server

The central element of the DSN is the *DSN Server* described in [7]. It runs on a public host and gives a client the possibility to interact with the DSN. In detail, it provides to control the DSN and the WSN attached by sending out commands to the BTnodes and target nodes. In addition, it allows to get log messages from the target nodes, as mentioned above. All messages from the target nodes are stored in a SQL-database. This database can then be read by different tools like e.g. the *DSNAnalyzer* developed in a master thesis [8]. The DSN Server provides remote programming of the target nodes over the BTnode-based DSN.

1.3.2.2 GUI node

The so called *GUI node* is also a BTnode device and builds the connection between DSN Server and DSN. The GUI node communicates with the DSN nodes over Bluetooth and is on the other side plugged over a serial connection to the host where the DSN Server runs. All log messages and command traffic as handled over the GUI node.

1.3.2.3 DSNAnalyzer as client

The DSNAnalyzer is a backend software tool that relies on the DSN Server. It has a graphical user interface and can be used to monitor the target node network, to send commands to it and to create statistics by analysing log message data. The WSN network can be stimulated and the target network can be reprogrammed over the DSN Server. Tests and measurements like range of DSN nodes and RSSI[2]-values of the nodes can be set up. An overview of the existing system used at SBT consisting of target node network, DSN, DSN Server and DSNAnalyzer can be seen in Figure 1.1.

1.3.3 WSNSpy

Another approach to observe a WSN is the *WSNSpy* developed by Hafner in [9]. In this project, a client tool was developed to passively overhear sensor network traffic by setting up a distributed spy network. Nodes of the same hardware as the target nodes run a spy application programmed in [9] to simulate the sensor network from the gathered message transmissions of the target nodes. The work consists of the spy network and a client tool for user interface. The client tool provides a GUI in order to show graphically the state of the target network and to do some analysis.

The system does neither make use of the sensor nodes themselves nor generates further network traffic in the frequency band of the target nodes. The WSNSpy only captures low-level information from the overheard messages. Test results described in [9] show, that the WSNSpy is able to overhear almost 100 % of the WSN network traffic. As a drawback compared to the DSN, it does neither support sending commands to the target nodes nor reprogramming.

[2] Received Signal Strength Indication (RSSI) is a measurement of the power that a received radio signal contains.

1.4 Problem statement and scope of thesis

Despite the users are very familiar with the BTnode-based DSN and its easy use, performance and reliability problems have occured with an increasing number of nodes using Bluetooth as communication technology and a single serial channel from the GUI node to the backend server. In addition, Bluetooth connections seem not to be stable over a long time. This is a real bottleneck, since all DSN data traffic is passing there. In parallel to the WSNspy, which is currently in use, a new system was desired to monitor at least 64 nodes. This master thesis scope was to design a new Deployment Support Network based on a wired concept in order to get rid of the limitations described above. It should be possible to send network-specific commands to the target and the DSN nodes, to upload new software to the target nodes, and to observe the target network by forwarding log messages from the target nodes to the DSN Server. Inferfaces to the work described here are the target nodes on the one side and the DSN Server on the other.
Therefore, a new adapterboard to connect the target node to some wired network was to develop. A design decision had to be taken which technology, either Ethernet or USB or both to use to transport data from the adapterboards to the DSN Server. We also addressed design details such as the boards power supply or the partitioning of software to several processors of the DSN devices. To still work with the DSN Server later on, we had to define an interface to our wired DSN. Except the ”hard work”, this master thesis basic outline also included the study and analysis of related work in this field and an in-deep evaluation at the end.

1.5 Chapter overview

The thesis is structured as follows: Chapter 2 describes related work. The actual most important WSN testbeds from the candidates point of view are discussed. The third chapter shows and explains the conceptual design of the different system components. In addition, design alternatives are put in relation to the chosen approach. A detailed documentation of the Hardware development as well as the software implementation is given in Chapter 4, 5 and 6 respectively. In Chapter 7 the evaluation and testing of an 8-node system is presented whereas Chapter 8 summarises and concludes the thesis and gives an outlook to further work.

Chapter 2

Related work

In this chapter, we give an overview of various services and testbeds for the development, debugging and monitoring of WSNs. The presented systems have a similar scope, but are not related to work performed at ETH or Siemens. We describe shortly each testbed's basic features and focus subsequently on the similarities, but also underline the differences to the DSN developed within this thesis.

2.1 TWIST

One of the latest presented testbed architectures for the observation and control of a WSN is TWIST developed by Wolisz et al. [10]. TWIST is a scalable and hierarchical testbed architecture for the deployment of WSNs, see Figure 2.1. It allows the user to configure the sensor nodes, provides network-wide (re)programming and supports several sensor node platforms. From the infrastructure's point of view, it is similar to our Ethernet-based DSN because it also bases both on Ethernet and USB communication and makes use of standardised hardware like hubs as well as open-source software. A nice feature of TWIST is the capability to control the power of the sensor nodes remotely. In the backbone a server is running to store received debug data or sensor measurement results. Several users can access the server for data exchange over the web. Network topologies are dynamically selectable and some testbed nodes can even be integrated as part of the sensor network application. In default configuration, TWIST is mainly for observation of sensor nets in indoor environments. Experiments show, that over one hundred sensor nodes can be serviced with TWIST. Despite the similarities, it differs from our approach:

- Our DSN has to provide a hardware interface to the already existing target nodes (Siemens A80) which actually neither have an USB or Ethernet connector nor the corresponding software stack. So an adapterboard is needed

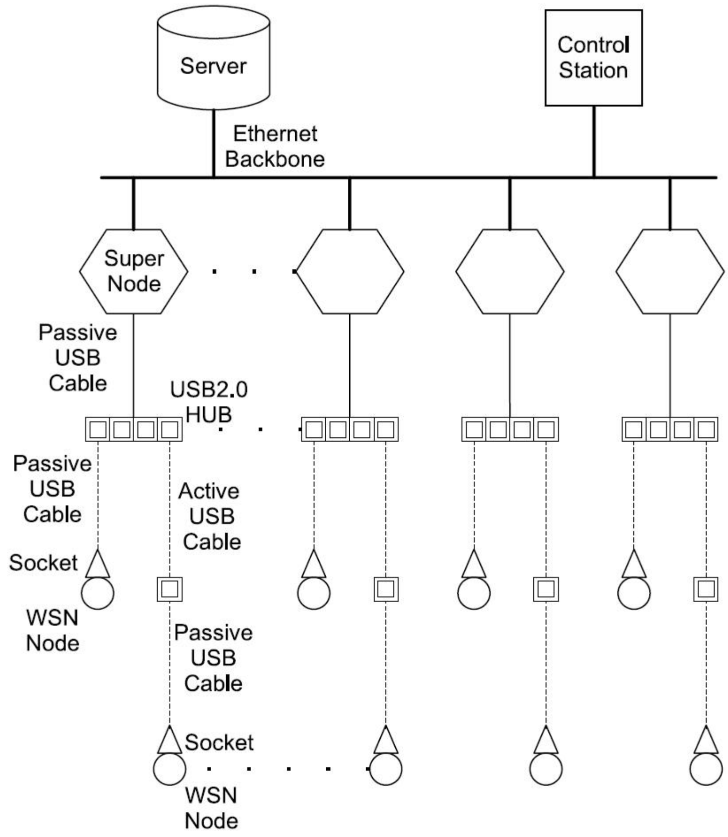


Figure 2.1: The TWIST architecture bases on standardised hardware. The WSN nodes (Telos motes) [11] are attached over USB cables and hubs to NSLU2 devices (Network Storage Link for USB 2.0 by Linksys) [18] acting as so called Supernodes. Running a modified Linux OS version, they use TCP/IP for communication to a server. The Supernodes are responsible for WSN node administration, status message capturing and code distribution. In the backbone, a web-based control station is the interface between user and testbed. Figure taken from [10].

to interface the A80.

- As the A80 does not support direct programming, we use an EEPROM on the adapterboard to store the application code image for the A80.
- The A80 uses the I^2C protocol to communicate to the DSN node, so an extra software had to be implemented to convert I^2C to USB.
- We support up to eight DSN and target nodes per one NSLU2 whereas TWIST supports two.
- Our adapterboard supports to plug an additional sensor device like the BTnode also for environment monitoring if desired.

- We provide a extension connector on our adapterboard for debugging reasons and I^2C communication bus access over a stereo jack.

2.2 Motelab

Motelab presented by Werner-Allen et al. in [13] provides a public permanent installed testbed for development and testing of wireless sensor network applications over a web interface. It is mainly a composition of software tools for controlling Ethernet-connected sensor network nodes and provides an easy access for debugging and collecting data. Tmote Sky devices [12] are used as target nodes. Several users may login over the Internet to a central server to download sensor data or upload executables for the testbed. The server handles the data logging, scheduling and reprogamming of the sensor nodes. Concerning the hardware, it is important to highlight that each sensor node is connected to Ethernet. Similar to our DSN, a piece of adapter hardware is integrated in between. A second cable is used for mains power supply. Both Motelab and our DSN use Ethernet and a web-based infrastructure in the backbone. An infrastructural disadvantage of Motelab is that each node makes use of two wired connections, one for Ethernet and one for power.

2.3 Tutornet

A tiered sensor network testbed represents Tutornet [14] currently running in Ronald Tutor Hall at University of Southern California, USA. The whole network consists of several clusters and allows testbed programming as well as collecting data from the sensor nodes. A cluster consists of a stargate and attached sensor nodes via USB. A stargate is a composition of a wireless network adapter connected to a USB hub. The stargates communicate via WLAN to a central testbed server that acts as top tier. Multi-hop routing between the central PC and stargate or among stargates is possible. As in Motelab, different users can access the testbed server over the Internet.

2.4 sMote

At UC Berkley, USA, various wireless network testbeds are in use at the moment such as Smote, Omega or Trio [15]. The latest is an outdoor testbed whereas the others are deployed within buildings. These nets are installed for development and test of WSN applications. It facilitates research on e.g. routing protocols or system architecture experiments.

We focus here on sMote, because it has an interesting feature related to this work. The nodes are supplied from Power Over Ethernet *(PoE)* [21] and not from batteries or mains power. The testbed also supports data capturing, debugging and reprogramming over a web interface. sMote consisted (it was replaced later by Motescope [15]) of 78 Mica2Dot wireless microsensor motes [16]. This example shows, that PoE is employed in scientific environments and would have been a feasible alternative for our DSN adapterboard's powering concept.

2.5 Summary

The presented examples are similar to our approach concerning the network architecture. They all make use of wired technology as either Ethernet, USB or both to connect the target nodes. Often the target nodes themselves have a USB interface so that they can be plugged directly to a network. However, the most important difference of all of them to our DSN is the sublayerd wired network with the adapterboards we use. Despite for example Motelab also uses something like an adapterboard to connect to Ethernet, no message administration or processing is done on that platform in contrast to our adapterboard software. We therefore need less communication software overhead in the target node application itself, as for example a TCP/IP stack.
Our architecture supports the use of both BTnode-based DSN and the developed Ethernet-based DSN. Observing and controlling the WSN using two independent technologies (wired communication and Bluetooth) could even improve the overall performance and reliability. For example to use the BTnodes where a wired connection is hardly possible to deploy would be a good alternative.

Chapter 3

Conceptual Design

In this chapter we document the conceptual design of the Ethernet-based DSN. We list the requirements and guidelines by SBT and clarify the interfaces to existing work. Next, we present possible alternatives to the proposed approach that we discussed at the beginning of the design phase. Afterwards, we describe the design of each system component that was developed within this work.

3.1 Requirements and guidelines

We already introduced the scope of the thesis in Chapter 1. Here we go into more details. Considering the thesis' proposal by SBT, this project's main focus was to design and to develop a new DSN based on Ethernet technology in order to deal with the reliability and throughput limitations of the BTnode-based DSN for increased network size issues. SBT wants to use a wired network to monitor a network of at least 64 sensor nodes. At the moment, the sensor node type is a Siemens A80, as already mentioned. Later on, a new radio module should be supported. The Ethernet-based DSN should guarantee the reuse of the software infrastructure for analysis of captured data, namely DSN Server and DSNAnalyzer respectively. Basically, the new system has to provide the following functionalities:

- **Data and event message logging**
  The core service of the DSN is the data and event message logging that allows the user to supervise the state of the target nodes e.g. if a connection to a WSN gateway was found. So far, the target nodes send out log messages to a BTnode connected over an I^2C bus [17]. The new system has to provide a possibility to capture these messages, too, but to transfer them over the Ethernet to the DSN Server.

- **Command service**
  The Ethernet-based DSN has to support the same functionality as the BTnode-

based to send such Remote Procedure Calls (RPCs) to configure the DSN or to request status information from the target node e.g. stored code version number and type, WSN connectivity information or a list of reachable nodes.

- **Remote programming of target**
  The BTnode-based DSN provides a remote programming service that allows to send a code image to the DSN node and storing it in a special memory. Our system needs to distribute the code over the network to the nodes.

- **Powering of both DSN node and target node**
  The A80 radio module has to be supplied from the DSN node. We desire just to use one cable for both power and communication to the DSN node, if possible.

- **64-node test system**
  Whatever the architecture looks like, the Ethernet-based DSN should allow its use with at least 64 distributed nodes.

- **Employment of BTnodes**
  Not strongly required but nice to have is the possibility to reuse the BTnode-based DSN with the new hardware. We will use this feature for our benchmarking of log message loss versus the wired system.

To sum up, the Ethernet-based DSN has to provide features similar to the BTnode-based DSN, but using wired technology. From this point of view it is pretty clear that we lack of proper adapterboards as DSN nodes to build a network to interface the DSN Server over Ethernet on the one side and the target nodes over an onboard I^2C bus on the other.

3.2 Projects motivation - or why not to buy it?

In this section we present the reasons that motivate this work and why we can not buy such an Ethernet-based DSN that fits our application:

1. If we consider the related work presented in chapter 2 we have to admit that no testbed for WSNs exists that could directly interface our application specific target nodes (A80), neither from the hardware's nor the software's point of view.

2. With the BTnode-based DSN, we used two devices to build one DSN node, namely adapterboard and BTnode. Now we only need one adapterboard.

3. The system should have a reasonable price especially for a 64-node network.

4. Already existing know-how and infrastructure is strongly recommended to be reused with the new system. For example the BTnode equipment has to be compatible with the new adapterboards and we want to employ the DSN Server and DSNAnalyzer.

To conclude the requirements section we can state that the existing BTnode-based DSN is not able to monitor and control the WSN in a convenient manner when the network size is increased up to about 64 nodes and **no suitable testbed exists that can do it for us so far.**

3.3 Proposal from SBT

As a first suggestion to build a wired DSN, SBT proposed to develop a proper Ethernet-adapterboard as DSN node to connect the target node directly to the network. From the hardware side, this adapterboard would basically consist of a microcontroller, an EEPROM to store target node software code, an Ethernet controller and connector, I^2C bus [17] and BTnode connector. In order to fulfill the basic requirements described above, the software had to include a TCP/IP stack to allow Ethernet communication, a webserver application and EEPROM programming procedures as well as an I^2C-communication stack. Figure 3.1 shows a block diagram. All together, those adapterboards would build the DSN interfacing both the target nodes and the DSN server over Ethernet switches in the backbone. The DSN node is supplied from Power over Ethernet *PoE* so that just one cable has to be plugged.

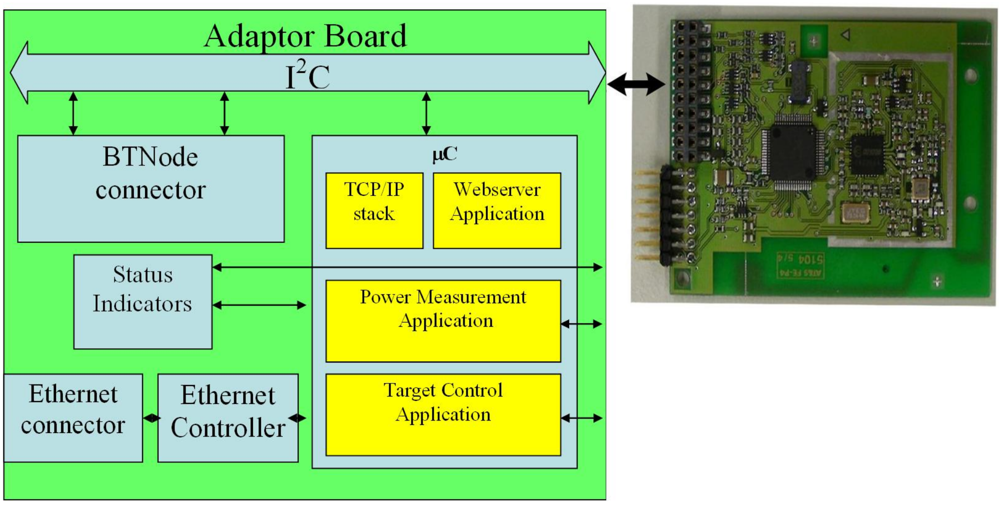


Figure 3.1: Block diagram of the proposed adapterboard as DSN node with a Siemens A80 radio module connected.

3.4 Fundamental DSN architecture options

As a starting point for the conceptual design, we took the proposal from SBT summarised aboved and added in the Appendix. We analysed PoE devices and concepts, availability and prices, the use of different microcontroller types and the existing know-how. Discussions with engineers from the Wireless and Communication Group of SBT yielded to seven architectural design options for our Ethernet-based DSN. The outcome is presented as follows:

- **A: Proposed approach and buying PoE switches**
  This solution fits the proposal and makes use of commercial so called PoE injectors at the network's root side and splitters close to the adapterboards. Two separate lines connect the DSN node to the splitter, one for data and one for power. On the adapterboard, a Texas Instruments *TI* MSP430 microcontroller is employed, see Chapter 4 for device information.

- **B: Proposed approach and buying PoE PCB module**
  Similar to the approach before, here we would buy a PoE Printed Circuit Board *PCB* module from TI and integrate it to the adapterboards PCB design. One Ethernet cable is needed for both power and data. A PoE injector is necessary at the network's root side. As before, we would solder a TI MSP430 controller on the adapterboard.

- **C: Proposed approach with a self-designed PoE circuit**
  Here, we would completely develop the PoE electric circuit on the adapterboard by our own. Nevertheless, we had to buy PoE injectors. Here as well, we would employ a TI MSP430 microcontroller.

- **D: Proposed approach with USB power**
  We would build the adapterboard with Ethernet connection but employ a separate USB cable just to power the device, not for communication. A CPU with more memory and processor speed is a further option.

- **E: Buying Ethernet gateway and developing adapterboard**
  There are commercial Ethernet to serial gateways with PoE available. We would therefore develop an piece of hardware to connect the target node with such a gateway. Only one cable is needed to the DSN node. The adapterboard software would not need a TCP/IP stack, since this is integrated in the gateway application.

- **F: Proposed approach with "high performance" CPU**
  This system requires to develop an adapterboard as described in the proposal. But since we do not rely on battery power here, we could even use a bigger

CPU as for example an ARM7 [22] processor to increase the computing power on the DSN node[1]. Every of the three PoE concepts is an option here.

- **G: Buying gateway and developing USB-adapterboard**
  We would buy a commercial platform, namely a Linksys (NSLU2) device [18], to work as Ethernet gateway using a TCP/IP Stack of an open source Operating System *OS*. Such a NSLU2 is used e.g. with TWIST described in chapter 2 as well. As DSN nodes, we would develop USB-adapterboards to connect the target nodes to the NSLU2. We had to implement software both for the adapterboard and the NSLU2. The NSLU2 is self-powered and the adapterboards are supplied from USB (over self-powered hubs).

[1] In fact, we doubt the proper functionality of the TCP/IP communication stack functionality with 2 KB RAM in the TI MSP430F169, despite several F169-platforms use this stack to run a webserver.

Design Approach Evaluation Table (October 12th, 2007):

Evaluation criteria: Solution:	Cost effort	Cost per DSN node	Existing HW and SW	Maintenance	Usability and Portability	My favorite	Performance	Risks and difficulties	Ranking
A: Self-development of Ethernet-adapterboard (MSP430), buy central or distributed injectors and splitters for PoE	+ no self-made PoE design - TCP/IP implementation / port of exsting stack (e.g. uIP), no approved operation, port likely to be difficult - onboard SW **(4)**	147 CHFr. **(4)**	+ uIP portable for MSP430 (TCP/IP stack) + schematics for Ethernet controller CS8900 available + part of old adapterboard SW can be reused **(7)**	+ MSP430 know-how, adapter-board specific **(5)**	+/- just 1 cable to DSN node next to the node 2 cables from splitter to adapterboard + additional PoE injector, but data switch needed anyway **(3)**	**(2)**	+/- relative little mem ory (116 KB Flash, 8 KB Ram), even with biggest MSP430 type + low power IC, 2 mW @ 1 MHz - application extension? **(4)**	- HW assembly - computing power MSP **(3)**	**154**
B: Self-development of Ethernet-adapterboard (MSP430), buyTexas Instruments PoE module to integrate on PCB	+ PoE just to integrate with a second PCB-module - TCP/IP implementation /port of existing stack, port likely to be difficult +/- onboard SW **(3)**	186 CHFr. **(3)**	+ uIP portable for MSP430 (TCP/IP stack) + schematics for Ethernet controller CS8900 available - delivery time (3 weeks) for PoE module + old SW of adapterboard **(6)**	+ MSP430 know-how, adapter-board specific **(4)**	+ just 1 cable to DSN node from switch needed - needs more space in a housing + no additional devices **(4)**	**(7)**	+/- relative little memory (116 KB Flash, 8 KB Ram), even with biggest MSP430 type + low power IC, 2 mW @ 1 MHz - application extension? **(2)**	- HW assembly - computing power - high price for PoE module (80 Fr.) **(2)**	**124**
C: Self-development of Ethernet-adapterboard (MSP430) and design of PoE circuit on the adapterboard	- additional PoE to design and solder - TCP/IP implementation/port of existing stack, no approved operation - correct behaviour of PoE +/- onboard SW **(2)**	109CHFr. **(7)**	+ uIP portable for MSP430 (TCP/IP stack) + schematics for Ethernet controller CS8900 available - there is reference design of PoE with a LM5070, but difficult + old SW of adapterboard **(4)**	+ MSP430 know-how, adapter-board specific **(3)**	+ just 1 cable to DSN node from switch needed + no additional devices **(7)**	**(6)**	+/- relative little memory (116 KB Flash, 8 KB Ram), even with biggest MSP430 type + low power IC, 2 mW @ 1 MHz - application extend? **(1)**	- HW assembly - PoE circuit engineering - computing power MSP **(1)**	**108**
D: Self-development of Ethernet-adapterboard (MSP430F2618), extern supply over **USB** cable	+ no PoE to develop +/- TCP/IP implementation/port of existing stack, port likely to be difficult +/- onboard SW **(5)**	115 CHFr. **(6)**	+ uIP portable for MSP430 (TCP/IP stack) + schematics for Ethernet controller CS8900 available + part of old adapterboard SW could be reused **(3)**	+ MSP430 know-how, adapter-board specific **(6)**	- 2 cables to each DSN node needed (1 for power, 1 for data) - lots of adapters and hubs **(2)**	**(4)**	+/- relative little Mem (116 KB Flash, 8 KB Ram), + low powe IC, 2 mW @ 1 MHz - application extend? **(3)**	- HW assembly - computing power MSP **(4)**	**139**
E: Buy high computing power **Ethernet interface gateway**, self-development of adapterboard to connect to radio module, PoE	+ no PoE to develop (available) + no TCP/IP implementation/port - write drivers for adapterboard **(6)**	500 CHFr. **(1)**	+ old SW/HW of adapterboard **(2)**	+ re-use possible, general purpose gateway, SW **(2)**	+ just 1 cable to DSN from switch required + PoE module is on Ethernet platform - 3 PCB's in the end, packaging? + no additional devices **(6)**	**(1)**	+ Ethernet + full TCP/IP/DHCP protocol **(7)**	- HW assembly - SW of adapterboard **(7)**	**178**
F: Self-development of Ethernet-adapterboard with **bigger uC** (e.g. ARM7), PoE	+/- PoE is integrated - a lot of know-how HW/SW of uC to acquire - TCP/IP implementation / port of uIP on ARM7 **(1)**	139 CHFr. **(5)**	+ uIP portable for ARM LPC2124 (TCP/IP stack) + schematics for Ethernet controller - old SW of adapterboard **(1)**	- uC: no know-how **(1)**	+/- just 1 cable to DSN from switch, depending on PoE solution choice - no low power (Btnodes?) + portable **(5)**	**(5)**	+ more memory (256 KB Flash, 16 KB Ram), + high computing power **(5)**	- HW assembly - SW know-how, familiarisation +/- PoE circuit **(5)**	**116**
G: Buy platform (NSLU2) as Ethernet gateway, linux, & design adapterboard to connect radio module over USB	+ no PoE to develop, supply over USB designated - USB to platform SW, I²C + no TCP/IP implementation / port - Embedded Linux know-how **(7)**	198 CHFr. **(2)**	+ no TCP/IP stack implementation nedded, all on platform - no connection SW yet + old SW of adapterboard + debian linux available **(5)**	+ well-known linux platform reusable + adapter-board **(7)**	- relative big composition of platform, new adapterboard to new radio module - no PoE possible (except hack), separate power /data to platform - for a 64-node-net, a lot of NSLU2s needed, how many? **(1)**	**(3)**	+ actually just limited by adapterboard + high computing power **(7)**	- SW connection to NSLU2 **(7)**	**202**
Criteria weight:	**7**	**2**	**6**	**3**	**4**	**1**	**8**	**5**	

3.5 Comparison and design decision

In order to determine which of the design alternatives we follow to build the new DSN nodes, we defined criteria and gave them weights. Cost effort is the most important one for us and has weight 7. The criteria are:

- cost effort with existing know-how
- estimated cost per DSN node (including injectors, splitters, hubs, PCB manufacturing, components)
- already existing HW and SW e.g. TCP/IP stack for the TI MSP430
- maintenance of the system
- usability and portability
- the candidate's favourite approach (”fun factor”)
- computing performance of the DSN node
- risks and difficulties

We compared each architecture variant from A to G by making an in-deep study of advantages and drawbacks of each according the defined criteria above. We gave points to the different design variants. The higher the number of points for a criterion the better. The **Design Approach Evaluation Table** on the previous page illustrates the results. We finally took our decision from the ranking of the design variants and from three core questions:

1. Shall we use Power over Ethernet on the DSN node or not?
2. Shall we buy a standard computing platform (NSLU2) for Ethernet connectivity and use USB to connect the adapterboard?
3. If we chose a variant from A to D or F, what microcontroller type to use (TI MSP430, ARM7)? Or are there even alternatives?

The fact that an existing TCP/IP protocol stack like *µIP* from Adam Dunkels [23] is likely to be difficult to port without any bugs on a self-developed HW platform made us decide to employ a NSLU2 box with Ethernet connectivity for our DSN. The needed TCP/IP stack is available when we us an OS, namely Linux. The power concept says that the Linksys platform is supplied from mains power and the adapterboards are powered over USB (hubs). As a result of the whole evaluation, we took design variant G. Figure 3.2 illustrates the new Ethernet-based DSN.**We focus on a modular 8-node test DNS for the work presented**

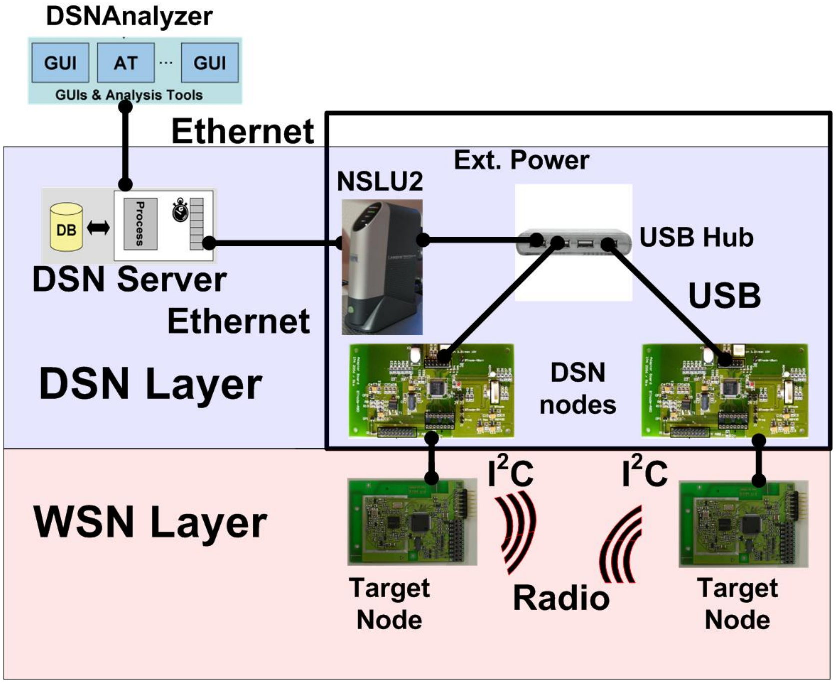


Figure 3.2: The conceptual design of the Ethernet-based DSN. The black frame defines the components to develop or to add within the work described in this thesis.

in this thesis. Later on, this various 8-node modules can be arranged together to form a 64-node network, as required by SBT. The DSN Server is redesigned in parallel to this thesis to replace the serial communication by Ethernet. To test our 8-node DSN within this work, we will implement software based on Python scripts to allow Ethernet communication for logging and sending commands. To conclude, we specify the different system parts to implement:

1. Design and development of prototypes of an USB-adapterboard as DSN node including hardware components choice, schematics, PCB layout and assembly of prototypes.

2. Adapterboard firmware to connect the target node over I^2C bus and the NSLU2 over USB.

3. Application software for the NSLU2 concerning the functional requirements of the Ethernet-based DSN to work as gateway to USB and to provide a network client interface.

4. Planning and installation of a test DSN consisting of 8 nodes.

3.5.1 NSLU2 device

Since the NSLU2, see Figure 3.3 is a core component of the Ethernet-based DSN, we mention the most important product features here. Further information can be found in the datasheet available from [18]. The NSLU2 hardware provides four status LEDs, the CPU is clocked at 266 MHz and includes 32 MB of SDRAM as well as 8 MB of Flash memory. In addition, it supports a 100 MBit Ethernet network connection. The original purpose of the NSLU2 is to attach up to two external USB-connected hard drives and to serve as network-attached storage *NAS*. However, the NSLU2 can be used for other applications. Therefore, the *NSLU2-Linux project* [20] was launched. An USB disk can be plugged to one of the USB slots and run a Debian Linux [19] version on it.
For our DSN, we do the same and use one USB-slot to plug an USB-Stick and run a Debian. We use the other USB slot to connect a hub with attached DSN nodes.

Figure 3.3: Picture of the Linksys Network Storage Link for USB 2.0 (NSLU2).

3.6 USB adapterboard hardware

In this section we present the conceptual design of the adapterboard hardware. The DSN node has to support the receiving of commands and to send out log messages from the target node. Furthermore a BTnode connector is required and a battery supply circuit, since the adapterboard is not powered over USB when we want to use the wireless BTnode-based DSN. Figure 3.4 shows a block diagram of the new DSN node.

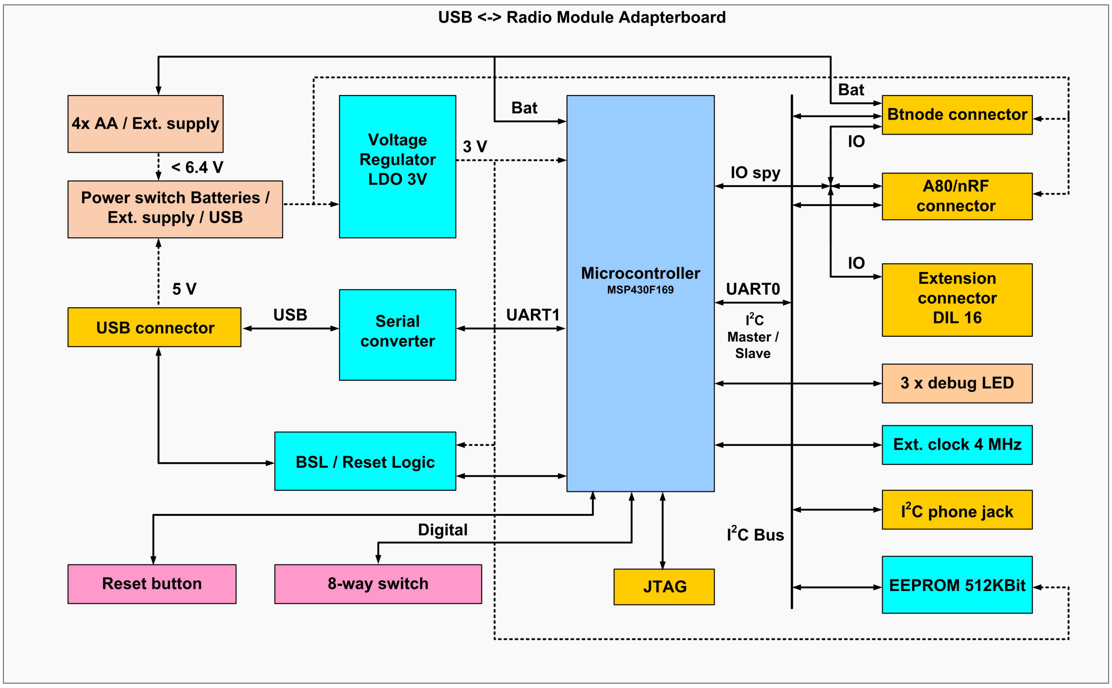


Figure 3.4: Block diagram of the adapterboard with microcontroller, target node interface, BTnode connector, extension connector and stereo jack. On the board, peripherals are connected over an I^2C bus.

3.6.1 Key features

Most parts of the adapterboard hardware do not make high technological demands and are standard implementations and we can refer to reference designs. Nevertheless, the interfaces are application specific here.
To access the adapterboard, there is the USB interface with a connector, an USB-serial converter and a microcontroller. Since the TI MSP430F169 controller is well-known to the engineers at Siemens and fits the application, we chose this type. To upload code to the controller's flash, both Bootstrap Loader *BSL* [24] and JTAG [25] are provided. We decided to operate the adapterboard at 3V, so a voltage regulator is used that converts the input voltage from either batteries or USB input to 3 V. The BTnode and the target node both are powered from USB or batteries directly. On the board, an EEPROM is accessible to store the code image for the target node. The EEPROM is connected to the controller over a central I^2C bus that also interfaces the BTnode and certainly the target node. A 16-pin extension connector provides access to several in- and output signals of the target node for debugging reasons. The adapterboard has to be capable to reset the target node to invoke its reprogramming procedure. Therefore, an general purpose output pin of the controller is connected to the A80 reset pin. Note that

the A80 reset signal is active high.

3.6.2 EEPROM

We do integrate on the adapterboard a stereo jack that interfaces the I^2C bus to connect an I^2C spy device. The EEPROM supports hardware setting of the three least significant I^2C address bits and also has a write protection input pin. Therefore, we use a 8-bit DIL switch. The four spare bits of the DIL act as general purpose inputs to the controller. To supervise the battery state, we employ a voltage divider for a measurement. The output voltage value is forwarded as input to a controller's ADC[2] input.

3.6.3 Power concept

As a nice feature, a special circuit disconnects the battery supply line if USB power is present. As we will use self-powered USB hubs in front of the NSLU2 device, we can expect a maximum of 100 mA per port for one adapterboard. When the BTnode is not plugged, we certainly use less than 100 mA to supply all the components on the adapterboard and the target node itself. If the BTnode is plugged, the batteries deliver enough current. We did not perform detailed calculations then.

3.6.4 Housing

At the end, the assembled adapterboard PCB gets a housing, namely a blue box (*Blue Box*). The Blue Box contains a battery pack and a cable to plug to a supply connector on the PCB. Fixed magnets on the bottom of the Blue Box allow to attach it to a magnetic side board for example. The Blue Boxes are available ”as is” from SBT. The components are listed in the Appendix. So far, it was clear that we had to consider the Box size for the hardware implementation.

3.7 USB adapterboard firmware

We designed the adapterboard firmware according to the requirements listed in section 3.1 to support the DSN application. In this section, we explain the conceptual design and core functionalities and do not discuss implementation specific properties like e.g. command format. In addition, we do rather specify and explain the different features of the firmware than show the exact program flow. We treat this topics in particular later in Chapter 5. A schematic overview of the adapterboard software stack is given in Figure 3.5.

[2] Analog Digital Converter

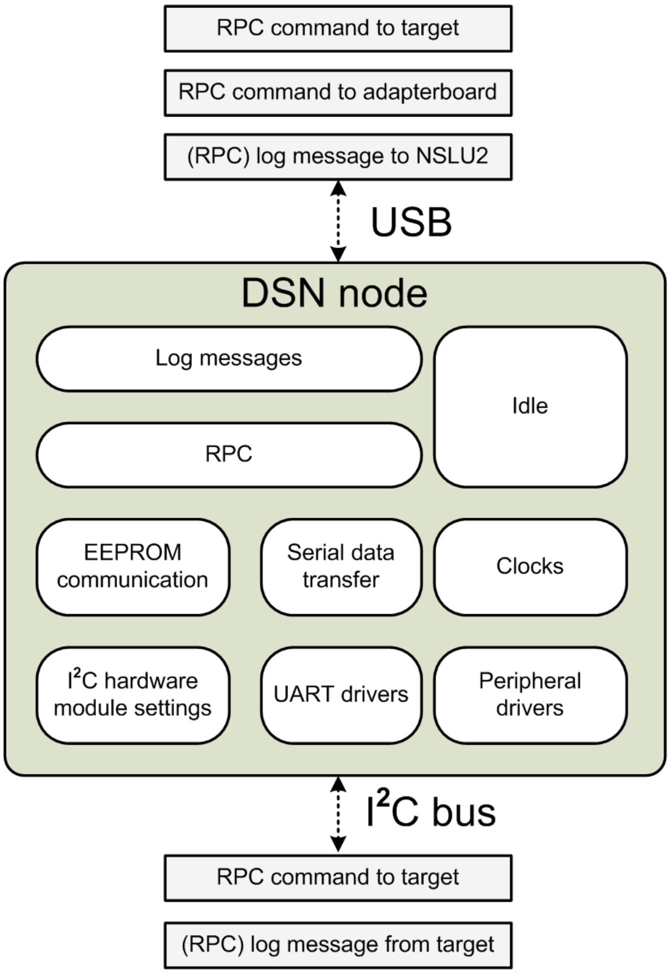


Figure 3.5: Schematic overview of the adapterboard software. Basically, the software has to handle JSON-RPC commands, program the EEPROM with a code image and display log messages from the target node over USB.

3.7.1 JSON-RPC command transfer

For command transmission to the DSN nodes we use JSON-RPC [26] because the target node already uses this mechanism. JSON-RPC is a lightweight remote procedure call protocol. Two communication peers can establish a connection and each of them is allowed to invoke a method provided by the opponent. To start a remote method, one of the peers has to send a request and receives a response back, except the request was a notification.

Any JSON-RPC command execution is invoked by the NSLU2 Ethernet gateway or the DSN Server respectively in our DSN. The DSN node receives the command string and then starts the requested procedure, e.g. reset the target

JSON-RPC command	Function
target.reset	The adapterboard has to drive a specific output pin high and low to reset the target node.
target.disable	The adapterboard has to drive a specific output pin high to hold the target node in reset.
log.disable	The target node stays under operation, but no log messages are displayed over USB.
log.enable	The adapterboard displays log messages.
write.eeprom[data]	The firmware writes a data packet to the I^2C interfaced EEPROM and awaites an ack.
forward.command[method]	A buffer is filled with the JSON-RPC command to transmit to the target node. The buffer is read and cleared when the target polls.
configure.master	The target node is hold in reset and the adapterboard is configured as I^2C master.
configure.slave	The target node is reset after the adapterboard is configured as I^2C slave.
read.bytes	Read a number of bytes from the EEPROM and display them over USB.
adapterboard.reset	Both the DSN node and target node are reset.

Table 3.1: Overview of the JSON-RPC commands the adapterboard supports.

node. Table 3.1 shows the commands that the adapterboard as DSN node has to support.
Important to know is the fact that all JSON-RPC commands have to be replied with a response from the receiver back to the sender. Therefore, the application sends back a JSON-RPC formatted string as response at the end of each executed procedure if no error occurred.

3.7.2 Command forwarding to target node

The adapterboard has to support command forwarding to the target node. Here, the request for the target is packed in the JSON-RPC command ”forward.command”. Whenever the adapterboard receives this command, it unpacks the data and writes the command to a buffer waiting for the target to poll the I^2C bus. Actually, the target polls just at specific I^2C addresses. We use the concept of double buffering. If a command buffer is full and the target has not polled yet, the next is filled and queued. **The execution of commands has the highest priority in the adapterboard’s firmware.**

3.7.3 Log message capturing and display

The default action for the adapterboard is to listen to the I^2C bus, to gather log messages from the target nodes and to write them out over USB. In order to do that, the DSN node operates as I^2C slave and has the specific I^2C 8-bit address 20 (what is clear so far, since the A80 writes to this address). We are going to use the MSP430F169 hardware I^2C peripheral interface for communication that requires application specific settings.

3.7.4 EEPROM access for programming

In order to write a code image we employ the JSON-RPC protocol as well. The reason is that we just can't store the whole image in the microcontroller's memory, so we have to transfer the data packet per packet to the controller who writes the data to the EEPROM.
In default configuration, the target node works as I^2C master and checks an I^2C-attached memory at the addresses 80 to 82 for new application code after start-up. **If new code is found, the target node reprograms itself.** The EEPROM address can be set by the DIL Switch, see section 3.6. An EEPROM programming procedure therefore requires to take different steps controlled by the NSLU2 device:

1. **I^2C master and target disable**
   First step to take when the EEPROM should be written is to configure the adapterboard as I^2C master and hold the target in reset, otherwise the adapterboard can not write anything to the EEPROM. Therefore, the adapterboard has to execute the command ”target.disable”.

2. **Code packet transmission**
   The adapterboard receives a JSON-RPC command ”eeprom.write” containing a packet of the code image as parameter, typically 64 or 128 bytes, and sends the data immediately to the EEPROM. Afterwards, the adapterboard replies with an acknoledgement to the NSLU awaiting the next packet.

3. **I^2C slave and target reset**
   When all data has been written to the EEPROM, the adapterboard is reconfigured as I^2C slave and the target is reset to trigger its self-reprogramming.

3.7.5 I^2C master and slave mode operation

Per default, the adapterboard operates as I^2C slave and the target is the master. But to write to the EEPROM, it has to become master, as described above. The firmware has to provide the possibility to change between the two modes. The

A80 software communicates with 100 kHz on the I^2C bus. If the adapterboard is master, the bus clock speed is set to 400 kHz to increase the EEPROM writing speed.

3.7.6 Battery measurement

The ADC of the controller allows to make a measurement of the actual voltage state of the batteries, if inserted. The firmware therefore provides the corresponding functions to enable and disable the measurement and to read the voltage value. The battery measurement functions are not used in the current firmware revision.

3.7.7 Watch Dog timer

To give the user the possibility to change the software mode of the adapterboard, a so called Watch Dog timer *WDT* is employed. The WDT interrupts the program flow and observes the 4-bit DIL switch input value and resets the adapterboard firmware if the input has changed.

3.7.8 Hardware utilisation and settings

A block of software is responsible to initialise the controllers hardware such as the external oscillator, pin configurations, Watch Dog and UART baudrate selection. We also have to implement the drivers for the USB (UART1) as well as the I^2C bus functionality.

3.7.9 Idle mode

When the adapterboard is not occupied with DSN services execution, the firmware falls into an idle mode.

3.7.10 Operating System

We decided not to run an OS on our DSN nodes. A suitable OS would have been the Tiny Microtheading Operating System *TinyOS* [27]. This is a free and embedded open source OS and there exists a slight port for the MSP430 controller using about half a kilobyte of memory (RAM). However, we do not use such an OS for the following reasons:

- We do not rely on a critical timing and scheduling of the tasks, that is typically handled by an OS. In our application, no more than one or two tasks are executed quasi-parallel. For example the display of a log message is interrupted by the arrival of a JSON-RPC command and its execution.

- The code overhead requiering additional memory resources is not in relation to the provided OS-functions we could employ.
- Most of the device drivers used here are standard e.g. UART1 for serial communication. So we implement that drivers by our own and do not use an OS' I/O protocol stack.

3.8 NSLU2 software

The NSLU2 software is mainly responsible for DSN and target node administration and providing a fast USB to Ethernet and vice versa data gateway. The NSLU2 allows to install a full Debian Linux OS as described above which includes the TCP/IP stack for our application. We do not have to care about memory here, since the 32 MB RAM is certainly enough for our application. Hereby, we have already answered the question about to employ an OS or not. This gives us the opportunity to use **multi-threading** as basic operation concept. Figure 3.6 presents the basic components of the NSLU2 software.

The DSN interfaces are on one side the USB attached adapterboards and on the other side an Ethernet network. To meet the requirements, we defined the following core features for the NSLU2 software to implement:

- **Scanning for DSN nodes after start-up**
  After start-up, the NSLU2 software has to scan for plugged DSN nodes. It tries to open a serial connection and creates a thread object for each. This object handles all the data transfer to and from one single DSN node.
- **Command transmission**
  To allow a DSN client (DSN Server) to send a JSON-RPC command to the DSN node or target node respectively over the network, we employ the Extensible Markup Language Remote Procedure Call *XML-RPC protocol* [28]. This protocol simply gives the user the possibility to call functions from remote applications over the Internet. The XML decribes the command decoding and HTTP [29] is the transport mechanism. Therefore, we include a web server in our NSLU2 application. The client may invoke the transmission of a JSON-RPC command to the nodes by sending a request looking like "send.rpc.cmd[port name, JSON-RPC command]" to the NSLU2. After having received this request, the application unpacks the JSON string, sends it to DSN node, awaits a response and sends it back to the client over a TCP socket. We will specify a special parameter to indicate when the NSLU2 has to send the JSON-RPC command to all connected DSN nodes. Table 3.2 lists the supported XML-RPC commands to call by the client.

XML-RPC command	**Function**
ports.list	The NSLU2 sends back a list of all serial ports with a plugged DSN node.
send.rpc.cmd[port name, JSON-RPC cmd]	Sends a command to the DSN node or to the target node
eeprom.write.cmd	Invokes the EEPROM programming procedure on the NSLU2

Table 3.2: Overview of the functions to call using the XML-RPC protocol on the NSLU2.

- **EEPROM programming procedure**
  The target programming is similar to the command transmission. At first, the user has to upload an EEPROM image. Therefore, he can call a website and upload the file. Once an image is present, the programming can start. The client has to send a command looking like ”eeprom.write.cmd[port name]” to the NSLU2. By the way, we decided to fragment the EEPROM code image on the NSLU2 and not on the PC for a first try. The NSLU2 then starts the programming of the EEPROM on the adapterboard by sending the data packets. The DSN node is responsible for the correct writing to the EEPROM. The NSLU2 application receives back the acknoledgements (*acks*) until the whole image has been transferred.

- **Log message capturing and storage**
  The main task of the NSLU2 is to capture the **raw** log messages from the different DSN nodes by reading the serial lines. The application proceeds the messages and stores them in a database, a so called logger pool. If a client logs in, the NSLU2 send the messages immediately over the TCP socket. In addition, the last 1000 log messages are displayed in a web site.

- **Web- and XML-RPC server**
  To upload the EEPROM image, to watch the log messages and to send XML-RPC commands to the NSLU2, we run a web server. The web server supports the request of web pages dynamically.

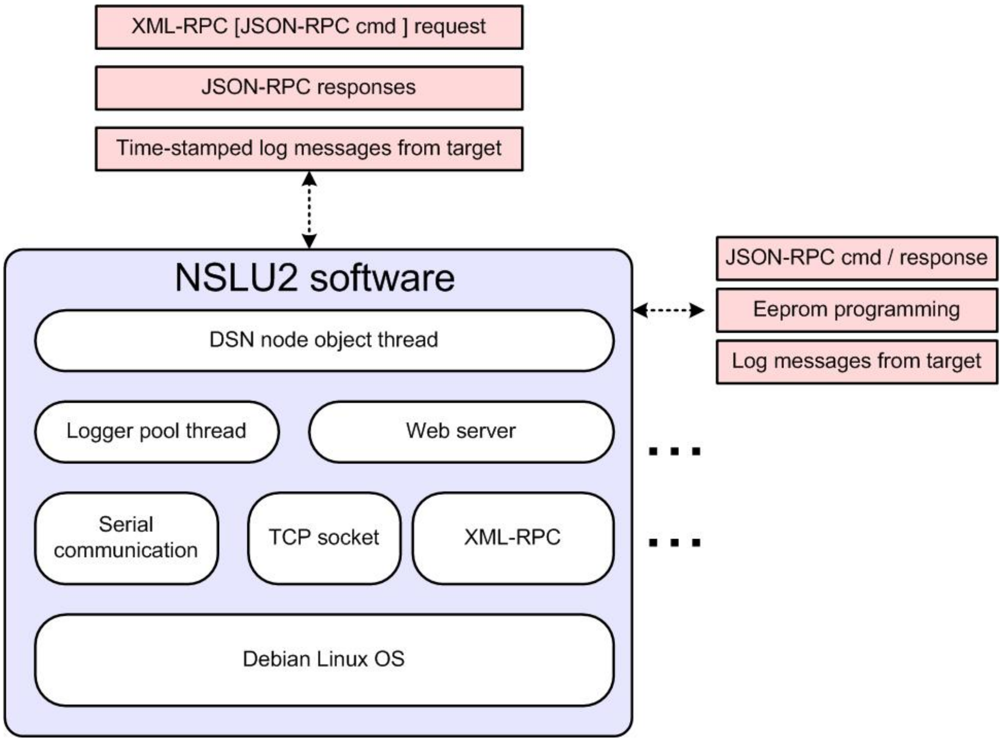


Figure 3.6: The NSLU2 software runs with a Debian Linux OS. The software provides an USB to Ethernet gateway, processes commands to the DSN nodes and forwards log messages to a network client such as the DSN Server.

Chapter 4

Hardware development

This chapter documents the development of the adapterboard hardware. Based on the conceptual design and the blockdiagram showed in the previous chapter we explain the different functional parts of the PCB, describe the components chosen and show the commissioning of prototypes. For detailed electrical circuitry or components information, see the schematics and components list in the Appendix.

4.1 Tools

For the development of the hardware, we took the standard software for electronics design Altium-Designer[1]. This tool provides all the functions to draw PCB schematics and layout and the candidate had already worked with it in previous projects. For simulation of electric circuits we employed SIMetrix[2] that is an open source software.

4.2 Hardware overview

Actually, we speak in most cases about the A80 as the target node. **Important is the fact that the adapterboard should work with any target node supporting the I^2C interface.** Some layout considerations had to be taken therefore and are explained later. Basically, the adapterboard has to provide an USB on one side and an I^2C interface on the other to communicate to peripheral components like target node or EEPROM. Since we want to reuse the BTnodes with batteries for some use cases, only a low power microcontroller is suitable for this adapterboard. As a matter of fact, we have to provide both USB and battery supply despite USB is the default solution.

[1] Version 2004

[2] SIMetrix Intro 5.3

Note that the specific designators are used here to refer to a specific pin of a component e.g. pin U4.1 means pin 1 of the chip U4 (MSP430F169 here).

4.3 Hardware architecture and system components

4.3.1 Microcontroller

We chose the MSP430F169[3] from Texas Instruments for this adapterboard as computing platform for the following reasons. It is a low-power 16-bit RISC microcontroller with 2 KB of RAM and 60 KB of flash memory. The average instruction cycle time is about 125 ns. It can be clocked up to 8 MHz and supports two physically separated USART modules that can be used either in UART, SPI or I^2C mode. That is exactly what fits our application, as we need an serial interface to the NSLU2 device (USB) and I^2C to connect the target node. So we do not need to implement I^2C in software. Many sensor node platforms e.g. TmoteSky [12] use this type of controller and the A80 as well. So a lot of know-how about this controller type is available at SBT.

We use the internal oscillator of the MSP430 for clock generation. But since it is not very accurate and robust to environmental conditions change, we added an external quartz crystal of 4 MHz to verify proper operation for timing constrained parts like UART communication. To have the possibility to set a general purpose input value of the MSP430 we integrated a DIL switch of eight pins, see Table 4.1. Four pins are used as controller inputs. A use case could be to restart the controller up on a change of this switch settings.

The MSP430 needs decoupling power capacitors to stabilise the input voltages. We dimensioned them according the controllers datasheet. A strong requirement is the capability of the controller to reset the target module to invoke the self-programming procedure. Therefore, U4.20 is used as output and connected to the reset input pin of the target connetor (pin X5.1). If desired, the target module can drive the two LEDs D9 and D10 (green and red) on the adapterboard PCB. For this purpose R17 and R18 have to be soldered. With adapterboard Rev0 and Rev1, they are not.

4.3.2 USB interface

For USB communication we integrated a connector X1 for data and power. The FT232BL[4] chip from FTDI interfaces the microcontroller. This module is an USB-UART converter and provides a virtual serial port on the host when the adapterboard is connected. In addition, it can be used for hardware flow control. Host

[3] datasheet under http://focus.ti.com/docs/prod/folders/print/msp430f169

[4] datasheet under http://www.ftdichip.com/Documents/DataSheets

drivers are open source and available from FTDI. We employed an external EEPROM for the FTDI here - not to mistake with the bigger I^2C-EEPROM to store the code image - for USB configuration settings storage. The chip requires a clock source. According the datasheet, we integrated a 6 MHz crystal with two 22 pF filter capacitors.

We use a ferrit bead in series to the USB power line to reduce noise being radiated down to the USB cable and secondly to protect the adapterboard from high current peaks. To indicate data receive or transmission, the FTDI can drive a green and a red LED respectively. If the adapterboard consumes more than 100 mA (BTnode start-up) the user has to make sure that the host or the USB hub delivers more than 100 mA (USB 1.0) or use battery power in parallel. A hub is allowed to deliver up to 500mA per line (practically it depends also on the amount of adapterboards plugged).

The 3.3 V supply output pin U1.6 is connected to U1.12 over the Shottky Diode D7 with a forward voltage of about 0.3 V at 10 mA to set the UART signal high level to 3 V.

4.3.3 Power module

The microcontroller, EEPROM and NAND ICs are supplied from a 3 V source voltage. We employ a Low-Dropout Regulator (*LDO*) here. This type of voltage regulator offers a minimum between the input and the output voltage (which is actually the dropout) so that it can still regulate the output voltage. In general, the LDO allows the battery to discharge to a few hundred millivolts above the desired output voltage. We do not use a switching regulator because with an LDO, we have a small noise. Since switching regulators do operate between 50 kHz and 1 MHz, they could affect the analog circuits or even the I^2C communication. The ultra LDO LP2985[5] from National delivers a 3 V output voltage and fits our application. It has a guaranteed 150 mA output current with the smallest possible size (SOT-23 micro SMD package). Furthermore, it requires just a few external components.

Power for the BTnode and the target node is delivered from either batteries or USB directly, since both devices have a voltage regulator on their PCBs. The battery voltage must not be higher than about 6.2 V for the new radio module. The decoupling capacitors are chosen according the regulators reference schematics.

We added a nice feature to the adapterboard. When the USB cable is plugged, the power switch should automatically separate the battery supply line in order to save energy. Figure 4.1 shows the schematic of this power module. When USB

[5] datasheet under http://www.national.com/pf/LP/LP2985

is plugged, the npn Transistor T1[6] is floating. Therefore, the gate voltage of the n-JFET T2[7] is almost zero and the gate-source voltage below the cut-off voltage of about -3.2 V (the four AA batteries lowest voltage is about 0.8 V each which yields to a sum of 3.2 V). As a result, the n-JFET is turned off separating the battery supply line. If no USB is plugged, the gate-source voltage of T2 is almost equal to zero due to the high impedance R12 and therefore T2 is turned on. We made a SIMetrix simulation of the power switch circuitry that worked satisfyingly. **Alternatively, if the power switch would not operate properly, we added a switch that can be soldered to turn the battery power on and off by hand.**

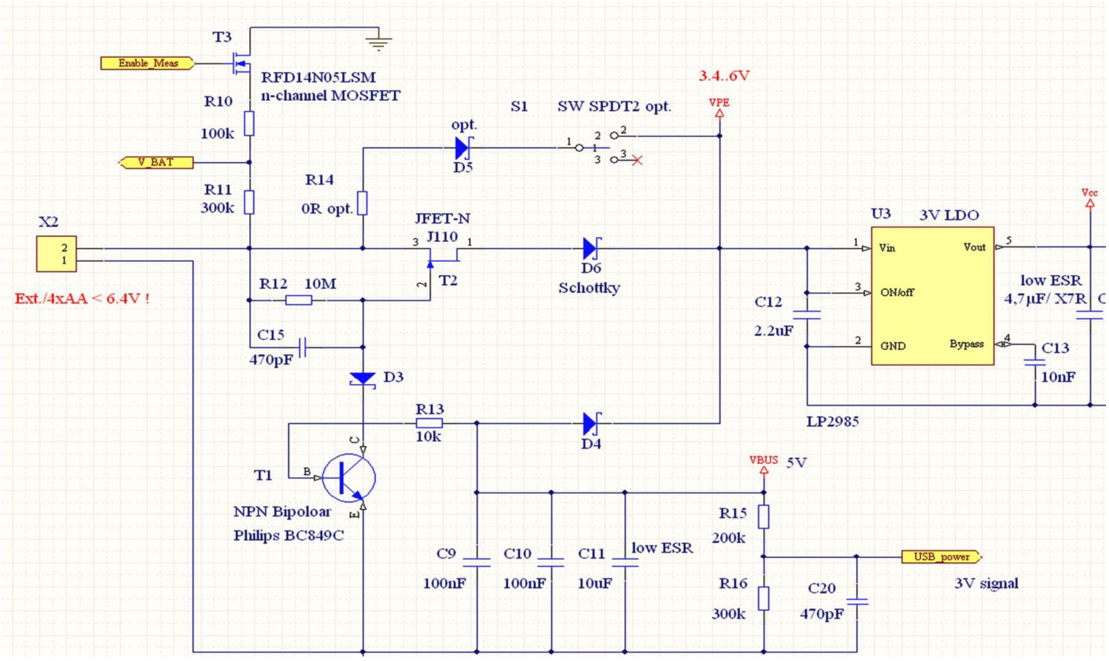


Figure 4.1: Power module of adapterboard with battery connector X2, switch and voltage regulator on the right.

4.3.4 Battery measurement circuit

Using a simple high-impedance voltage divider, the controller can measure a forth of the current battery voltage value, namely the signal V_Bat, as input. The MSP430 can drive the MOSFET T3 to enable the measurement by software.

[6] datasheet under http://www.farnell.com/datasheets/12788.pdf
[7] datasheet under http://www.vishay.com/docs/70231/70231.pdf

4.3.5 I^2C bus

The adapterboard communicates with the peripheral devices e.g. the target node over an on-board I^2C bus. According to the bus specification [17], two 1.5k-pull-up resistors are required for the signals SCL and SDA because the bus signals are idle high. Since any target node could have the pull-up resistors on its PCB, no pull-up resistors are needed any more on the adapterboard. Therefore, the two soldered pull-up resistors are not directly connected to V_{cc} and can be "pulled down" by the MSP430 software over the outputs U4.28 and U4.30.
We utilise two Diodes D11 and D12 for bus overload protection. The resistors R32 to R35 are for current limitation. The peripheral phone jack X4 allows the use of a debugging device with I^2C master, slave and spy functions. The device e.g. can act as I^2C slave on a specific I^2C address.

4.3.6 Bootstrap Loader circuit

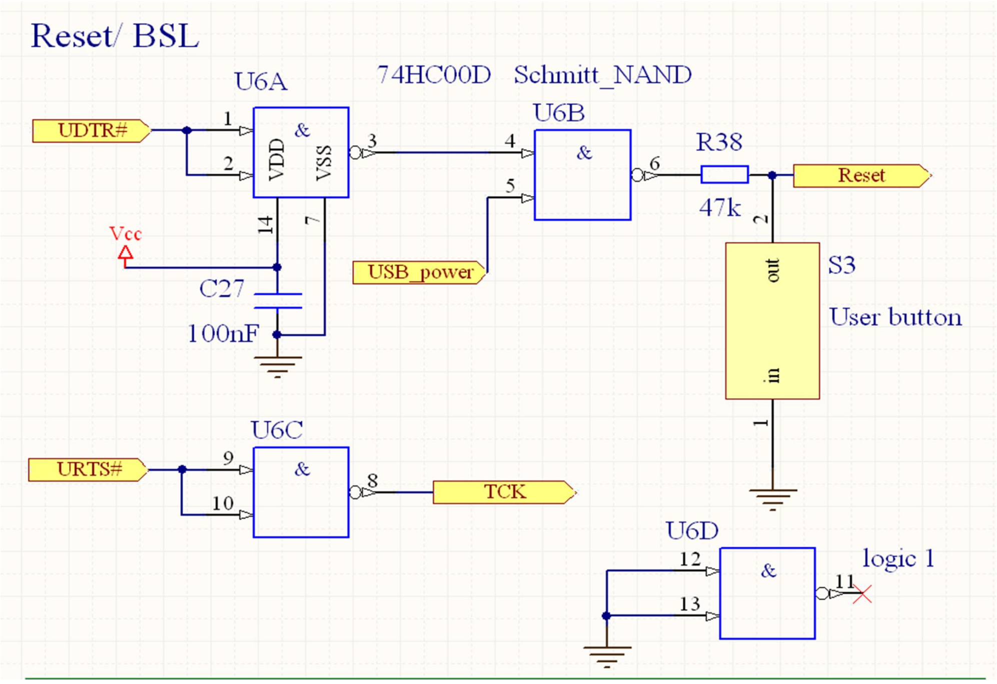


Figure 4.2: The Bootstrap Loader circuit disables UDTR# when USB is not plugged.

The MSP430 microcontroller supports programming over the serial interface by using the so called Bootstrap Loader (*BSL*), see [24] for further information. Both the controller's flash and data memory can be modified. Therefore, the UART protocol is used. To invoke the BSL, a certain BSL entry sequence has to be applied to specific device pins TCK and Reset of the MSP430. Therefore, the

pin	signal name	function
1	Eeprom_WP	EEPROM write protection (active high)
2	Eeprom_A0	EEPROM I^2C address lsb
3	Eeprom_A1	EEPROM I^2C address bit 1
4	Eeprom_A2	EEPROM I^2C address bit 2
5	U4.6	controller input
6	U4.5	controller input
7	U4.4	controller input
8	U4.3	controller input

Table 4.1: Usage of the DIL switch pins.

signals UDTR# and URTS# of the FTDI chip are used. Both are active low. To start the BSL, UDTR# has to hold the controller in reset by applying the low voltage signal (0 V here) to the controllers reset pin U4.59. But if no USB is connected e.g. when we have the BTnode with batteries operation, the FTDI chip's UDTR# having a low level would also drive pin U4.59 low, despite no BSL should be invoked. To overcome this problem, we designed a small but effective BSL circuit. This solution employes a quad 2-input NAND gate chip[8] with ESD protection supplied by V_{cc} from the voltage regulator. The NAND verifies that the controller can only be reset when both USB is plugged and the UDTR# is active, see Figure 4.2. The URTS# has to be inverted for proper operation, since it is active low. To allow to reset the adapterboard by hand, a user button is applied.

4.3.7 JTAG

To allow debugging and programming of the adapterboard, alternatively a JTAG [25] programmer can be employed plugged to connector X3.

4.3.8 EEPROM

One of the most important components on the adapterboard is the EEPROM chip U5 interfaced to the I^2C bus. We chose the 24LC512 CMOS Serial EEPROM from Microchip[9] with 64 KB of memory. It is supplied also from V_{cc} and supports a bus speed of 400 kHz. A 128-byte page write mode is available with a 5 ms maximum write cycle time.

We use the 8-pin DIL switch S2 to set the three least significant address bits of the EEPROM by hand, see Table 4.1. In addition, the provided write protection function at U5.7 can be activated by setting pin 1 of the DIL switch high.

[8] datasheet under http://www.alliedelec.com/Images/Products/Datasheets

[9] datasheet under http://ww1.microchip.com/downloads/en/devicedoc/21754E.pdf

4.3.9 Peripheral connectors

The adapterboard has three connectors X5 to X7 to interface the A80, the BTnode and an extension connector.

The X5 is a 20-pin connector. Pin 1 is the input for target reset which is **active high**, as already mentioned before. Over pin 19, the radio module is supplied from V_{Pe}. Pin X5.14 and X5.16 are for I^2C communication. The other connected pins are general purpose in- and outputs controllable by the target node software and connected to X7 each. The Jumpers J1 to J3 are for reuse of the old adapterboard software version and do connect the serial interface signals TXD and RXD of the BTnode with the I^2C signals SDA and SCL. These signals are also available at X7 and as inputs at the controller. **Jumper J2 has to set Out1 high when the A80 is plugged.**

The BTnode connector is attached to the I^2C bus (pins X6.12 and X6.13) and the input for the supply voltage V_{Pe} is at pin X6.17 and X6.18 respectively. To allow the BTnode to measure the actual battery voltage state on the adapterboard, the signal V_Bat is yielded to X6.10 as input over a 0 Ω resistor.

The extension connector X7 is for debugging reasons e.g. to add LEDs or to observe the I^2C bus signals. The general purpose in- and output signals from the target In1 to In3 and Out1 to Out3 are accessible here. Furthermore, all these signals are yielded to the controller as interrupts (if enabled).

4.4 Layout considerations

After having completed the adapterboard schematics, we made the layout. Here, we used the constraints listed in Table 4.2. For more detailed layout information, see the Appendix. Generally, we placed the components according to functional belongings. Decoupling capacitors are as close as possible to the chips. In order not to create antennas we took care to make the ground planes area-wide with no thin tails. We tried to place the trough-hole components e.g. X5 so that it would not touch the metallic part of the Blue Boxes battery case later in order to prevent short-circuits. We also tried to keep the signal wires close together to minimize inductive coupling but also short to reduce capacitive coupling. The latest criteria often conflicts with the first one. However, in most cases we stroke a balance between the two.

4.4.1 Mechanical design constraints

To fix the adapterboard in the Blue Box with screws later, we mad holes at each corner of the PCB. In addition, we made two holes to employ two plastic spacer

PCB design constraint	Size
number of layers:	2
minimum space between wires, vias:	0.2 mm
signal layer width:	0.2..0.3 mm
power line width:	0.2..0.8 mm

Table 4.2: Overview of the different wire-widths

for A80 short-circuit protection. We use plastic screws since we want no electromagnetic induction from the BTnode antenna. The antenna is situated directly above a screw. The PCB has a size of about 125 mm times 70 mm. We ordered 12 PCBs from PCB Pool/Beta LAYOUT GmbH[10] in Germany for the price of 1160 CHFr. Considering the component costs a single assembled adapterboard has a price of about 165 CHFr.

4.5 Prototype assembly and commissioning

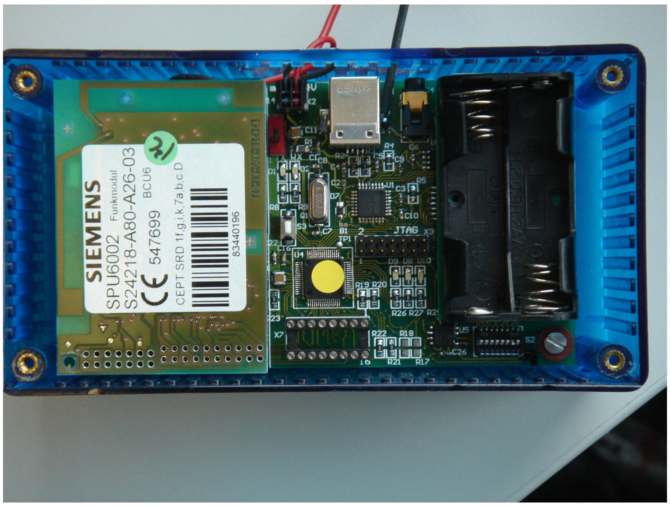


Figure 4.3: The new USB-adapterboard in a Blue Box. A Siemens A80 and a BTnode are plugged.

After having received the manufactered PCBs we started with the assembling of four adapterboards. This task asked for a special assembling order and a precise work, since most of the components are relatively small and SMDs[11], see Table 4.3. First of all, we soldered the power circuit and then the FTDI chip and its peripherals e.g. control LEDs. Step by step we integrated the various functional

[10] http://www.pcb-pool.com
[11] Surface Mounted Devices

parts and verified their proper behaviour e.g. powering the board from batteries and USB. In between, we could already test the USB interface by plugging the device to the PC and installing drivers when the FTDI chip had been recognized. Alternatively, all the ICs could be soldered first because they are in general more difficult to put on the PCB.
At the end, we managed to load a piece of test software over the BSL to the controller's flash at the first try. When the assembled adapterboard was ready, we inserted it into the Blue Box, see Figures 4.3 and 4.4.

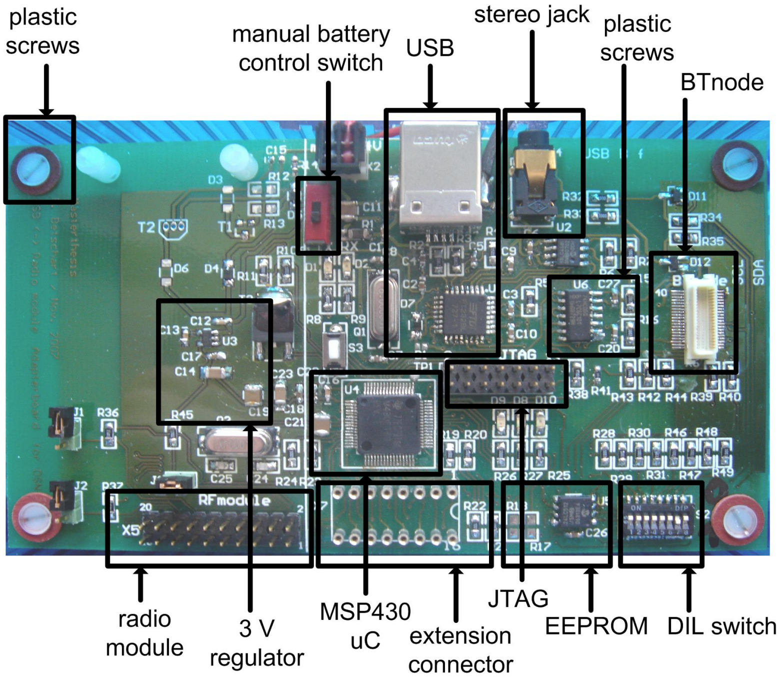


Figure 4.4: Overview of soldered components of adapterboard.

Once four adapterboards run a testsoftware that let the green, yellow and red LED blink, we assembled the remaining eight PCB. Because we could not manage to get the power switch working as intended after spending hours with trying with different other transistors and resistors, we decided to solder the second possibility namely the S1, D5 and R15 to switch off battery by hand. During the commissioning of the adapterboards it happened sometimes that the BSL or the supply of the controller did not work up on the first try due to bad soldering. Finally, they all behaved as intended, see chapter 7. We determined some failures of the schematics and in the layout what we have improved and released under adapterboard Rev1. In the Appendix, this revision's schematics and layout drawings are inserted. Important to mention here is that we added the resistor R50 close to the NAND chip to allow JTAG programming since this did not work with Rev0.

Step	Soldered components	Assembly check
1	X1, X2, C1, R1, S1, D5, C20, D4, R10, R11, R14, R15, R16, C11, C19, U3, D7, C12, C13, C14, T3, C17, C22	Turn off battery power by hand, 3 V output required from voltage regulator.
2	U1, R2, R3, C4, C5, R4, Q1, C7, C8, C2, C3, C9, C10, R8, R9, D1, D2, U2, R6, R7, C6, R5	Connect PCB to PC over USB, try to install drivers.
3	U4, U6, C27, R22, R23, C18, C22, C16, R24, R19, R20, R21, R25, R26, R27, D8, D9, D10, C23, Q2, C25, C24, C21, R38,	Load test software onto the controller to let some LEDs blink.
4	X3, X4, R32, R33, R34, R35, R36, D11, D12, R37, U5, C26, J1, J2, J3, R45, X5, R21, X7, X6, R44, R43, R43, R39, R40, R41, R28, R29, R30, R31, R46, R47, R48, R49, S2, S3	Reset controller, plug A80, plug BTnode and check for blinking

Table 4.3: Assembly plan of the adapterboard hardware step by step (without automatic power switch circuitry).

Chapter 5

Firmware implementation

The following chapter explains the implementation of the actual adapterboard firmware based on the conceptual design described in Chapter 3. We do rather focus on the firmware development here than on how to program the MSP430, since this is shown in the Readme in the Appendix. For implementation details we refer to the source code available from the CD inserted at the end of this thesis.

5.1 Programming language and environment

We use the standard C [30] as the programming language for the MSP430 controller firmware for the following reasons:

- The controller's hardware architecture is designed for C. Pointers, arrays and stacks are relatively easily programmable.
- Texas Instruments provides development tools like compiler and debugger for the MSP430 for free.
- Various libraries and code examples are available as open source.
- The candidate had already certain experience in working with C.

We employed the *MSPGCC* toolchain [31] including compiler, linker, assembler and debugger. There are also various commercial toolsets available but the MSPGCC compiler is suitable enough for our purpose. We do not need highly optimized object code size or speed here what is traditionally provided by commercial distributors.

5.2 Firmware architecture

The conceptual firmware design was made in Chapter 3. Here we show in more detail e.g. how the firmware proceeds the commands and log messages. The source

Files	Implementations
config.h	baudrate definitions, EEPROM address definition memory settings
hardware.h	port definitions, WDT configuration, clock settings
usb.h	baudrate settings, buffer allocation, event definitions, evenhandler declarations
usb.c	evenhandler table definition, UART1 settings, UART1 interrupt handler definition, idle eventhandler, definition, JSON-RPC command processing, JSON-RPC procedure call
peripheral.h	adc function declarations, battery measurement functions declarations
peripheral.c	WDT eventhandler definition, battery measurement function definitions, timer definition, DIL switch settings
rpc_commands.c	JSON-RPC commands declaration, JSON-RPC command table definition, JSON-RPC command definitions
i2c_bus.h	EEPROM page size definition, target command buffer allocation
i2c_bus.c	log message buffer allocation, I^2C module configurations, I^2C bus and EEPROM access function definitions, UART1 interrupt handler definition
main.c	WDT interrupt handler, hardware, UART1 initialisation, baudrate selection, DIL switch read, global interrupt enable, I^2C slave configuration, target reset, eventhandler start

Table 5.1: Source code organisation of the adapterboard firmware.

code is organised as listed in Table 5.1. We explain the firmware's program flow as follows and treat the most important parts of the code in the following subsections.

5.2.1 Program flow

In Figure 5.1 a flow chart of the firmware is drawn. When the controller is reset, it starts initialising the hardware, the WDT and configures itself as I^2C slave in order to receive log messages. After configuring, it executes test routines if the corresponding code section is uncommented in main.c. As a last step, the firmware resets the target node and gives the control to the eventhandler.

We implemented four eventhandlers. The firmware executes the idle even-

Eventhandler	**Priority**
event_command_key_received	1
event_log_message	2
event_wdt	3
event_idle	4

Table 5.2: Overview of the implemented eventhandlers and priorities.

SMCLK	12.5 kHz
ACLK	4 MHz
MCLK	4 MHz
UART1 baudrate	230400 kbps
EEPROM page size	128 B
EEPROM I^2C bus address	81
JSON-RPC command buffer size	330 B
JSON-RPC target command buffer size	120 B
log message buffer size	230 B
MSP430 I^2C slave address, bus clock speed	20, 100 kHz
MSP430 I^2C master address, bus clock speed	84, 400 kHz

Table 5.3: Adapterboard firmware settings.

thandler toggling the red LED when no higher prioritised handler has been launched or is running. The WDT wakes up with a dutycycle of about 2.6 seconds on an interrupt and launches the WDT eventhandler. Note that the eventhandler routines always run to completion but are preempted by interrupts for sure.
Table 5.2 shows the implemented handlers and its priorities (priority 1 is the highest). Whenever an eventhandler has finished, the controller checks for launched handlers and runs them according to the priorities. That concept allows the quasi-parallel execution of different tasks like JSON-RPC command processing or log message transfer. Important to mention here is that the MSP430 is I^2C bus slave but can be configured as I^2C bus master over an JSON-RPC command (required for EEPROM programming).

5.2.2 Settings

Here we list the actual settings of important parameters, see Table 5.3. Note that these values are fixed at the compile time and the user can not change them dynamically e.g. over an JSON-RPC command.
The controller employs two clock sources (external 4-MHz-oscillator and the in-

ternal DCO) to create the three clock signals SMCLK, ACLK and MCLK, see [32] for signal and register naming. The UART1 for USB communication and UART0 for I^2C use the ACLK for baudrate and bus clock speed calculation. The I^2C bus runs at a clock speed of 100 kHz when the target node is the master, 400 kHz when the adapterboard controller is the master. The MCLK is the controller's main clock and set to 4 MHz whereas the WDT uses the SMCLK. **Note that according to [32], the I^2C bus clock must not be higher than the clock source (ACLK) divided by ten**.

5.2.3 JSON-RPC processing

The adapterboard recognizes a JSON string and executes the requested procedure according the format defined by the JSON-RPC specification [26]. A list of the supported commands can be found in table 3.1. Basically, the whole command execution procedure starts when the controller has received the character ”{” up on a UART1 interrupt. All the following characters are stored in a ringbuffer then. After having stored a character, always the event event_command_key_received is launched checking if already a ”}” has arrived. This character indicates the command string's end. In that case, the firmware reads the buffer and executes the corresponding function then. The functions are defined in a global table. Figure 5.2 shows an example of a JSON-RPC command format to invoke the adapterboard to reset the target.
When a command should be forwarded to the target, the firmware receives a request with the method ”forward.command” and fills a buffer that is polled and flushed by the target node later with the unpacked JSON-RPC string for the target. The adapterboard always replies back to the sender when a command execution was succesfull.

The method that forwards a command to the target node employs double buffering. One buffer is filled whereas the other one can be read by the target. This concept ensures that no command characters are mixed, since the A80 reads only one byte per I^2C interrupt. Figure 5.3 shows the concept.

5.2.4 Log message output

Any data arriving from the target module is handled as log messages (despite it is a JSON-RPC answer) and displayed over USB. Therefore, the characters are stored in a ringbuffer. When an EOL[1] character has been received, the event event_log_message is launched. This eventhandler lights the green LED, prints out the log message and clears the LED again. In between, the ringbuffer can be filled with new log message characters up on an I^2C interrupt.

[1] End of line

5.2.5 I^2C and EEPROM access

The firmware provides lots of functions to access the EEPROM[2] (not all of them are utilised in the actual revision).
Besides the functions I2C_configure_slave and I2C_configure_master, the method write_page is employed to transfer a code image packet from the NSLU2 to the EEPROM. To increase the transfer speed, the function polls the EEPROM for acks to determine the internal write cycle's completion. Figure 5.4 shows the execution steps of the function write_page.

5.2.6 Low power mode

We used the Low Power Mode 0 (*LPM0*) provided by the MSP430 in a previous firmware version. The idle eventhandler put the controller into this mode. This yielded to missed log messages from the target node. We suppose that the LPM0 and its wake-up procedure influence the correct hardware setting of the UART0 interrupt flags due to clock timing problems. So in the actual firmware version the LPM0 is disabled. **Note that the configuration of the UART0 for I^2C has to be done precisely according to the instructions in** [32].

5.2.7 Target node programming

Here, we describe the already implemented software behaviour of the target node. After each reset of the target node, the target node bootloader checks if an update is required and set a certain flag. It scans the I^2C bus at addresses 80 to 82. It polls for an EEPROM, if one is not accessible, it goes to the next. If a code segment is detected and the a tag entry is new, it loads the code block. In order not to load the same software image again, the target node updates the tag entry in the EEPROM by writing the own serial number.

[2] A lot of information and how to access the EEPROM by software can be found in http://ww1.microchip.com/downloads/en/devicedoc/21754E.pdf

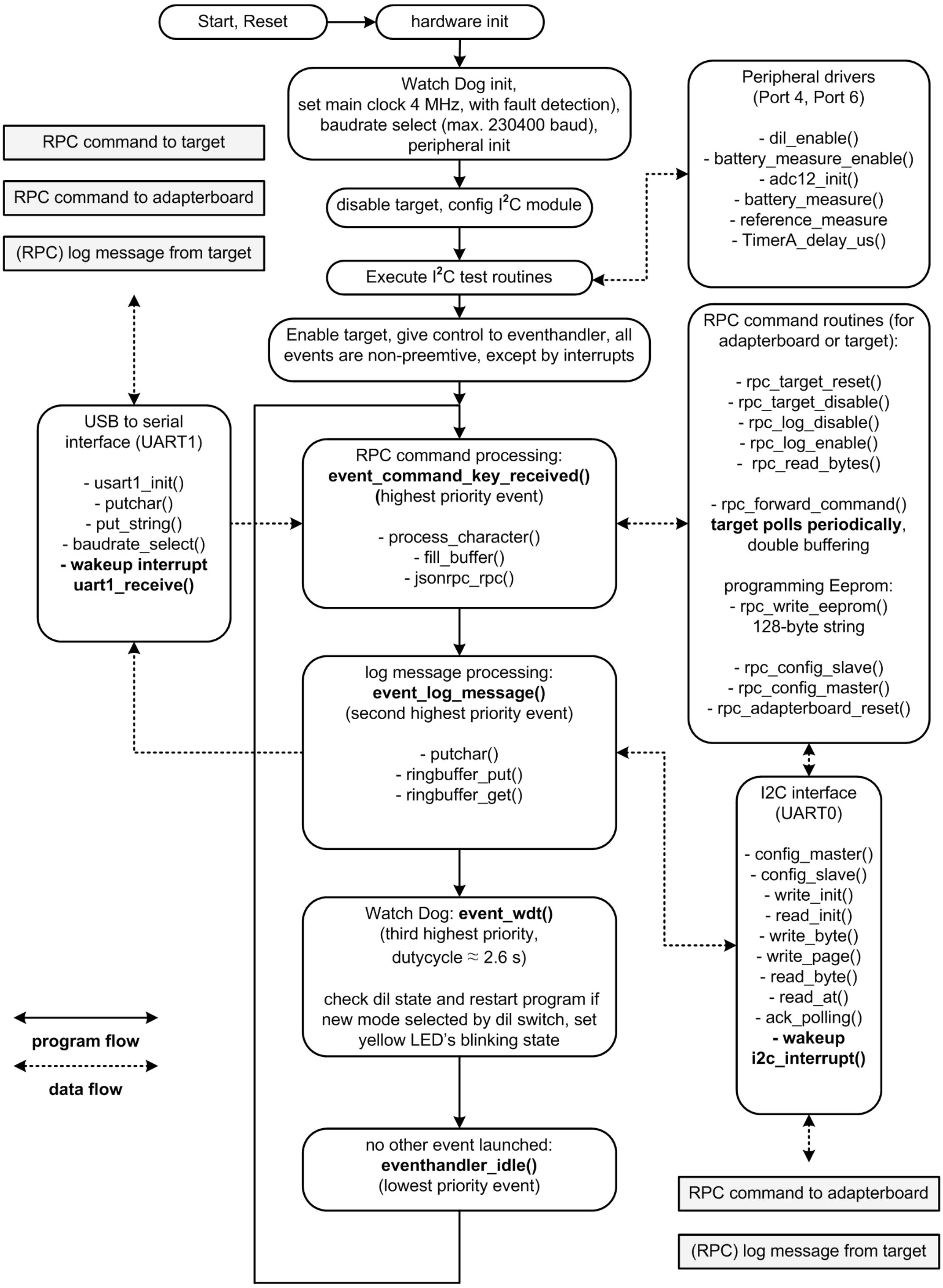

Start, Reset
hardware init
Watch Dog init, set main clock 4 MHz, with fault detection), baudrate select (max. 230400 baud), peripheral init
disable target, config I^{2}C module
Execute I^{2}C test routines
Enable target, give control to eventhandler, all events are non-preemtive, except by interrupts
RPC command to target
RPC command to adapterboard
(RPC) log message from target
Peripheral drivers (Port 4, Port 6)
- dil_enable()
- battery_measure_enable()
- adc12_init()
- battery_measure()
- reference_measure
- TimerA_delay_us()
USB to serial interface (UART1)
- usart1_init()
- putchar()
- put_string()
- baudrate_select()
- wakeup interrupt uart1_receive()
RPC command processing: event_command_key_received() (highest priority event)
- process_character()
- fill_buffer()
- jsonrpc_rpc()
RPC command routines (for adapterboard or target):
- rpc_target_reset()
- rpc_target_disable()
- rpc_log_disable()
- rpc_log_enable()
- rpc_read_bytes()
- rpc_forward_command()
target polls periodically, double buffering
programming Eeprom:
- rpc_write_eeprom()
128-byte string
- rpc_config_slave()
- rpc_config_master()
- rpc_adapterboard_reset()
log message processing: event_log_message() (second highest priority event)
- putchar()
- ringbuffer_put()
- ringbuffer_get()
I2C interface (UART0)
- config_master()
- config_slave()
- write_init()
- read_init()
- write_byte()
- write_page()
- read_byte()
- read_at()
- ack_polling()
- wakeup i2c_interrupt()
Watch Dog: event_wdt() (third highest priority, dutycycle ≈ 2.6 s)
check dil state and restart program if new mode selected by dil switch, set yellow LED's blinking state
program flow
data flow
no other event launched: eventhandler_idle() (lowest priority event)
RPC command to adapterboard
(RPC) log message from target

Figure 5.1: Program flow of the adapterboard firmware. After a start-up sequence, the firmware switches between various eventhandlers to allow quasi-parallel execution of JSON-RPC commands and log message capturing.

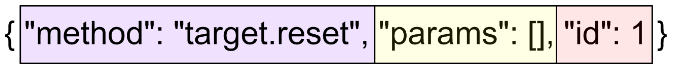


Figure 5.2: A JSON-RPC command includes three parameters. Method is the string containing the name of the method in the global table to invoke, params is an array of arguments to pass to the method if required and the id. The id is necessary to match the JSON-RPC response later with the corresponding request.

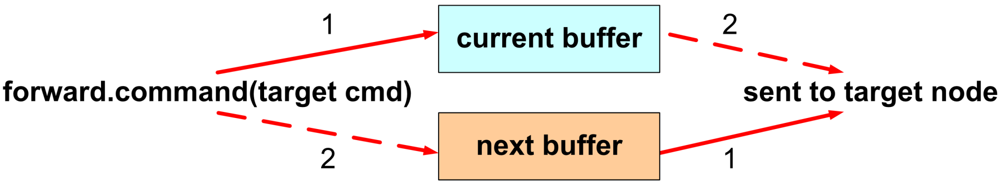


Figure 5.3: If the JSON-RPC command is sent to the adapterboard to forward a command to the target node, the adapterboard fills a buffer and sets a pointer to a second buffer. The target polls at I^2C bus address 20 and reads the first buffer. When a second command should be transferred, the command is written to the second buffer. Next, the target reads the second buffer and so on.

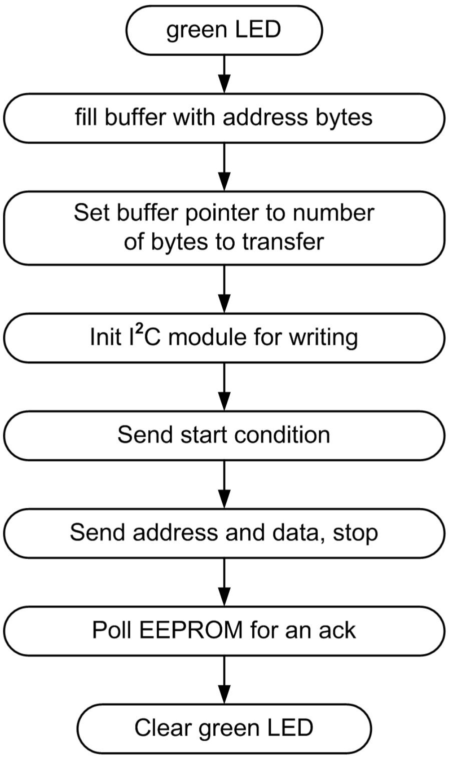

green LED
fill buffer with address bytes
Set buffer pointer to number of bytes to transfer
Init I²C module for writing
Send start condition
Send address and data, stop
Poll EEPROM for an ack
Clear green LED

Figure 5.4: EEPROM page write execution when the adapterboards controller is the bus master.

Chapter 6

NSLU2 software

We have defined the functional features of the NSLU2 software in the conceptual design in Chapter 3. In this chapter we address the implementation and explain how the software is organised.

6.1 Programming language

Python [33] is an object-oriented scripting programming language and distributed as open-soure even for commercial products. All what is needed to execute Python is an editor to create the scripts and the Python interpreter[1] executing the scripts line per line. We decided to use Python for the NSLU2 software development for the following reasons:

- Python runs under Linux/Unix and Windows, so we do not have to care about OS type if we would change the platform.

- Creating a TCP socket to communicate with a remote computer requires only a few lines of Python code.

- The language is not that difficult to learn and has relative short and simple code structures.

- Python Libraries are upgraded frequently and maintained carefully.

A drawback was that the candidate had no experience in programming with Python, but it took only a few days to get familiar with the basics.

Setting	Variable name	Value
IP of NSLU2	IP	10.169.26.4
Port number (HTTP, XML-RPC)	http_port	8080
TCP sockets	Port	7000
Serial baudrate	BAUDRATE	230400 kbps

Table 6.1: Settings in the NSLU2 software.

6.2 Software implementation

6.2.1 Overview and software settings

We installed a Debian Linux on the NSLU2 device and made some other NSLU2 OS ”preparations”, see in the Readme. The implemented software consists only of two scripts. We had to include various so called modules for different functionalites like time, sockets or the serial communication. A module is another Python file containing classes and definitions. The user can import such definitions into his main file and create instances of the classes or redefine methods of the classes.
The NSLU2 software is mainly the script ”tcpserver.py” including the module ”rpcservice.py”, which had to be adapted for our purposes. To start the software, the user has to start the first script on the NSLU2. Since the NSLU2 is connected to Ethernet, some settings are needed and are defined as global variables, see Table 6.1.

6.2.2 Software flow chart

We employ here multi-threading meaning all actions e.g. sending a web page to the client’s web browser are handled by different threads.
The initial procedure is to check for plugged adapterboards, see Figure 6.1. This is done by trying to open a serial connection for a specific port name e.g. /dev/ttyUSB0 for Linux here (COMx for Windows is implemented as well). Up on success, the script stores the available port names in a global list and starts a object thread for each. The treads are responsible for all interactions with a single adapterboard and target node respectively e.g. reading a log message from the serial line or sending a JSON-RPC command. Next, a further thread is started that listens to client computers to log in over a TCP socket and sends then the log messages. In addition, this thread updates the web page with the log messages from the target modules and the one with the list of plugged adapterboards. **Note that the actual version does not support un- and replugging of adapterboards.** As the last step, the software starts both the web and XML-RPC server

[1] We used Python 2.4.

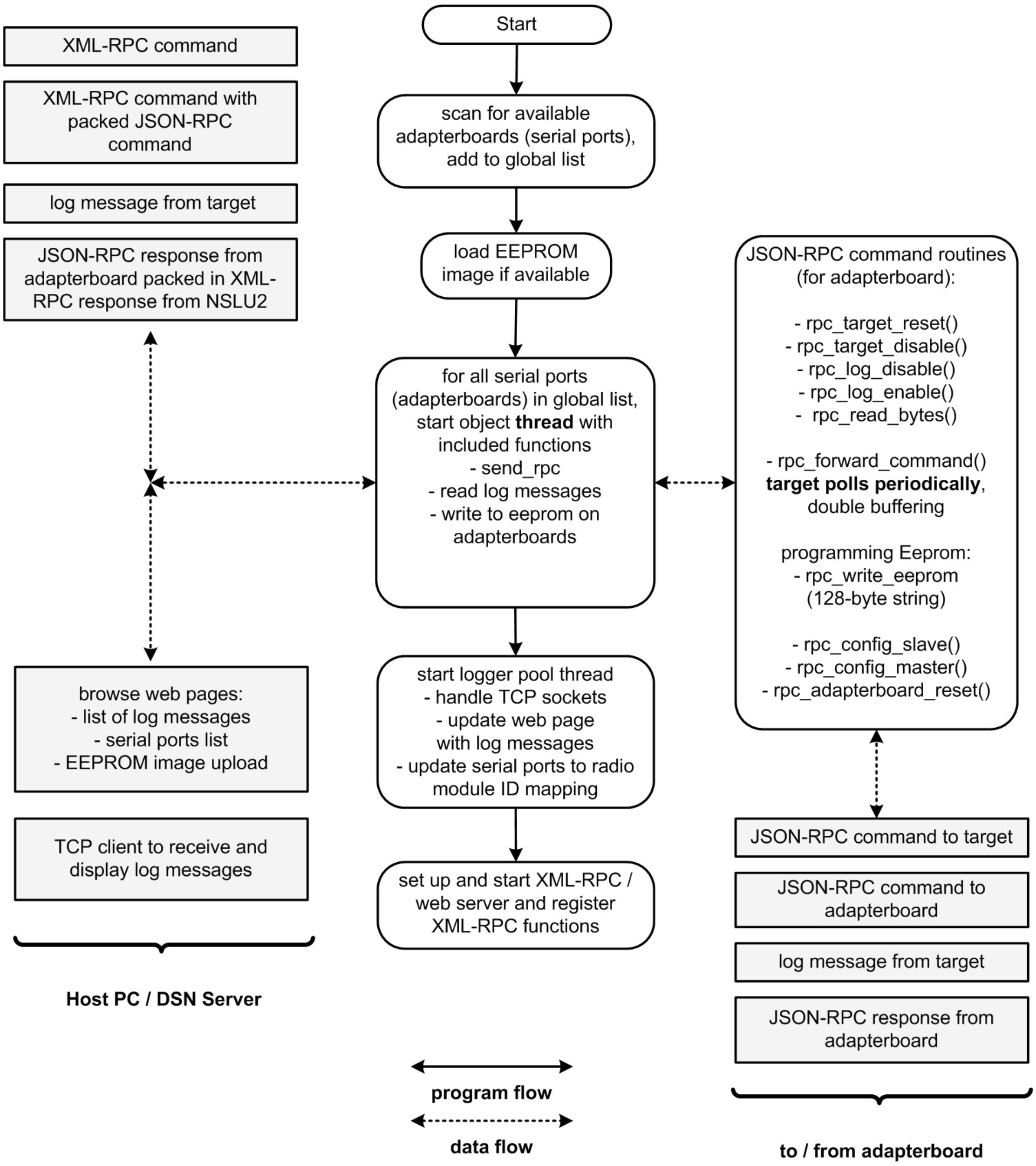

XML-RPC command
XML-RPC command with packed JSON-RPC command
log message from target
JSON-RPC response from adapterboard packed in XML-RPC response from NSLU2
browse web pages:
- list of log messages
- serial ports list
- EEPROM image upload
TCP client to receive and display log messages
Host PC / DSN Server
Start
scan for available adapterboards (serial ports), add to global list
load EEPROM image if available
for all serial ports (adapterboards) in global list, start object **thread** with included functions
- send_rpc
- read log messages
- write to eeprom on adapterboards
start logger pool thread
- handle TCP sockets
- update web page with log messages
- update serial ports to radio module ID mapping
set up and start XML-RPC / web server and register XML-RPC functions
program flow
data flow
JSON-RPC command routines (for adapterboard):
- rpc_target_reset()
- rpc_target_disable()
- rpc_log_disable()
- rpc_log_enable()
- rpc_read_bytes()
- rpc_forward_command()
target polls periodically, double buffering
programming Eeprom:
- rpc_write_eeprom (128-byte string)
- rpc_config_slave()
- rpc_config_master()
- rpc_adapterboard_reset()
JSON-RPC command to target
JSON-RPC command to adapterboard
log message from target
JSON-RPC response from adapterboard
to / from adapterboard

Figure 6.1: Flow chart of the execution of the NSLU2 software.

respectively.

Figure 6.1 also shows the data flow. A list of the supported XML-RPC functions is given in Chapter 3.

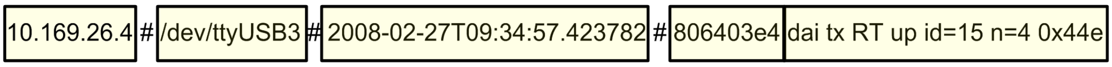
URL	Contents
http://10.169.26.4:8080/DSN/logs	list of the 1000 latest received log messages from all attached adapterboards
http://10.169.26.4:8080/DSN/list	table of busy serial ports with corresponding adapterboards and target node IDs
http://10.169.26.4:8080/upload.html	EEPROM image upload page
http://10.169.26.4:8080/DSN/test	DSN test page

Table 6.2: List of the supported web pages from the NSLU2 software.

6.2.3 Web applications

Once the NSLU2 software has been started, the user can request the web pages listed in Table 6.2 using a web browser.

6.2.4 Log message format



```
10.169.26.4#/dev/ttyUSB3#2008-02-27T09:34:57.423782#806403e4dai tx RT up id=15 n=4 0x44e
```

Figure 6.2: Data format of the log message sent to the PC.

In Picture 6.2 we can see the log message format sent to the PC and displayed on the web page. The NSLU2 software takes the raw message from an adapterboard and adds its IP address, the serial port name and a time stamp.

6.2.5 XML-RPC format

The exact command format to send to the NSLU2 is described in the Readme in the Appendix.

6.2.6 EEPROM programming with NSLU2

In Figure 6.3 we see the illustrated EEPROM programming procedure. The user has to upload an EEPROM code image for the target node first over the web broser (http://10.169.26.4:8080/upload.html). Afterwards, the transmission to the adapterboards is started when the NSLU2 receives the command looking like ”eeprom_write_cmd(port_name)” or ”eeprom_write_cmd(”broadcast”)” and executes the function ”eeprom_write”. The NSLU2 sends the command to hold the target node in reset and to configure the adapterboard as I^2C bus master, transfers the image

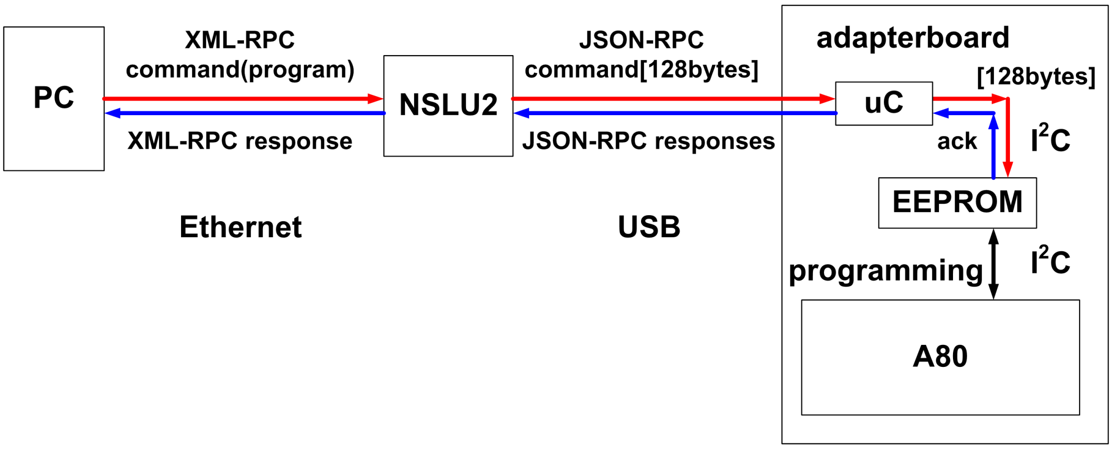


Figure 6.3: Schematic description of the EEPROM programming procedure started from PC.

data packets using the JSON-RPC protocol and sends the command to configure the adapterboard as I^2C bus slave and reset the target node again. Afterwards, the target node reloads the image from the EEPROM and reprogrammes itself. **Note that the image data bytes are encoded to hexadecimal representation before they are sent to the adapterboard.** The firmware decodes the data before sending to the EEPROM.

Chapter 7

Evaluation

7.1 Overview and test cases

This chapter describes the functional and non-functional performance evaluation of the new Ethernet-based DSN built within this work. In order to verify the correct functionality and expected behaviour of the different components as well as the whole system, we defined the following test cases:

1. **Adapterboard hardware**
   The adapterboard with plugged radio module as target node is the core component of the DSN. The correct hardware behaviour has to be inspected in detail.

2. **Adapterboard firmware with plugged A80**
   The DSN node software has to capture all the log messages from the A80 and to handle the JSON-RPC commands. An in-deep study of the application behaviour is highly recommended.

3. **NSLU2 software and Ethernet-based test DSN**
   This test is separated in two parts. One part proves the NSLU2 software basic functionality including adapterboard management. The second part tests a 8-node-DSN system's performance consisting of adapterboards, target nodes and a single NSLU2. We also compare our DSN with the BTnode-based one.

4. **Backwards compatibility to BTnode-based DSN**
   Here, we plug a BTnode to the adapterboard and investigate its correct operation as DSN node.

In the following sections, we present the results and conclusions obtained from the test cases described above. All tests took place inside the SBT building GS2a in Zug.

7.2 Adapterboard hardware

This section covers the hardware evaluation of the Adapterboard.

7.2.1 Measurement and test equipment

In this tests the following devices and tools were used:

- Tektronix Digital Oscilloscope TDS 1012B 100 MHz, 1 GS/s, No. 63999
- Multimeter FLUKE 183 TRUE RMS, Siemens No. 63404
- Device Under Test (DUT) adapterboard Rev0 in Blue Box No. 73
- Siemens A80 with ID 806403F7
- BTnode Rev3.24, 20/06, MAC:00043F00015B
- FDUZ221 MCLink-USB adapter with I^2C master, slave and spy functions
- Serial terminal program msp430-miniterm
- Serial USB cable for data traffic and powering of the adapterboard
- USB host: Fujitsu Siemens PC with Windows XP, SP2
- USB driver for standard FTDI Chip provided by FTDIChip
- USB driver for FDUZ221 provided by SBT
- A80 software: detectornode, Rev. 2513
- Adapterboard firmware Rev. 2456

7.2.2 Test set-up and description

Figure 7.1 shows the test set-up. An adapterboard is connected over an USB cable to the PC where a terminal program is running. Furthermore, we plugged an FDUZ221 to the phone jack of the adapterboard. The FDUZ221 is connected to the PC over USB. The BTnode is plugged and powered as well in some tests. We want to prove the correct supply and operation of the adapterboard with attached BTnode, despite the latest has no functional purpose here but acts as additional load. We measured the signal values as close as possible to the components. Ground for the PCB is taken from the PC over USB. The values are mean values if not declared otherwise.

7.2.3 Test results

7.2.3.1 Power measurements

In Table 7.1 the different voltage measurement results taken are shown. The measurements are illustrated in Figure 7.2. Table 7.2 shows an overview over the power usage for different operation states. Therefore, we measured the voltage over an 1Ω

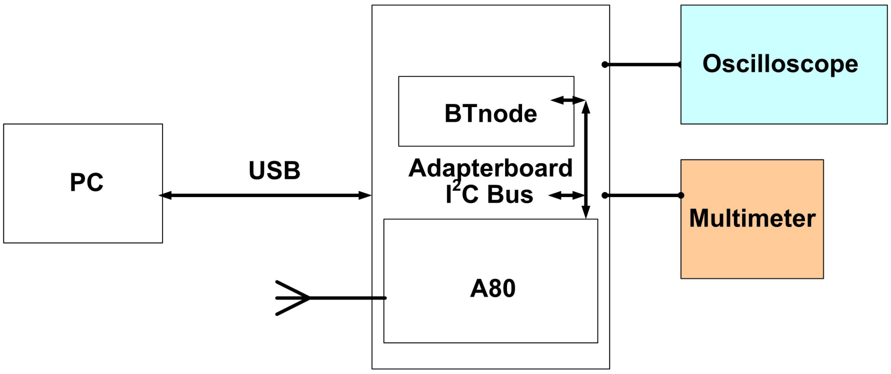


Figure 7.1: Test arrangement for the functional and performance tests of the adapterboard.

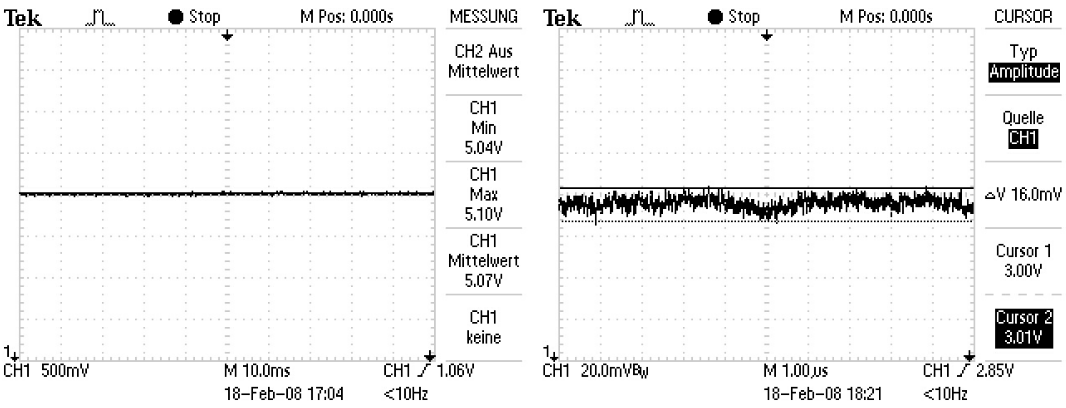
Signal name	TP	Min.	Max.	Result / Comment
V_{Bus}	at C11	5.04 V	5.10 V	5.07 V, **test passed**, ripple ok
V_{cc}	U4.1	3.00 V	3.01 V	3.00 V, **test passed**, ripple ok
V_{Pe}	X5.19	3.4 V	6.0 V	4.88 V, **test passed**
V_{Pe}	X6.18	3.6 V	20.0 V	4.96 V, **test passed**

Table 7.1: Source voltage measurements of the adapterboard.

resistor in series to R1 and computed the corresponding current and power value. Important to know is that the FTDI Chip has a typical current consumption of about 25 mA under normal operation. So we can almost neglect the MSP430 microcontroller's current for the power computation (we fixed V_{Bus} at 5.07 V). The CPU means here the TI MSP430 microcontroller on the adapterboard. Note that at start-up, the BTnode makes a Bluetooth inquiry (up to 198 mA at 3.3 V, see BTnode Rev3 hardware reference [6]) and consumes a high current of about 100 mA. If a USB hub can not deliver enough current, battery supply in parallel is used.

	Adapterboard only	Plus A80	Plus A80, BTnode
CPU, no LED	96.8 mW	109.5 mW	149 mW
CPU, red LED	133.8 mW	152.2 mW	215 mW
CPU, logging A80 red LED, green LED	178 mW	194.6 mW	255 mW
CPU, logging A80 no LED	-	112 mW	180 mW (474 mW)

Table 7.2: Power consumption of the adapterboard with A80 and BTnode plugged.

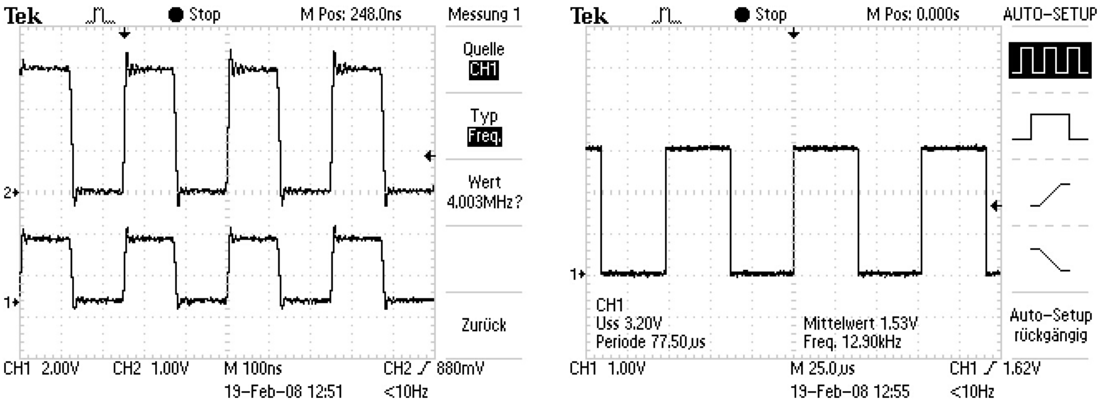


(a) (b)

Figure 7.2: The picture on the left shows the voltage V_{Bus} filtered with C11 and the one on the right illustrates the ripple of the microcontroller's source voltage V_{cc}.

7.2.3.2 Relevant signals

Here, we investigated the most relevant signals on the adapterboard. The adapterboard software employes three different clocks, see Figure 7.3 and Table 7.3. We verified the BSL signals, Figure 7.5, JTAG functionality and I^2C signals integrity. Note that the BSL signals are provided over the NAND-circuit on the PCB, see figure 7.4. Furthermore, we tested the proper USB communication over the FTDI Chip. The drifts for the clocks are ok, since we did not detect any incorrect software behaviour due to clock drifts during the implementation phase.



(a) (b)

Figure 7.3: Picture (a) shows the ACLK (below) and the MCLK. Both have a frequency of 4 MHz. The picture on the right is the SMCLK running at 12.9 kHz that wakes up the Watch Dog periodically.

Signal	**TP**	**Reference value**	**Result**
ACLK	U4.50	4.00 MHz	4.003 MHz, **test passed**, drift ok
SMCLK	TP1, U4.49	12.5 kHz	12.9 kHz, **test passed**, drift ok
MCLK	U4.48	4.00 MHz	4.003 MHz, **test passed**, drift ok

Table 7.3: Clock measurements of the adapterboard. The drift of the SMCLK is ok, since this clock is easily controllable by the firmware and the WDT timer does not need high accuracy.

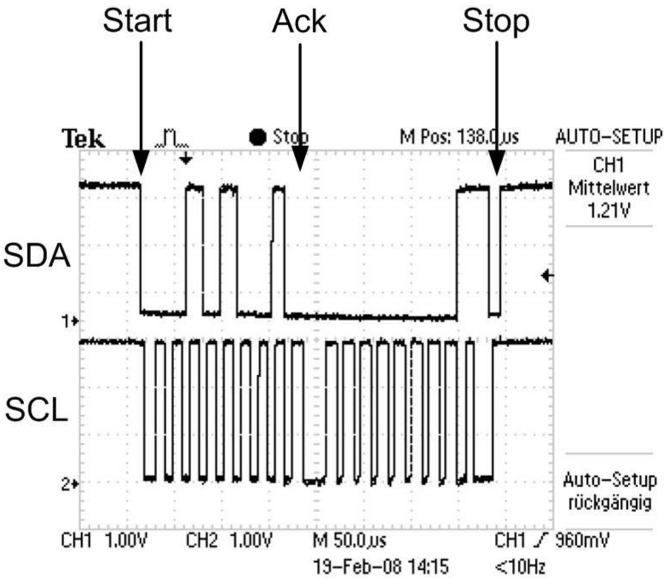


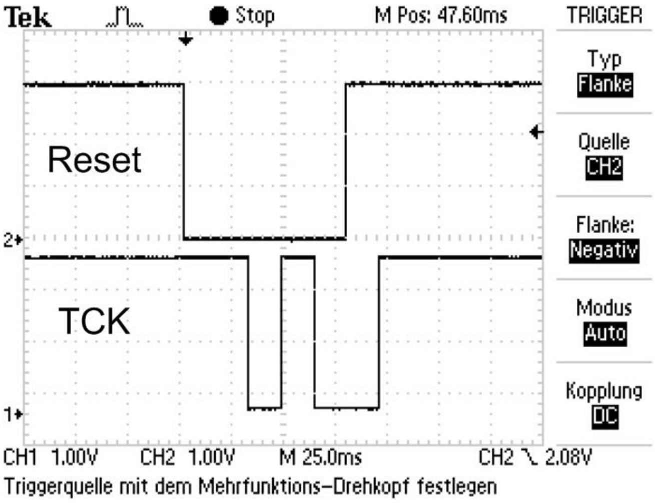
Figure 7.4: The picture shows the signals on the I^2C bus when the A80 is transferring a log message, above the SDA and below the SCL. The bus frequency is 100 kHz and the period therefore 10 μs. The communication works properly. A start condition is fulfilled when a high to low transition on the SDA line when SCL is high occurs. A low to high transition of the SDA signal indicates a stop condition when SCL is high. Data on SDA is stable while SCL is low. For a complete protocol specification, see [17].

7.2.3.3 Overall look and functionality

In this subsection we give an overall look over the functionality test results obtained during operation of the adapterboard, see Table 7.4. For example, the tests showed that when the adapterbaord is configured as I^2C bus slave (per default) and the A80 is disabled, the FDUZ221 is able to upload a code image to the EEPROM. We can state that the I^2C bus is fully functioning from the hardware side.

Functionality	**Test results and comments**
JTAG	Programming over the JTAG connector is working now, whereas the first assembly did not, since a 2.2k resistor is needed between U6.8 and X3.7. **test passed**
DIL switch	To set the input value for the MSP430 controller at S2.5 - S2.8 is possible, dynamic mode changing while the firmware is running could be employed. **test passed**
USB communication to the PC	Ok for baudrates: 9600 kbps, 19200 kbps, 57600 kbps, 115200 kbps, 230400 kps. Not ok for baudrate: 460800 kbps. **test passed**
EEPROM programming over FDUZ221	Programming of EEPROM over FDUZ221 ok **test passed**
Battery supply	Powering of the adapterboard, A80 and BTnode with 4 x AA batteries works properly. **test passed**
Power module	The automatic separation of the battery supply when USB cable is plugged did not work without any failure. If battery switch S1, D5 and R14 instead of C15, R12, R13, D3, D6 and T2 is soldered, the adapterboard behaves correctly. **test (not) passed!**
Driver installation	Un- and replugging to PC, device correct drivers are installed. **test passed**
Power up	When both A80 and BTnode connected to the adapterboard, all devices were powered and the supply did not break up. **test passed**

Table 7.4: Overview of the adapterboard functionality test results.



M Pos: 47.60ms

TRIGGER
Typ: Flanke
Quelle: CH2
Flanke: Negativ
Modus: Auto
Kopplung: DC

CH1 1.00V CH2 1.00V M 25.0ms CH2 ↘ 2.08V

Triggerquelle mit dem Mehrfunktions-Drehkopf festlegen

Figure 7.5: This signal composition of Reset and TCK from the NAND Chip U6 invokes the Bootstrap Loader (BSL) of the MSP430 to upload new code to the controller over the UART interface as described in [24]. The BSL program execution starts when the TCK input of the controller has received a minimum of two negative transitions and is low when Reset rises from low to high level, 3 V here. This works well on the adapterboard.

7.2.4 Conclusions

The test results show that the adapterboard fulfills the requirements concerning correct hardware functionality, except all functions of the power module. We can state that despite this part does not work correctly so far we can use the battery switch by hand instead and the DSN node hardware behaves as intended.
Despite we here documented just the test procedure for one adapterboard we can state that all 12 adapterboard prototypes finally worked fine.

7.3 Adapterboard firmware with plugged A80

7.3.1 Measurement and test equipment

In this tests the following devices and tools were used:

- Device Under Test (DUT) adapterboard Rev0 in Blue Box No. 73
- Siemens A80 with ID 806403F7 plugged to DUT
- FDUZ221 MCLink-USB adapter with USB cable
- terminal program msp430-miniterm
- USB cable for data traffic and power
- USB Host: Fujitsu Siemens PC with Windows XP, SP2
- USB driver for standard FTDI Chip provided by FTDIChip

- USB driver for FDUZ221 provided by SBT
- A80 software: detectornode, Rev. 2446
- A80 software: dummy logger, Rev. 2546
- adapterboard firmware, Rev. 2513
- msp430-bsl program
- Python script serial_client.py

7.3.2 Test set-up and description

In Figure 7.6 the test assembly is drawn. Here a serial connection is set-up over an FDUZ221 in parallel to the main USB connection from the PC. The A80 is connected, enabled and runs a dummy logger software. This application writes as log message ”This is the dummy log message from the A80, average log length” (except the header) with an **incrementing number** to the adapterboard. This string has an above-average log message payload of 80 bytes. In order to set the message intervals, we used the JSON-RPC command ”dummylogger.start[args]” and evaluated the messages sent.
The FDUZ221 is not even necessary for this test. We used it to spy the I^2C bus and to support the verification of the log message transfer. This test allowed us to state the percentage of successful transmitted log messages over USB.
As second test described in this section, we sent a long series of JSON-RPC commands using a Python script to the adapterboard and the A80 radio module respectively and checked whether there is the correct response received or if the DSN node replies at all. Here, the A80 runs the detectornode software. The log level ”ac” of the A80 is set to 0 (a = application and c = communication) which is equal to debug-level and generates much log traffic and is therefore suitable for our test purposes.
With this system under test, we also verified the communication to the EEPROM and the programming of the A80.

7.3.3 Test results

7.3.3.1 Log message transfer

The results obtained from the log message transfer test are shown in Table 7.5. This piece of firmware works correctly and satisfying. We determined that if the log interval of the dummy logger on the A80 is decreased to lower values than 20 ms (50 messages per second!), the adapterboard could still handle this traffic, but the A80 is to slow to produce it! Not all messages could be displayed over USB for those low dutycycles. It is possible that the FDUZ221 influences the proper

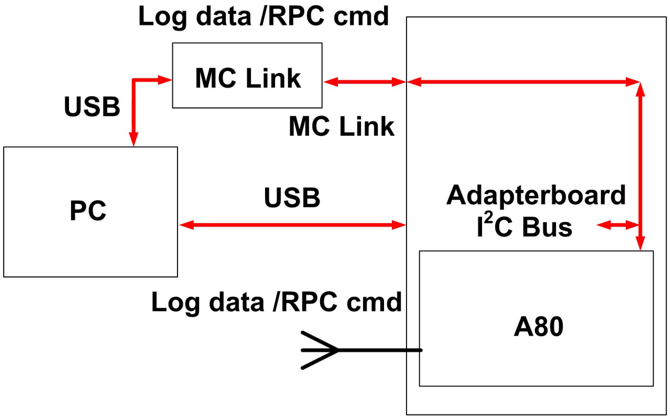
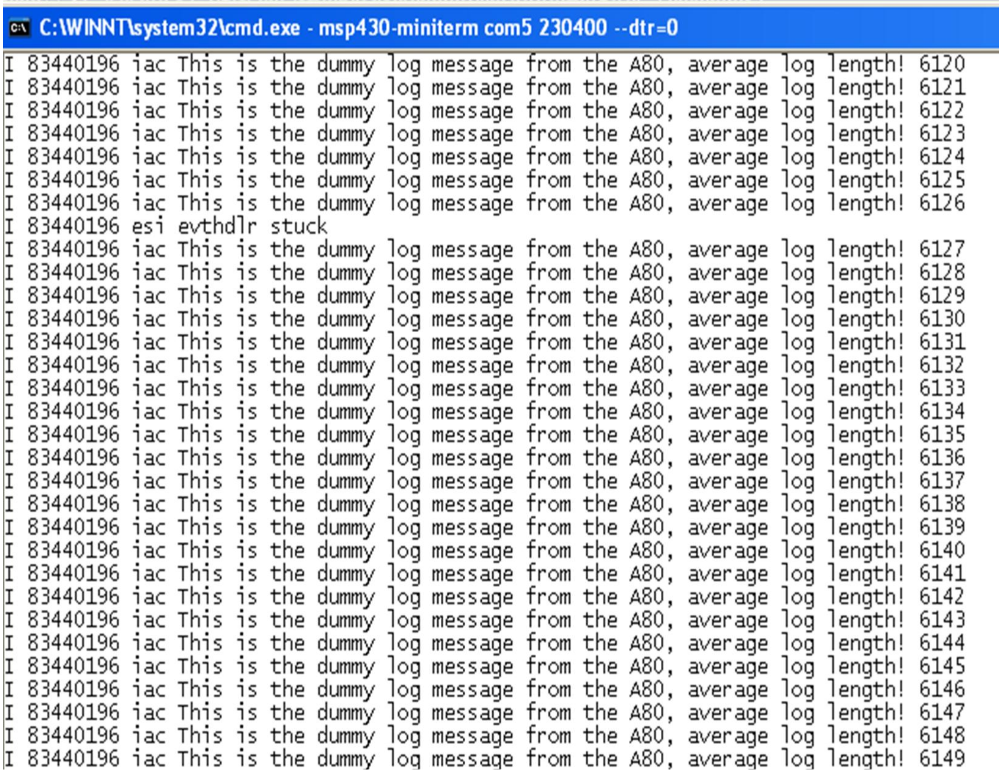


Figure 7.6: Schematic of the test set-up for the firmware evaluation.

communication (acks transfer) between MSP403 and A80. But this requires closer investigations.


C:\WINNT\system32\cmd.exe - msp430-miniterm com5 230400 --dtr=0

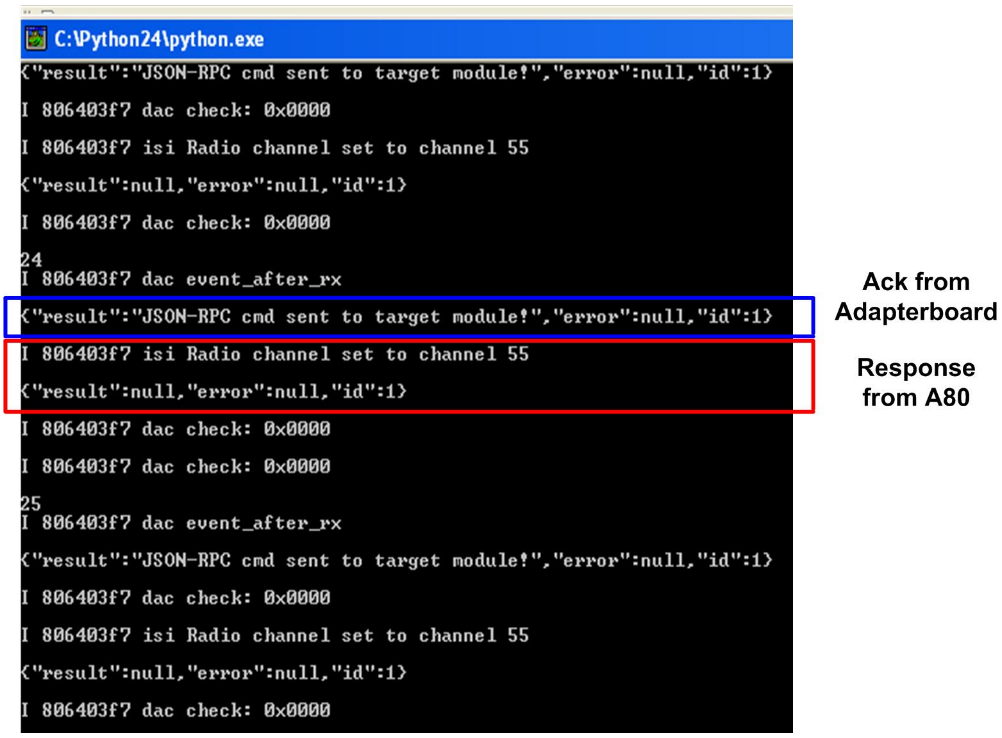
```
I 83440196 iac This is the dummy log message from the A80, average log length! 6120
I 83440196 iac This is the dummy log message from the A80, average log length! 6121
I 83440196 iac This is the dummy log message from the A80, average log length! 6122
I 83440196 iac This is the dummy log message from the A80, average log length! 6123
I 83440196 iac This is the dummy log message from the A80, average log length! 6124
I 83440196 iac This is the dummy log message from the A80, average log length! 6125
I 83440196 iac This is the dummy log message from the A80, average log length! 6126
I 83440196 esi evthdlr stuck
I 83440196 iac This is the dummy log message from the A80, average log length! 6127
I 83440196 iac This is the dummy log message from the A80, average log length! 6128
I 83440196 iac This is the dummy log message from the A80, average log length! 6129
I 83440196 iac This is the dummy log message from the A80, average log length! 6130
I 83440196 iac This is the dummy log message from the A80, average log length! 6131
I 83440196 iac This is the dummy log message from the A80, average log length! 6132
I 83440196 iac This is the dummy log message from the A80, average log length! 6133
I 83440196 iac This is the dummy log message from the A80, average log length! 6134
I 83440196 iac This is the dummy log message from the A80, average log length! 6135
I 83440196 iac This is the dummy log message from the A80, average log length! 6136
I 83440196 iac This is the dummy log message from the A80, average log length! 6137
I 83440196 iac This is the dummy log message from the A80, average log length! 6138
I 83440196 iac This is the dummy log message from the A80, average log length! 6139
I 83440196 iac This is the dummy log message from the A80, average log length! 6140
I 83440196 iac This is the dummy log message from the A80, average log length! 6141
I 83440196 iac This is the dummy log message from the A80, average log length! 6142
I 83440196 iac This is the dummy log message from the A80, average log length! 6143
I 83440196 iac This is the dummy log message from the A80, average log length! 6144
I 83440196 iac This is the dummy log message from the A80, average log length! 6145
I 83440196 iac This is the dummy log message from the A80, average log length! 6146
I 83440196 iac This is the dummy log message from the A80, average log length! 6147
I 83440196 iac This is the dummy log message from the A80, average log length! 6148
I 83440196 iac This is the dummy log message from the A80, average log length! 6149
```

Figure 7.7: Example of the log messages produced by the dummy logger. Here, the A80 received a log interval of 16.7 ms which obviously results in an event handler stuck on the A80.

A80 log interval	# logs sent from A80	# logs received over normal USB	# logs received over FDUZ221	USB Success rate
1 s	1200	1200	1200	100 %
0.5 s	1200	1200	1200	100 %
0.2 s	1200	1200	1198	100 %
20 ms	1200	1197	1197	99.75 %

Table 7.5: Correctly received log messages from the A80 over the adapterboards USB interface. The messages carry about 80 bytes payload. Logs means here log messages.

7.3.3.2 JSON-RPC command transfer



```
{"result":"JSON-RPC cmd sent to target module!","error":null,"id":1}
I 806403f7 dac check: 0x0000
I 806403f7 isi Radio channel set to channel 55
{"result":null,"error":null,"id":1}
I 806403f7 dac check: 0x0000
24
I 806403f7 dac event_after_rx
{"result":"JSON-RPC cmd sent to target module!","error":null,"id":1}
I 806403f7 isi Radio channel set to channel 55
{"result":null,"error":null,"id":1}
I 806403f7 dac check: 0x0000
I 806403f7 dac check: 0x0000
25
I 806403f7 dac event_after_rx
{"result":"JSON-RPC cmd sent to target module!","error":null,"id":1}
I 806403f7 dac check: 0x0000
I 806403f7 isi Radio channel set to channel 55
{"result":null,"error":null,"id":1}
I 806403f7 dac check: 0x0000
```

Figure 7.8: The terminal output on the PC from several JSON-RPC requests is shown. The client sends the command over the adapterboard to trigger the A80 to change the communication channel to 55, which is equal to 869.3875 MHz and was defined as a system parameter in the DSN.

In Table 7.6 the detailed results of the JSON-RPC request tests are described. The adapterboard replies to all JSON-RPC requests with the correct answer, even if every 0.5 seconds a command is sent. We sent a total of 1200 and 500 requests and repeated this three times. The results were always the same: no failures. Looking at the results obtained from sending JSON-RPC commands to the target

Request interval	**# JSON-RPC requests sent (cmd name)**	**# correct responses received**	**Success rate**
0.5 s	1200 (log.disable)	1200	100 %
0.5 s	500 (log.disable, log.enable)	500	100 %

Table 7.6: Results of sending JSON-RPC requests to the adapterboard, that always replied correctly and executed the corresponding method with no failure.

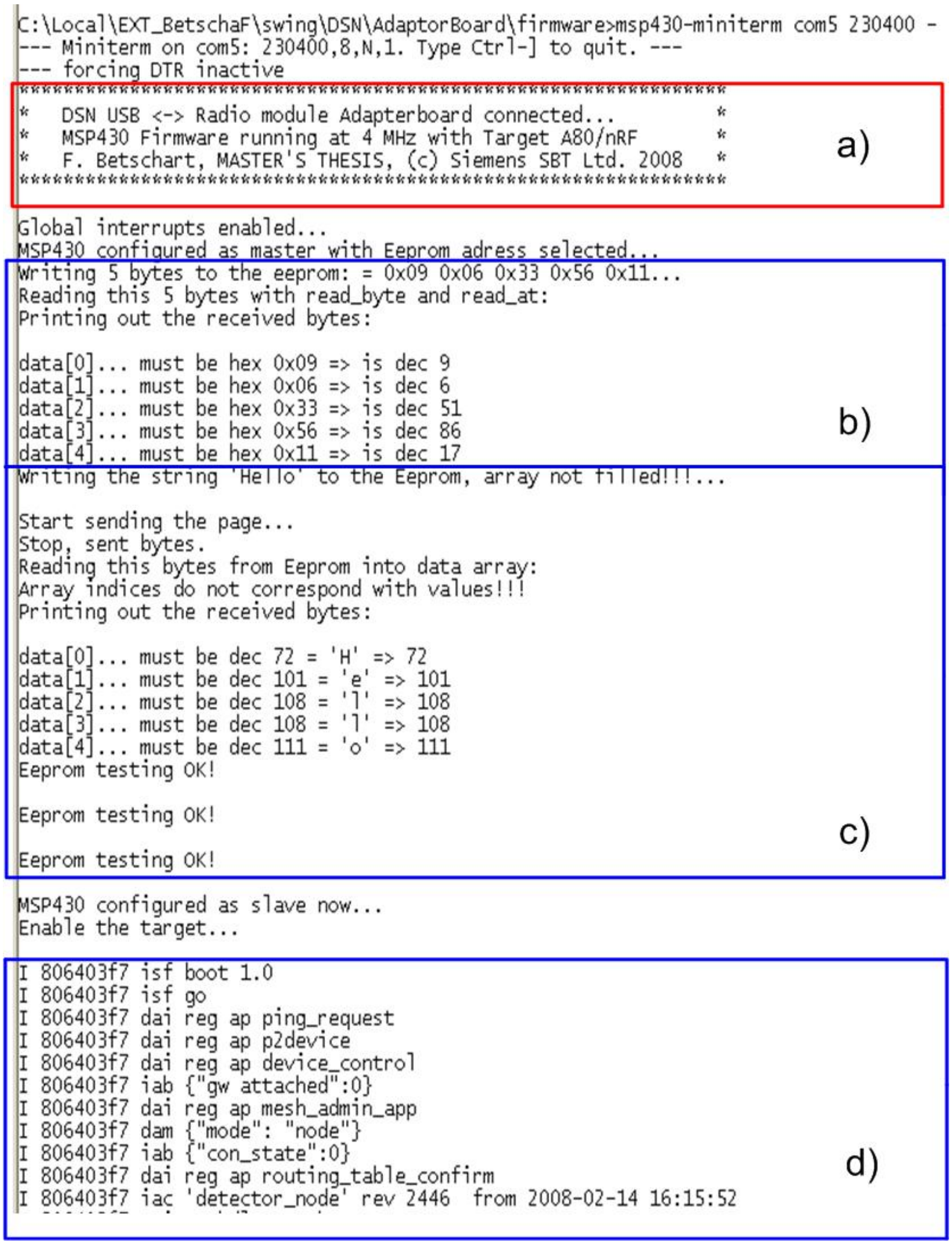
Request interval	**# JSON-RPC requests sent (cmd name to target)**	**# correct responses from A80**	**Success rate**
0.5 s	500 (info.set.chan[55])	498	99.6 %
0.5 s	500 (version)	494	98.8 %

Table 7.7: Results of sending JSON-RPC requests to the A80 over the adapterboard. The A80 replied mostly formal correctly.

in Table 7.7, we can see that the commands are forwarded correctly in most cases from the adapterboard to the A80. The adapterboard sends the corresponding acknoledgement ”JSON-RPC cmd sent to target module!” as well as the response from the target as JSON string to terminal program on the PC. Important is the fact, that the A80 polls the adapterboard every 0.5 seconds, so smaller request intervals would not make sense. It is possible, that the adapterboard captures a log message in between, that is ok. Figure 7.8 illustrates the client output on the PC from one test sequence, where the A80 was invoked to set the radio communication channel to 55 again and again. Twice, after about 100 and 350 repeated commands ”info.set.chan” sent to the A80, an error ocurred. We assume, that some characters were not transmitted correctly over the I^2C bus. The error message was equal to ”no such method”, which is by the way a JSON string too. That explains why not 100 % of the requests were successfull. Table 7.6 and 7.7 list the results.

7.3.3.3 Communication to EEPROM

To execute the EEPROM test routines, we had to uncomment the related code in main.c as already mentioned (uncomment ”test section”). The adapterboard writes bytes, reads bytes and writes a page successfully to the EEPROM when it is configured as I^2C master. We used the functions write_byte, read_byte, read_at and write_page that all showed the correct behaviour. Figure 7.9 presents the serial terminal output on the PC while the adapterboard executes the EEPROM test routines just after the firmware is started. After the test routines, the firmware configures the adapterboard as slave and continues normal operation including reset of target. The results obtained prove that the hardware and pins related to



```
C:\Local\EXT_BetschaF\swing\DSN\AdaptorBoard\firmware>msp430-miniterm com5 230400 -
--- Miniterm on com5: 230400,8,N,1. Type Ctrl-] to quit. ---
--- forcing DTR inactive
******************************************************************
*   DSN USB <-> Radio module Adapterboard connected...          *
*   MSP430 Firmware running at 4 MHz with Target A80/nRF        *
*   F. Betschart, MASTER'S THESIS, (c) Siemens SBT Ltd. 2008    *
******************************************************************

Global interrupts enabled...
MSP430 configured as master with Eeprom adress selected...
Writing 5 bytes to the eeprom: = 0x09 0x06 0x33 0x56 0x11...
Reading this 5 bytes with read_byte and read_at:
Printing out the received bytes:

data[0]... must be hex 0x09 => is dec 9
data[1]... must be hex 0x06 => is dec 6
data[2]... must be hex 0x33 => is dec 51
data[3]... must be hex 0x56 => is dec 86
data[4]... must be hex 0x11 => is dec 17
Writing the string 'Hello' to the Eeprom, array not filled!!!...

Start sending the page...
Stop, sent bytes.
Reading this bytes from Eeprom into data array:
Array indices do not correspond with values!!!
Printing out the received bytes:

data[0]... must be dec 72 = 'H' => 72
data[1]... must be dec 101 = 'e' => 101
data[2]... must be dec 108 = 'l' => 108
data[3]... must be dec 108 = 'l' => 108
data[4]... must be dec 111 = 'o' => 111
Eeprom testing OK!

Eeprom testing OK!

Eeprom testing OK!

MSP430 configured as slave now...
Enable the target...

I 806403f7 isf boot 1.0
I 806403f7 isf go
I 806403f7 dai reg ap ping_request
I 806403f7 dai reg ap p2device
I 806403f7 dai reg ap device_control
I 806403f7 iab {"gw attached":0}
I 806403f7 dai reg ap mesh_admin_app
I 806403f7 dam {"mode": "node"}
I 806403f7 iab {"con_state":0}
I 806403f7 dai reg ap routing_table_confirm
I 806403f7 iac 'detector_node' rev 2446  from 2008-02-14 16:15:52
```

Figure 7.9: Tested functions write_byte, read_byte, read_at and write_page of the adapterboard firmware. The data transmission works properly in both directions. Under a) the start-up sequence of the adapterboard is shown, b) proves the correct writing and reading of several bytes whereas c) verifies the proper page write. The last paragraph d) shows a typical log output after an A80 reset.

the connection between EEPROM and MSP430 are used the right way and that also this communication behaves as intended. Finally, we tested to change the EEPROM address by setting the three least significant bits of the address by the DIL switch pins 2, 3 and 4. Also this test was succesful. The adapterbaord was not able to write anything without changing the address in the firmware as well. Note that the default address is set to 81 in the firmware.

7.3.3.4 A80 flashing

To verify the proper A80 flashing after a reset, we used the FDUZ221 to load an code image to the EEPROM. Afterwards we reset the whole firmware which therefore resets the A80. The A80 was able to load the code from the EEPROM, to flash itself successfully and to display the status information about new software version.

7.3.4 Conclusions

The evaluation of the log message transfer shows that the adapterboard is capable to read the effective number of log messages sent out from the A80 over the I^2C bus and to transmit the data correctly to a client PC over USB. The performance is about 50 log messages per second and node, which corresponds to at least 4 KB/s and is higher than with the BTnode-based DSN, evaluated by Dyer et al. in [4]. In this paper, a rate of three messages per node and second is reached, assumed a 100 % message yield and a 10-node-net.

We can conclude, that the stereo connector who interfaces the I^2C bus is fully functioning as well. An FDUZ221 can gather all the log messages simultaneously to the adapterboard firmware and transmit them to a client PC. Furthermore we can state that the adapterboard receives the JSON-RPC commands and executes them correctly with intervals of at least 0.5 seconds. Also the forwarding of requests to the A80 works properly in most cases. The A80 fails to detect a JSON-RPC command in only about 0.4 % and 1.2 % respectively of randomized JSON-RPC requests, which is an acceptable rate. Bit erros during the I^2C transmission are possible. The EEPROM on the adapterboard is obviously accessible to store and read data from the adapterboard's controller, the FDUZ221 over the stereo link and from the A80.

Finally, we can state that we have a proper functioning adapterboard as DSN node according the required functionality. The adapterboard does not limitate the performance of the Ethernet-based DSN when the number of nodes is increased up to 64.

7.4 NSLU2 software and Ethernet-based test DSN

7.4.1 Measurement and test equipment

In this tests we used the following hardware:

- 8 adapterboards Rev0
- 8 Blue Boxes No. 20, 71, 73, 74, 76, 77, 78 and 84
- 8 Siemens A80 radio modules plugged with IDs 83440152, 806403E4, 806403F7, 8344016E, 83440189, 8064040A, 8064040D and 83440182
- 10 USB 2.0 cables, different lengths from 60 cm up to 4.5 m
- 2 USB 2.0 hubs Maxxtro 7-port, self-powered
- USB host: Fujitsu Siemens PC with Windows XP, SP2
- 1 Linksys NSLU2 device with a installed OS Debian GNU/Linux 4.0
- 1 RJ45 Ethernet cable, not crossed!
- 1 USB stick as external NSLU2 memory

software:

- USB drivers for standard FTDI Chip from FTDIChip on PC
- A80 software: detectornode, Rev. 2446
- A80 software: gatewaynode, Rev. 2436
- A80 software: dummy logger, Rev. 2546
- Putty ssh client on PC
- adapterboard firmware, Rev. 2513
- Python script rpc_client.py, tcpclient.py
- NSLU2 software: Python script tcpserver.py, Version 1.0, Rev. 2545

7.4.2 Test set-up and description

To verify the correct functionality of the NSLU2 in our Ethernet-based DSN, we tested it in two steps. First, we focussed on the basic software features. For this purpose, we built a test system as shown in Figure 7.10 consisting of eight adapterboards with an A80 each equipped and a single NSLU2 device. We attached the NSLU2 to the Ethernet and gave it the static IP address 10.169.26.4. We connected the adapterboards over two USB hubs to the NSLU2. Figure 7.11 illustrates the device's arrangement as it was during this test section.
Eight target nodes build a wireless network with one gateway node (A80, ID 83440152) and seven detectornodes. We accessed the DSN by requesting e.g. "http://10.169.26.4:8080/DSN/logs", see Figure 7.12 for message examples, and

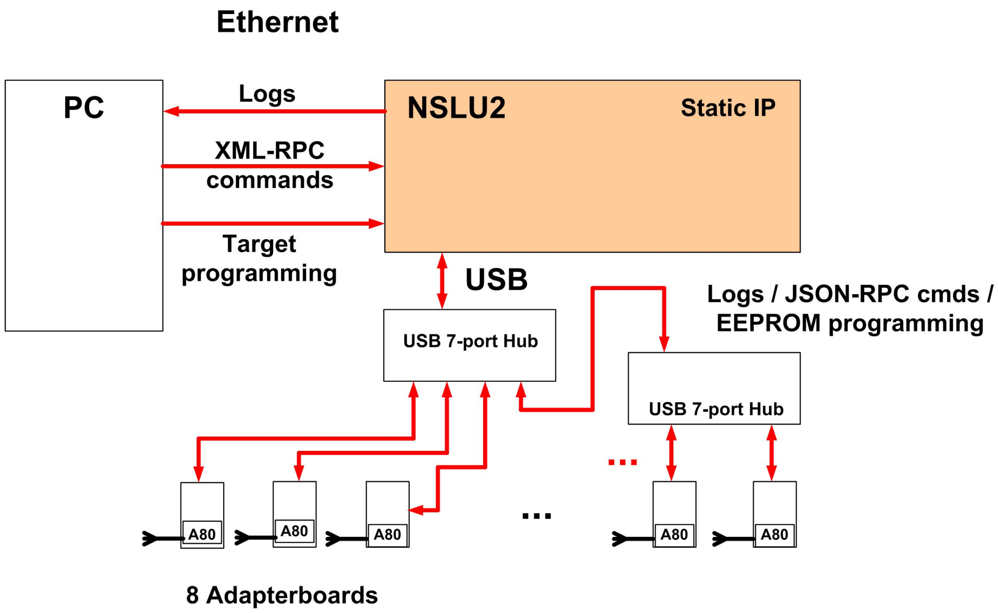


Figure 7.10: Build-up of the 8-node DSN test system. The eight adapterboards with an A80 plugged are connected over USB hubs to a NSLU2 as Ethernet gateway. The client PC interacts with the DSN.

loaded an EEPROM image to the NSLU2 over the web browser. Through XML-RPC, we sent commands with the Python script rpc_client to invoke the corresponding procedures on the NSLU2 such as to reply with a list of the attached nodes or to send JSON-RPC requests to the adapterboard and target respectively. We also tested the correct programming of all the A80 (loading EEPROM image to EEPROM, flashing of A80 and reset). Here, we are just interested in the proper software behaviour and not in the performance.

As a second step we analysed the performance of our 8-node DSN. Therefore, we employed two different software versions for the A80 nodes. Once the A80 acted as detectornode or gateway respectively, once all used the dummy logger software and did nothing than just transmit a dummy log message as described before to the DSN node. We checked, if all messages from all target nodes were transmitted correctly to the PC.

We used the dummy logger software running on the A80[1] to determine the maximum messages that the NSLU2 software can handle.
To make a statement about the target programming performance, we measured the time from sending the programming command to the NSLU2 from the client PC until all A80 finished flashing from the Eeprom. A significant value is the

[1] We always used our Ethernet-based DSN to reprogram the target nodes!

Round Trip Time *RTT* of the command transfers from PC to the adapterboards and A80 respectively. We determined the RTT for several usecases.

7.4.3 Test results

7.4.3.1 Functional evaluation

Table 7.8 gives an overview and the results obtained from the functional tests. We can admit that all the functional software features behave as intended, except the writing of the code image to the adapterboards EEPROM. Mostly just 6 or 7 adapterboards could be programmed simultaneously. The software uses individual programming threads and the JSON-RPC protocol to transfer the data to the adapterboard. A closer investigation of the proper thread execution is needed.

7.4.3.2 Performance evaluation

From Table 7.9 we obtain, that the performance of our 8-node DSN is ok, despite we loose some log messages during operation. A test with dummy logger software running on the A80 did not result in an improved performance versus the normal detector- or gatewaynode software. In decrease of the log message interval rapidly reduced the message yield at the NSLU2. We determined that the highest number of handled messages was about 25 per second for eight nodes. In terms of programming the EEPROM we measured in all our tests a duration from sending the XML-RPC command to the NSLU2 until the A80 finished flashing of about **four minutes for eight nodes serviced by a single NSLU2 using the implemented JSON-RPC protocol**.
We assume that if we increase the DSN size to 64 nodes, we do not even need much longer than this four minutes to program the whole net since we can use eight of our 8-node test nets in parallel.

Any JSON-RPC command request to just one DSN node over the NSLU2 worked in 40 of 40 cases perfectly with a short RTT of about one second. The periodic **broadcast** of JSON-RPC commands to all eight nodes worked in most cases properly, but not that fast as intended. The adapterboard replied correctly each time and the JSON-RPC answers were retransmitted to the PC. The web server behaves as intended. The XML-RPC requests such as "ports_list" or "update_list_disable" perform with a RTT of 0.5 seconds. Table 7.10 shows an overview of the results.

Functionality	**Test results and comments**
Check for plugged DSN nodes, (port name), initialisation of serial port	The detection of the available port names and starting of corresponding object thread works properly. The software start-up lasts between 5 and 40 seconds each time. **test passed**
Request of web pages: ../DSN/list ../DSN/logs ../DSN/test ../upload.html (web application)	The browser displays all the websites, see 7.13, the site containing the log messages is updated as intended, see Figure 7.12 **test passed**
Code image upload to NSLU2	The upload of an new eeprom image works correctly, web server (GET, POST) is ok. **test passed**
Send XML-RPC requests to NSLU2: - ports_list() - send_rpc_cmd(args) - eeprom_write_cmd(args) - update_list_disable() - update_list_enable() (XML-RPC server)	Ok for all. **test passed**
Forwarding of JSON-RPC commands to adapterboard and A80	Ok. **test passed**
Writing code image to 1 adapterboard, A80 reprogramming	Ok, worked for 20 tries with no failure. **test passed**
Writing code image to 8 adapterboards simultaneously	Success rate of 80 % **test (not) passed**

Table 7.8: Overview of the results from the functional test of the NSLU2 Software. For a detailed desription of the command formats see the Readme in the Appendix.

DSN with 7 detectornodes, 1 gatewaynode			
measurement time	300 s	675 s	850 s
# log messages sent from A80	5230	15405	19260
# correctly received from NSLU2	5212	15220	19052
# correctly received over Ethernet	5212	15220	19052
# Success Rate	99.66 %	98.80 %	98.92 %
# log messages/s/node	2.17	2.82	2.83

Table 7.9: Results of the first Ethernet-based DSN log message transfer performance test.

XML-RPC requests to NSLU2				
command name	**# cmds sent**	**# Responses to PC**	**RTT**	**Success rate**
(”ports_list”)	20	20	0.5 s	100 %
(”update_list_disable”)	20	20	0.5 s	100 %
JSON-RPC requests				
command name	**# cmds sent**	**# Responses to PC**	**RTT**	**Success rate**
(”broadcast”, ”system.listMethods”)	20	20	3 s	100 %
(”broadcast”, ”target.reset”)	20	18	2 s	90 %
(”broadcast”, ”forward.command”, ”{”method”: ”info.set.chan”, ”params”: [55], ”id”: 1}”)	20	17	7 s	85 %
(”broadcast”, ”forward.command”, ”{”method”: ”log.level”, ”params”: [”ac”,0], ”id”: 1}”)	20	20	4 s	100 %

Table 7.10: Overview of command transmission performance tests and the RTTs. The JSON-RPC requests are packed in the XML-RPC request ”send_rpc_cmd(args)” that is sent to the NSLU2 and then forwarded to the DSN nodes.

7.4.3.3 Target command RTT computation

We now want to estimate the relative high RTTs of the **broadcasted** JSON-RPC commands to the target nodes without measuring the exact time slices. When we broadcast a JSON-RPC command, the NSLU2 forwards them sequentially awaiting each time the adapteboard to reply. The adapterboard firmware on the other hand fills a buffer that the A80 flushes after having polled for commands. The polling dutycycle of the A80 for new commands from the adapterboard is actually set to 0.5 seconds.
For the computation we make the following definitions:

- command processing time on client PC: t_{PC}
- transmission from PC to NSLU2: t_1
- processing on NSLU2 (for each node): t_{NSLU2}
- transmission from NSLU2 to adapterboard: t_2
- processing on adapterboard (receive, buffer filling): t_{Ab}
- polling interval A80: $t_p = 0.5$ s
- processing time on A80: t_{A80}

$$t_{PC} < t_{NSLU2} < t_{Ab} \approx t_{A80} < 20ms \tag{7.1}$$

$$t_1 << t_2 << t_p = 0.5s = 500ms \tag{7.2}$$

$$t_2 \approx \frac{120B * [8+1+1]bit}{230400baud} \approx \frac{1200bit}{230400kbps} \approx 5.2ms \tag{7.3}$$

The default serial baudrate between NSLU2 and adapterboard equal to 230400 kbps. The upper bound of 20 ms for the processing time is obtained from the adapterboards performance test, compare Table 7.5. With above equations and the neglected t_{PC}, t_{NSLU2} and t_1 we can now estimate an upper bound for the RTT:

$$RTT < 8 * (2 * t_2 + 2 * t_{Ab} + t_p + 2 * t_{A80}) \approx 4.72s \tag{7.4}$$

So we can state, that the worst case RTT is almost 5 seconds, despite the average should be lower. However, this explains the dimension of the RTTs.

7.4.3.4 Results of software performance investigation

In this paragraph we document further investigations made concerning the log message handling performance of the NSLU2 software. From the results obtained from the previous tests we identified the NSLU2 software as the bottleneck in our 8-node-DSN, since the adapterboards are able to send out up to 50 log messages per second. Between the client PC and the NSLU2 the communication works properly and fast over the network, if we consider Table 7.9. Here we used the dummy logger software on the A80 and just observed the console output on the NSLU2 over the Putty ssh client. We can imagine several performance limitation reasons around the NSLU2 software:

- **Using more NSLU2 devices per DSN node**
  This is certainly the easiest solution that would increase the DSN performance but also the most expensive one.

- **Change of reading strategy from serial port in NSLU2 software**
  For the log message transfer the NSLU2 software reads from all available serial ports a single line when the corresponding thread is running. The thread invokes the processing procedure and goes to ready state again. Modifications of code as to read up to 1000 bytes when the thread is running and then to make a data processing byte per byte showed that a higher log message rate can be achieved. The problem is hereby, that the EEPROM writing speed decreases drastically, since the processing of the JSON-RPC responses lasts much longer.

- **Tries with a shorter dummy message**
  Another approach to find out the performance limition causes was to reduce the dummy logger message length on the A80 to 25 bytes. Here, we investigated a higher log message throughput on the NSLU2 (not exactly determined). This yielded to the conclusion, that the OS buffer size is probably not a performance limitating value.

- **Hardware flow control**
  To use hardware flow control for the communication from NSLU2 to the adapterboards would be an alternative. Since we had connected the RTS# (Request To Send) from the FTDI USB-serial converter chip with a microcontroller input on the adapterboard we used it here to check, whether this bit is toggled whenever we loose log messages in the NSLU2. We observed that this handshake signal is never toggled (rise from low to high) for none of the adapterboards (measured at U1.23). Therefore, we assume that no hardware buffer overflow occures in the NSLU2.

- **Disable other activities when reading log messages**
  We could not increase performance when we disabled actions like updating the log message pool or the the web page display in the code.

From above considerations we could not figure out a certain possibility to increase the overall performance.

7.4.4 Conclusions and improvements

From the test results obtained from this section we conclude that our 8-node Ethernet-based test DSN fulfills the functional requirements and performs acceptable. It provides a good DSN for the WSN consisting of A80 radio modules. Sending commands to both adapterboard and A80 as well as remote programming is possible. As intended, the DSN allows us to capture log messages from the A80 over the wired network from a client PC (or later the DSN Server) in the backbone.
What has to be improved is the performance, since the actual configuration with a single NSLU2 and eight adapterboards is not able to handle more data than a BTnode-based DSN described in [4]. There, a rate of three messages per node and second is achieved, with a 100 % message yield and 10 nodes. But they took the results when each node sent 100 messages, we measured for longer period what is certainly more significant. No statements are made in [4] about the programming time of the target nodes.
The bottleneck in our system is the NSLU2 software. Either the number of DSN nodes per NSLU2 or the NSLU2 software has to be adapted to improve the performance especially when the network size is increased up to 64 or 128 nodes in the future.

7.5 Backwards compatibility to BTnode-based DSN

7.5.1 Measurement and test equipment

In this tests we used the following devices and tools:

hardware:
- adapterboard Rev0 in Blue Box No. 84 with four metal screws
- adapterboard Rev0 in Blue Box No. 75 with four **plastic** screws
- Siemens A80 radio module plugged with ID 83440196
- 2 USB 2.0 cables
- USB Host: Fujitsu Siemens PC with Windows XP, SP 2
- Laptop Fujitsu Siemens with Windows Professional Version 2002, SP 2
- BTnode Rev3.24, 0.50103, 20/06, MAC: 00043F0001B5

software:

# log messages sent from A80	# logs USB	# logs FDUZ221	# logs BTnode	Success rate
10200 (30/s)	10200	10200	10200	100 %

Table 7.11: Test results when the adapterboard transmits log messages.

- USB drivers for standard FTDI Chip provided by FTDIChip
- A80 software: detectornode, (Rev. 2446)
- A80 software: dummy logger, (Rev. 2546)
- adapterboard firmware (Rev. 2513)
- DSNAnalyzer, DSNServer
- 2 x terminal program msp430-miniterm

7.5.2 Test set-up and description

This test concludes the evalution of the Ethernet-based DSN. Here, the crucial question is if a BTnode plugged the adapterboard is able to send the log messages out over Bluetooth to the GUI node and the DSN Server respectively. We therefore attached an A80 to the adapterboard and a BTnode as well. The A80 ran the dummy logger software so that we could count the messages it produced. We tried to capture the log messages over three independent channels, namely the adapterboards USB interface, the FDUZ221 and the BTnode. The test assemby is drawn in Figure 7.15.

We took two different adapterboards to investigate the influence of the plastic screws who fix the PCB in the Blue Box. We did neither test to send JSON-RPC commands over the BTnode to the target node nor EEPROM programming because that does not work here anyway. The adapterboard firmware needs to be adapted. The firmware sends a zero byte out to the I^2C when the target polls and that is exactly was the target always receives here.

7.5.3 Test results

The BTnode behaves as intended. It is possible to gather log messages from the A80 over all three channels, see Table 7.11. We found out, that we have to switch the battery power on additionally to make sure proper BTnode operation. Otherwise, not enough current is delivered only over USB. If so, the BTnode did not always appear as DSN node in the DSNAnalyzer.

We could not determine a significant difference between plastic- and metall-screw-adapterboard's BTnode behaviour.

7.5.4 Conclusions

The results prove that it is possible to run both DSNs (Ethernet and Bluetooth) simultaneously at least for log message capturing and we can assume that the adapterboard developed within this work supports the use of the BTnode-based DSN.

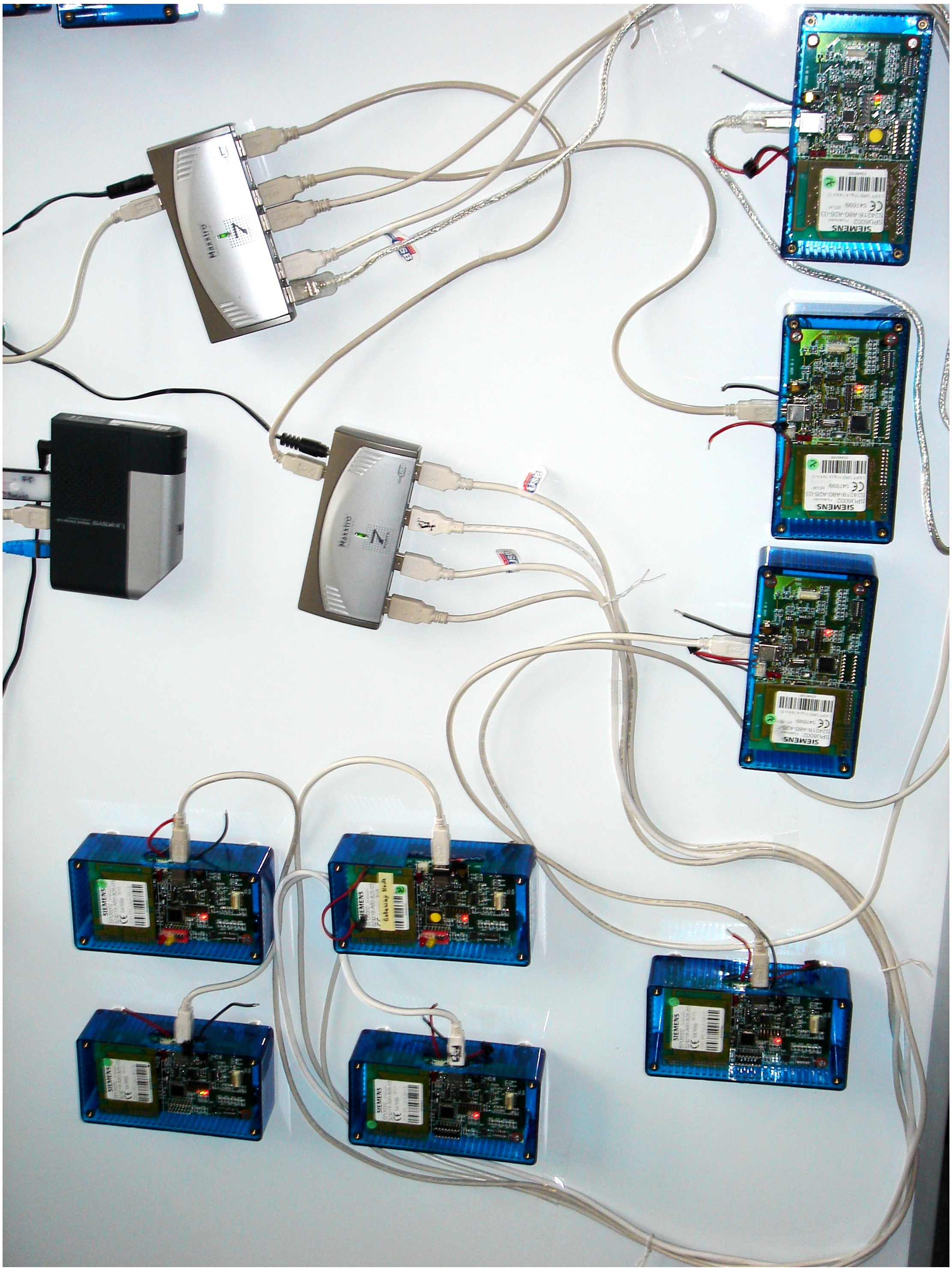
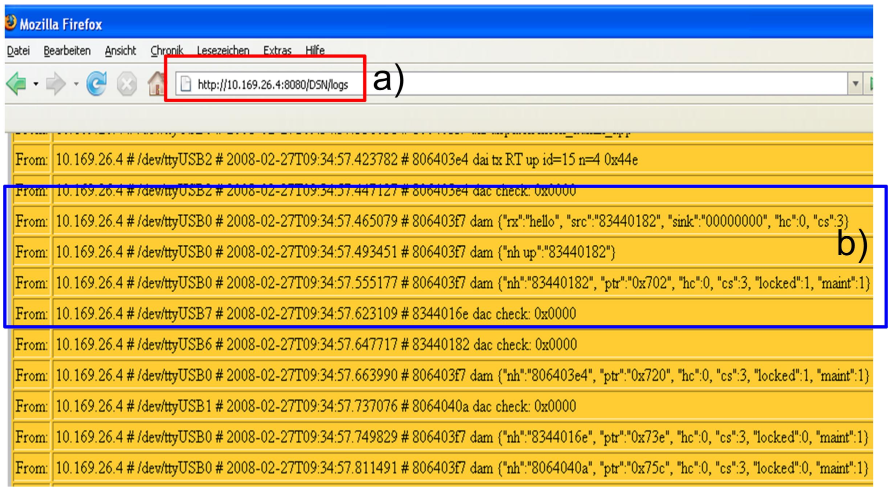
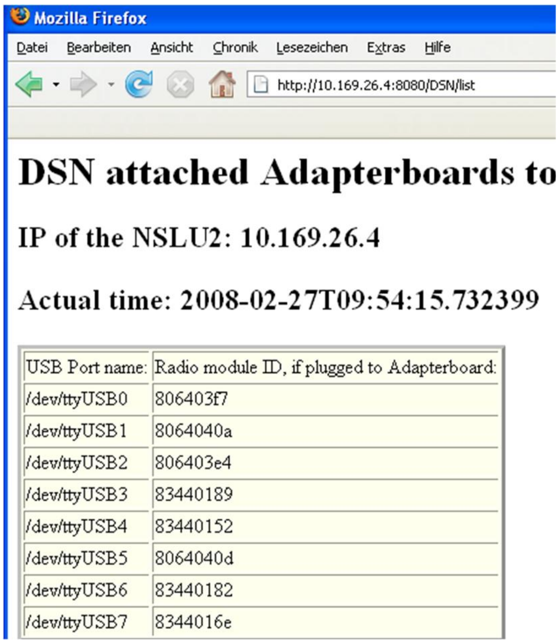


Figure 7.11: A complete 8-node test DSN built within this work.


Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://10.169.26.4:8080/DSN/logs
a)
b)

```
From: 10.169.26.4 # /dev/ttyUSB2 # 2008-02-27T09:34:57.423782 # 806403e4 dai tx RT up id=15 n=4 0x44e
From: 10.169.26.4 # /dev/ttyUSB2 # 2008-02-27T09:34:57.447127 # 806403e4 dac check: 0x0000
From: 10.169.26.4 # /dev/ttyUSB0 # 2008-02-27T09:34:57.465079 # 806403f7 dam {"rx":"hello", "src":"83440182", "sink":"00000000", "hc":0, "cs":3}
From: 10.169.26.4 # /dev/ttyUSB0 # 2008-02-27T09:34:57.493451 # 806403f7 dam {"nh up":"83440182"}
From: 10.169.26.4 # /dev/ttyUSB0 # 2008-02-27T09:34:57.555177 # 806403f7 dam {"nh":"83440182", "ptr":"0x702", "hc":0, "cs":3, "locked":1, "maint":1}
From: 10.169.26.4 # /dev/ttyUSB7 # 2008-02-27T09:34:57.623109 # 8344016e dac check: 0x0000
From: 10.169.26.4 # /dev/ttyUSB6 # 2008-02-27T09:34:57.647717 # 83440182 dac check: 0x0000
From: 10.169.26.4 # /dev/ttyUSB0 # 2008-02-27T09:34:57.663990 # 806403f7 dam {"nh":"806403e4", "ptr":"0x720", "hc":0, "cs":3, "locked":1, "maint":1}
From: 10.169.26.4 # /dev/ttyUSB1 # 2008-02-27T09:34:57.737076 # 8064040a dac check: 0x0000
From: 10.169.26.4 # /dev/ttyUSB0 # 2008-02-27T09:34:57.749829 # 806403f7 dam {"nh":"8344016e", "ptr":"0x73e", "hc":0, "cs":3, "locked":0, "maint":1}
From: 10.169.26.4 # /dev/ttyUSB0 # 2008-02-27T09:34:57.811491 # 806403f7 dam {"nh":"8064040a", "ptr":"0x75c", "hc":0, "cs":3, "locked":0, "maint":1}
```

Figure 7.12: An extract of the web page ../DSN/logs from the Firefox web browser. It displays the 1000 latest log messages sent out from the A80 nodes. Under a) one can see the URL and b) shows the format with IP address, port name, time stamp and processed message.


Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://10.169.26.4:8080/DSN/list

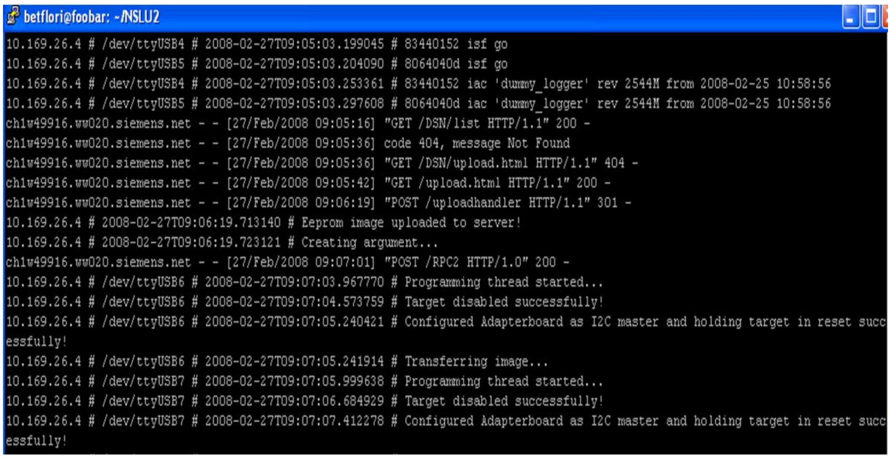
DSN attached Adapterboards to

IP of the NSLU2: 10.169.26.4

Actual time: 2008-02-27T09:54:15.732399

USB Port name:	Radio module ID, if plugged to Adapterboard:
/dev/ttyUSB0	806403f7
/dev/ttyUSB1	8064040a
/dev/ttyUSB2	806403e4
/dev/ttyUSB3	83440189
/dev/ttyUSB4	83440152
/dev/ttyUSB5	8064040d
/dev/ttyUSB6	83440182
/dev/ttyUSB7	8344016e

Figure 7.13: The picture shows /DSN/list that informs the user about the DSN configuration. The first time an A80 sends a log message, this page is updated with the new serial port name and the corresponding A80 module ID.



```
10.169.26.4 # /dev/ttyUSB4 # 2008-02-27T09:05:03.199045 # 83440152 isf go
10.169.26.4 # /dev/ttyUSB5 # 2008-02-27T09:05:03.204090 # 8064040d isf go
10.169.26.4 # /dev/ttyUSB4 # 2008-02-27T09:05:03.253361 # 83440152 iac 'dummy_logger' rev 2544M from 2008-02-25 10:58:56
10.169.26.4 # /dev/ttyUSB5 # 2008-02-27T09:05:03.297608 # 8064040d iac 'dummy_logger' rev 2544M from 2008-02-25 10:58:56
ch1w49916.ww020.siemens.net - - [27/Feb/2008 09:05:16] "GET /DSN/list HTTP/1.1" 200 -
ch1w49916.ww020.siemens.net - - [27/Feb/2008 09:05:36] code 404, message Not Found
ch1w49916.ww020.siemens.net - - [27/Feb/2008 09:05:36] "GET /DSN/upload.html HTTP/1.1" 404 -
ch1w49916.ww020.siemens.net - - [27/Feb/2008 09:05:42] "GET /upload.html HTTP/1.1" 200 -
ch1w49916.ww020.siemens.net - - [27/Feb/2008 09:06:19] "POST /uploadhandler HTTP/1.1" 301 -
10.169.26.4 # 2008-02-27T09:06:19.713140 # Eeprom image uploaded to server!
10.169.26.4 # 2008-02-27T09:06:19.723121 # Creating argument...
ch1w49916.ww020.siemens.net - - [27/Feb/2008 09:07:01] "POST /RPC2 HTTP/1.0" 200 -
10.169.26.4 # /dev/ttyUSB6 # 2008-02-27T09:07:03.967770 # Programming thread started...
10.169.26.4 # /dev/ttyUSB6 # 2008-02-27T09:07:04.573759 # Target disabled successfully!
10.169.26.4 # /dev/ttyUSB6 # 2008-02-27T09:07:05.240421 # Configured Adapterboard as I2C master and holding target in reset succ
essfully!
10.169.26.4 # /dev/ttyUSB6 # 2008-02-27T09:07:05.241914 # Transferring image...
10.169.26.4 # /dev/ttyUSB7 # 2008-02-27T09:07:05.999638 # Programming thread started...
10.169.26.4 # /dev/ttyUSB7 # 2008-02-27T09:07:06.684929 # Target disabled successfully!
10.169.26.4 # /dev/ttyUSB7 # 2008-02-27T09:07:07.412278 # Configured Adapterboard as I2C master and holding target in reset succ
essfully!
```

Figure 7.14: Typical output on the NSLU2 console. The A80 runs the dummy logger software that waits for the start command after start-up. Then the new image is uploaded to the NSLU2 over the web browser and the user starts the programming procedure.

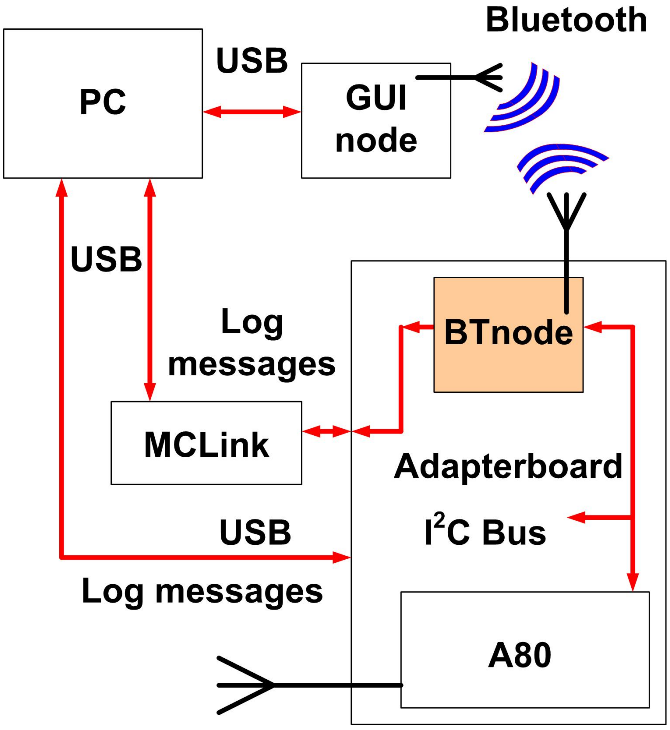


Figure 7.15: Test set-up of the backwards compatibility test.

Chapter 8

Conclusions and future work

We summarise here the achievements of the work presented in this thesis and make a conclusion based on the evaluation presented in the previous chapter. Furthermore, we give an outlook to future work.

8.1 Summary

The scope of this work was to develop a Deployment Support Network based on a wired approach. After having studied and evaluated different design variants, we decided to build a hybrid system consisting of Ethernet and USB for communication. To connect the target nodes over I^2C and to provide an USB interface to the backbone, we developed a new adapterboard including schematics, layout and assembling of twelve prototypes. Afterwards, we wrote firmware in C for the adapterboard. According to the required functionality, the firmware supports:

- Log message reading from the target nodes over I^2C bus and sending them out over USB.
- Receiving and execution of JSON-RPC commands including forwarding to the target node.
- Writing a code image for the target node to an external EEPROM.

The adapterboards are connected over USB hubs to a so called NSLU2 to complete the DSN. This commercial device acts as gateway from several USB ports to Ethernet. We implemented software running in this NSLU2. The software provides basically:

- Receiving and processing XML-RPC commands from a remote PC or the DSN Server.

- Extracting JSON-RPC commands from XML-RPC requests to forward to the addressed DSN nodes.
- Capturing log messages from all the USB-interfaced DSN nodes and sending them over Ethernet to a network client. The NSLU2 software adds the log messages a timestamp, the serial port name and the IP address. The main log message processing is done in the backbone.
- The user can upload a code image over the web browser and start the image transmission to one or several adapterboards remotely.

8.2 Conclusions

From the evaluation of a 8-node DSN with a single NSLU2 we can obtain that the Ethernet-based DSN is able to monitor the target network and that the implemented software features on the adapterboard as well as on the NSLU2 work properly. Concerning the performance we have to admit that the Ethernet-based DSN delivers not higher log message data rates than the BTnode-based one so far, despite a single adapterboard could send about 50 messages per second. The system performance can be improved, since the reliability is excellent and the NSLU2 software is not optimised.

If desired, the BTnodes can be employed with the new adapterboards to run the Bluetooth DSN supplied from batteries. Some modifications in the adapterboard firmware are nedeed in that case.

8.3 Design approach review

We go out from the **Design Approach Evaluation Table** in Chapter 3. There we judged the different design approaches for an Ethernet-based DSN.

From the cost effort's point of view, we could employ the fully functioning TCP/IP stack to communicate over the network by using the NSLU2's Linux OS stack. The only drawback is that we use one more device between DSN node and the backbone with the DSN Server, but this seems not to have bad influence on the portability of the system so far. Furthermore, we managed to finish off within 26 weeks having a running system.

Despite we use no PoE, only one cable to the DSN node for power supply is needed. Additionally battery power can be used. Concerning the components cost we can say that we estimated 198 CHFr. per node and we finally had to pay 216 CHFr. So this is a reasonable price compared to the other alternatives.

The adapterboard can be reused with the BTnode-DSN as well and even for other

applications, where USB is needed. The NSLU2 and the hubs are general purpose devices, and could be employed anywhere, so usability is guaranteed here.
We assumed that we would have at least no data traffic bottleneck at the NSLU2 as part of our DSN and that the performance is limited by the USB-adapterboards. By now, it seems the other way round. But neither the NSLU2 software is optimised nor the number of DSN nodes per NSLU2. **So not to connect each DSN node to Ethernet was the right design decision**.
This approach provides a lot of optimisation possibilites whereas the most of the others do not. Furthermore, we minimised the risks and difficulties that could have occurred during this work. We can state that despite we had several hard- and software implementation problems to solve we finally managed to get an Ethernet-based DSN running.

8.4 Outlook

Here we propose the next steps to take in order to run a 64-node DSN:

- Investigation of the NSLU2 software and performance optimisation. In addition, a proper software interface for the DSN Server has to be defined based on the already implemented software.
- In order to increase network size, several NSLU2s with attached DSN nodes should operate together for first test reasons.
- The DSN Server has to be redesigned for Ethernet connectivity and to address the adapterboards.

Bibliography

[1] Makoto Suzuki, Shunsuke Saruwatari, Narito Kurata, Hiroyuki Morikawa: *A High-Density Earthquake Monitoring System Using Wireless Sensor Networks.* In SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems, pages 373–374, Sidney, Australia, 2007.

[2] Siemens: *Application for funding (Safety Critical Sensor Networks for Building Applications).* KTI/CTI Application Document, Siemens Internal, Switzerland, 2006

[3] Konrad Lorincz, David J. Malan, Thaddeus R.F. Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoffrey Mainland, Matt Welsh: *Sensor Networks for Emergency Response: Challenges and Opportunities.* IEEE Pervasive Computing. October to December 2004.

[4] Matthias Dyer, Jan Beutel, Lothar Thiele, Thomas Kalt, Patrice Oehen, Kevin Martin, Philipp Blum: *Deployment support network - a toolkit for the development of WSNs.* In Proceedings of the 4th European Conference on Wireless Sensor Networks, pages 195-211. Springer, Berlin, Germany, January 2007.

[5] Bluetooth, `http://www.bluetooth.com`

[6] *Btnode, A Distributed Environment for Prototyping Ad Hoc Networks.* Rev3 - product brief, `http://www.btnode.ethz.ch`, March 2006.

[7] Thomas Kalt: *Online sensor network analysis tool.* Master thesis, Swiss Federal Institute of Technology (ETH) Zurich, Switzerland, 2006.

[8] Patrice Oehen: *DSNAnalyzer: Backend for the deployment support network.* Master thesis, Swiss Federal Institute of Technology (ETH) Zurich, Switzerland, 2006.

[9] Severin Hafner: *Distributed Spy-Software Tool WSNSpy.* Master thesis, Swiss Federal Institute of Technology (ETH) Zurich, Switzerland, 2007.

[10] Adam Wolisz, Vlado Handziski, Andreas Köpke, Andreas Willig: *TWIST: A Scalable and Reconfigurable Wireless Sensor Network Testbed for Indoor Deployments.* In Proc. of the 2nd Intl. Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality, (RealMAN 2006), Florence, Italy, May 2006.

[11] Telos - Ultra low power IEEE 802.15.4 compliant wireless sensor module, `http://www.ece.osu.edu/~bibyk/ee582/telosMote.pdf`

[12] Tmote Sky `http://www.eecs.harvard.edu`

[13] Geoffrey Werner-Allen, Patrick Swieskowski, Matt Welsh: *MoteLab: A Wireless Sensor Network Testbed.* In Proceedings of the 4th international symposium on Information processing in sensor networks, Los Angeles, California, USA, 2005.

[14] Ki-Young Jang, Marcos Vieira, Sumit Rangwala, Omprakash Gnawali, Ramesh Govindan: *Tutornet: A Tiered Wireless Sensor Network Testbed,* `http://enl.usc.edu/projects/tutornet`

[15] sMote, `http://www.millennium.berkeley.edu/sensornets`

[16] Mica Motes, `http://www.xbow.com`

[17] I^2C bus spec, `http://www.educypedia.be/electronics/I2C.htm`

[18] NSLU2 datasheet, `http://www.linksys.com`

[19] `http://www.cyrius.com/debian/nslu2`

[20] `http://www.nslu2-linux.org`

[21] Power over Ethernet, `http://www.poweroverethernet.com`

[22] ARM, `http://www.arm.com/products`

[23] uIP, `http://www.rowley.co.uk/msp430/uip.htm`

[24] MSP430 BSL,
`http://focus.ti.com/lit/an/slaa089d/slaa089d.pdf`

[25] JTAG, `http://www.embedded.com/story/OEG20021028S0049`

[26] JSON-RPC, `http://json-rpc.org/wiki/specification`

[27] TinyOS, `http://www.tinyos.net`

[28] XML-RPC, `http://en.wikipedia.org/wiki/XML-RPC`

[29] HTTP, `http://www.w3.org/Protocols`

[30] C, `www.cprogramming.com`

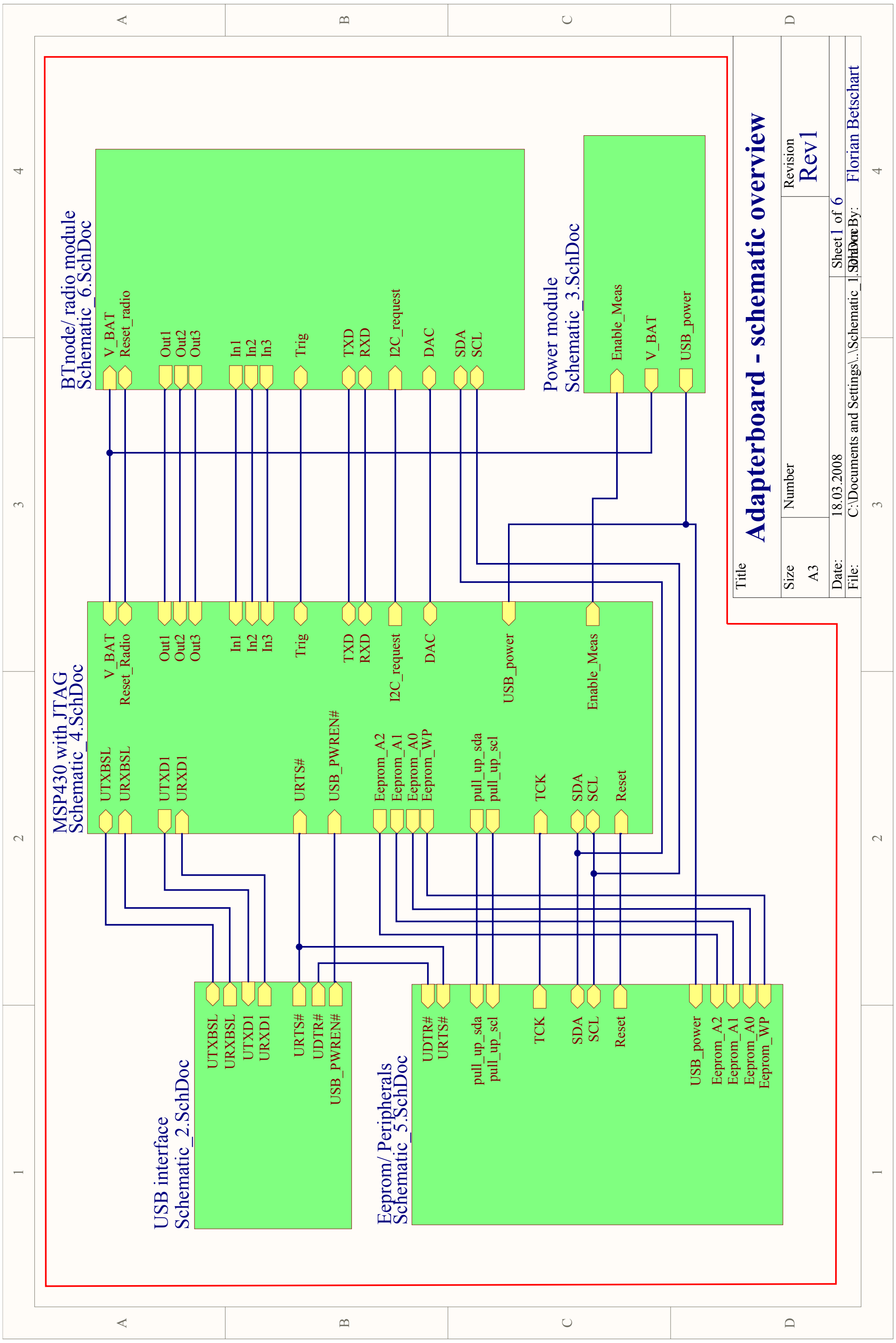
[31] MSPGCC Toolchain, `http://mspgcc.sourceforge.net`

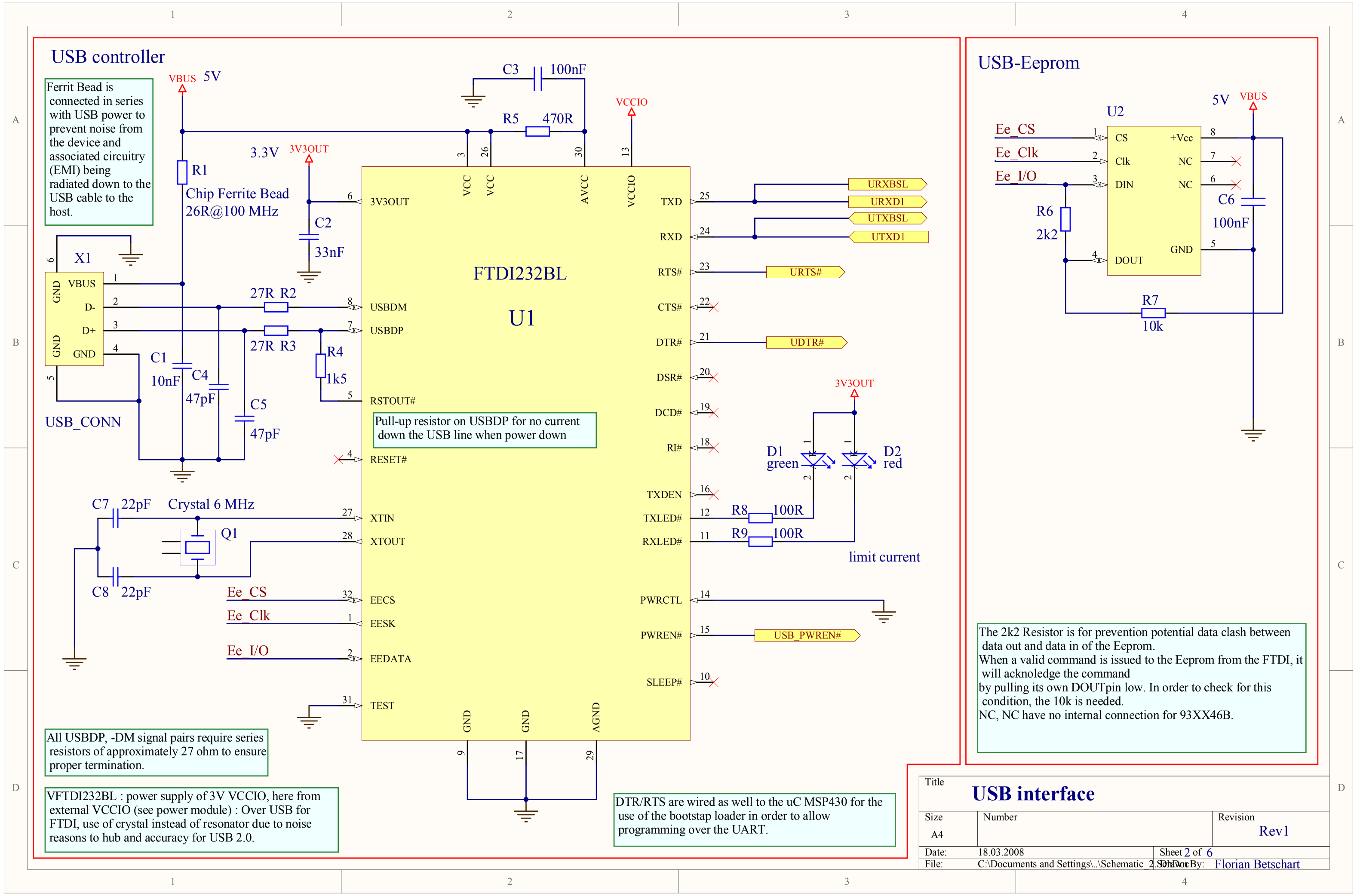
[32] TI MSP430 manual,
`http://focus.ti.com/lit/ug/slau049f/slau049f.pdf`

[33] Python, `http://www.python.org`

Appendix A

Schematics and Footprints


A
B
C
D
1
2
3
4
BTnode/ radio module
Schematic_6.SchDoc
V_BAT
Reset_radio
Out1
Out2
Out3
In1
In2
In3
Trig
TXD
RXD
I2C_request
DAC
SDA
SCL
Power module
Schematic_3.SchDoc
Enable_Meas
V_BAT
USB_power
MSP430 with JTAG
Schematic_4.SchDoc
V_BAT
Reset_Radio
Out1
Out2
Out3
In1
In2
In3
Trig
TXD
RXD
I2C_request
DAC
USB_power
Enable_Meas
UTXBSL
URXBSL
UTXD1
URXD1
URTS#
USB_PWREN#
Eeprom_A2
Eeprom_A1
Eeprom_A0
Eeprom_WP
pull_up_sda
pull_up_scl
TCK
SDA
SCL
Reset
USB interface
Schematic_2.SchDoc
UTXBSL
URXBSL
UTXD1
URXD1
URTS#
UDTR#
USB_PWREN#
Eeprom/ Peripherals
Schematic_5.SchDoc
UDTR#
URTS#
pull_up_sda
pull_up_scl
TCK
SDA
SCL
Reset
USB_power
Eeprom_A2
Eeprom_A1
Eeprom_A0
Eeprom_WP
Title
Adapterboard - schematic overview
Size
A3
Number
Revision
Rev1
Date: 18.03.2008
Sheet 1 of 6
File: C:\Documents and Settings\..\Schematic_1.SchDoc
Drawn By: Florian Betschart



Ferrit Bead is connected in series with USB power to prevent noise from the device and associated circuitry (EMI) being radiated down to the USB cable to the host.

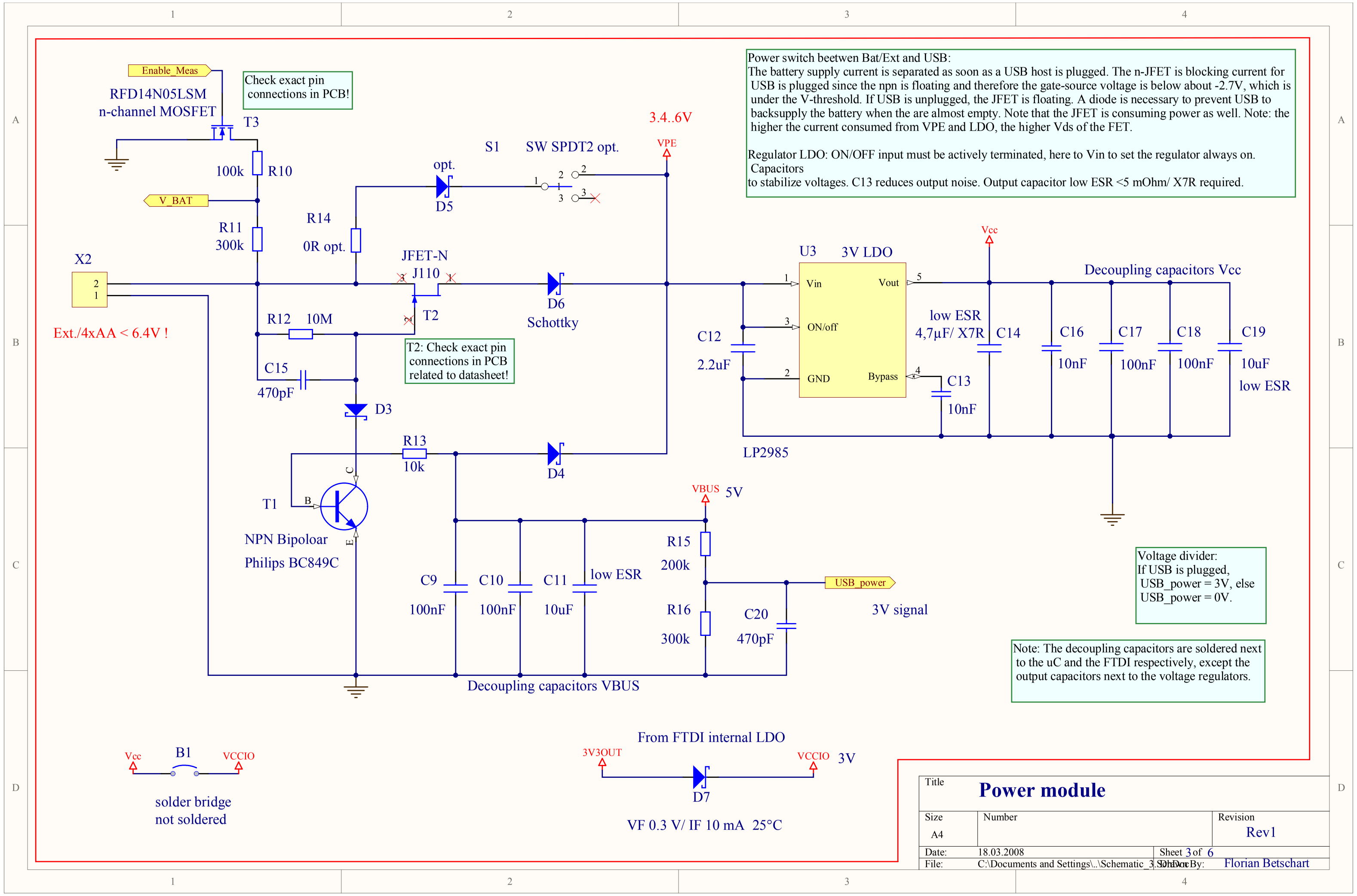
Pull-up resistor on USBDP for no current down the USB line when power down

All USBDP, -DM signal pairs require series resistors of approximately 27 ohm to ensure proper termination.

VFTDI232BL : power supply of 3V VCCIO, here from external VCCIO (see power module) : Over USB for FTDI, use of crystal instead of resonator due to noise reasons to hub and accuracy for USB 2.0.

DTR/RTS are wired as well to the uC MSP430 for the use of the bootstap loader in order to allow programming over the UART.

The 2k2 Resistor is for prevention potential data clash between data out and data in of the Eeprom.
When a valid command is issued to the Eeprom from the FTDI, it will acknoledge the command
by pulling its own DOUTpin low. In order to check for this condition, the 10k is needed.
NC, NC have no internal connection for 93XX46B.



Check exact pin connections in PCB!

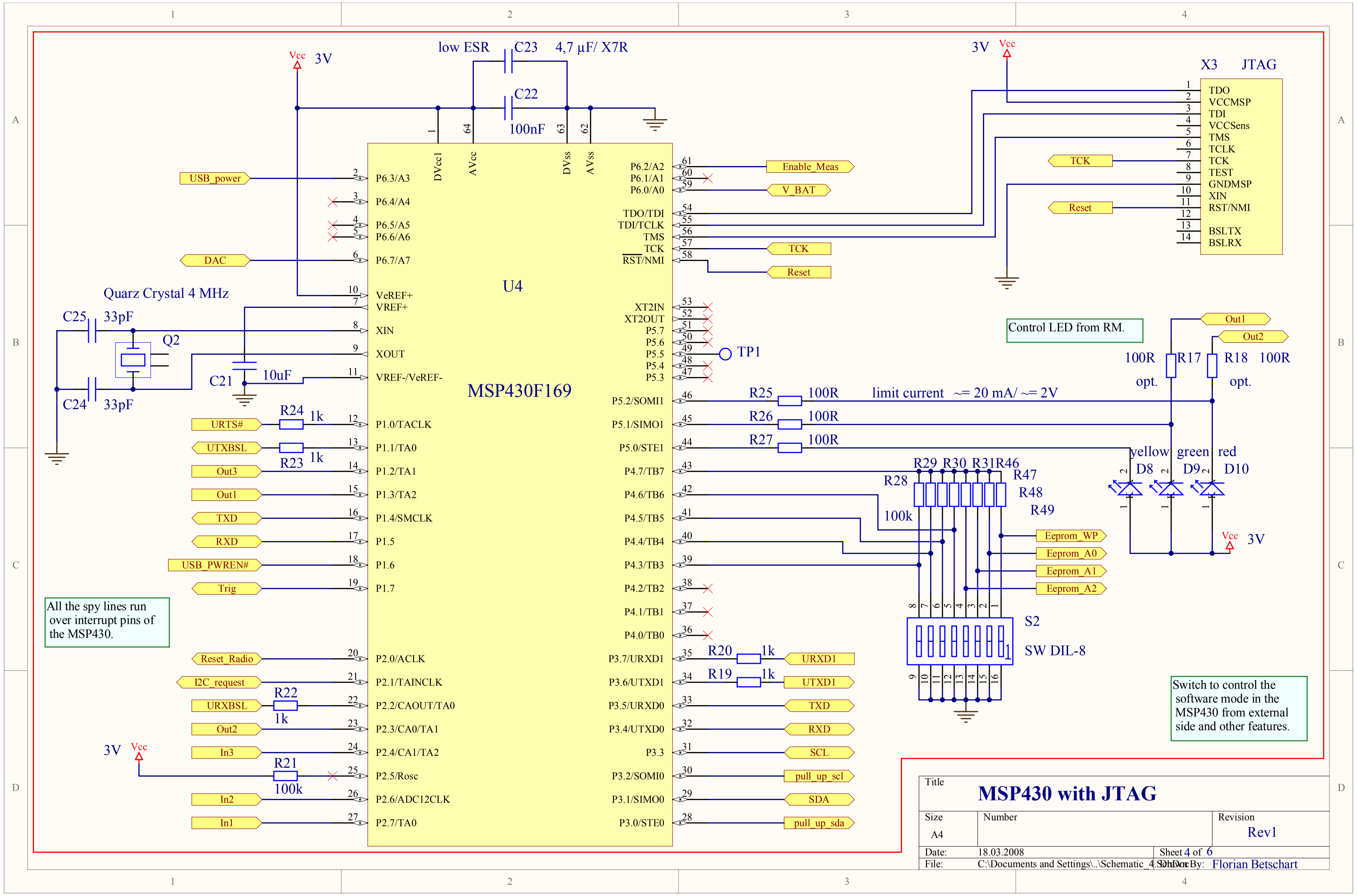
T2: Check exact pin connections in PCB related to datasheet!

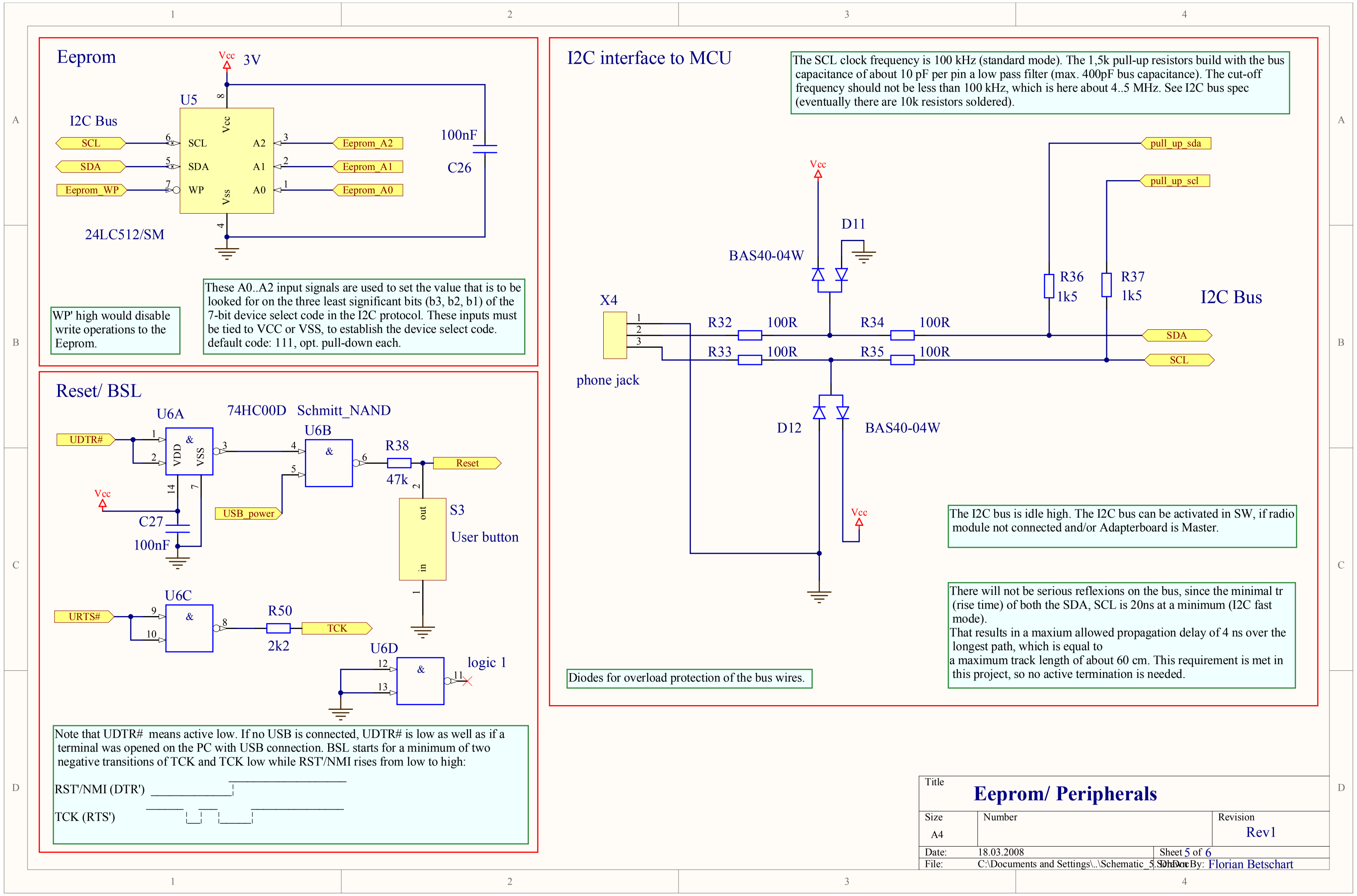
Power switch beetwen Bat/Ext and USB:
The battery supply current is separated as soon as a USB host is plugged. The n-JFET is blocking current for USB is plugged since the npn is floating and therefore the gate-source voltage is below about -2.7V, which is under the V-threshold. If USB is unplugged, the JFET is floating. A diode is necessary to prevent USB to backsupply the battery when the are almost empty. Note that the JFET is consuming power as well. Note: the higher the current consumed from VPE and LDO, the higher Vds of the FET.

Regulator LDO: ON/OFF input must be actively terminated, here to Vin to set the regulator always on.
Capacitors
to stabilize voltages. C13 reduces output noise. Output capacitor low ESR <5 mOhm/ X7R required.

Voltage divider:
If USB is plugged,
USB_power = 3V, else
USB_power = 0V.

Note: The decoupling capacitors are soldered next to the uC and the FTDI respectively, except the output capacitors next to the voltage regulators.


low ESR
C23
4,7 µF/ X7R
C22
100nF
Vcc
3V
X3
JTAG
TDO
VCCMSP
TDI
VCCSens
TMS
TCLK
TCK
TEST
GNDMSP
XIN
RST/NMI
BSLTX
BSLRX
TCK
Reset
USB_power
DAC
Quarz Crystal 4 MHz
C25
33pF
Q2
C24
33pF
C21
10uF
R24
1k
R23
1k
URTS#
UTXBSL
Out3
Out1
TXD
RXD
USB_PWREN#
Trig
All the spy lines run over interrupt pins of the MSP430.
Reset_Radio
I2C_request
URXBSL
R22
1k
Out2
In3
R21
100k
In2
In1
U4
MSP430F169
DVcc1
AVcc
DVss
AVss
P6.3/A3
P6.4/A4
P6.5/A5
P6.6/A6
P6.7/A7
VeREF+
VREF+
XIN
XOUT
VREF-/VeREF-
P1.0/TACLK
P1.1/TA0
P1.2/TA1
P1.3/TA2
P1.4/SMCLK
P1.5
P1.6
P1.7
P2.0/ACLK
P2.1/TAINCLK
P2.2/CAOUT/TA0
P2.3/CA0/TA1
P2.4/CA1/TA2
P2.5/Rosc
P2.6/ADC12CLK
P2.7/TA0
P6.2/A2
P6.1/A1
P6.0/A0
TDO/TDI
TDI/TCLK
TMS
TCK
RST/NMI
XT2IN
XT2OUT
P5.7
P5.6
P5.5
P5.4
P5.3
P5.2/SOMI1
P5.1/SIMO1
P5.0/STE1
P4.7/TB7
P4.6/TB6
P4.5/TB5
P4.4/TB4
P4.3/TB3
P4.2/TB2
P4.1/TB1
P4.0/TB0
P3.7/URXD1
P3.6/UTXD1
P3.5/URXD0
P3.4/UTXD0
P3.3
P3.2/SOMI0
P3.1/SIMO0
P3.0/STE0
Enable_Meas
V_BAT
TCK
Reset
TP1
R25
100R
R26
100R
R27
100R
limit current ~= 20 mA/ ~= 2V
Control LED from RM.
Out1
Out2
100R
R17
R18
100R
opt.
opt.
yellow
green
red
D8
D9
D10
Vcc
3V
R29 R30 R31R46
R47
R28
R48
R49
100k
Eeprom_WP
Eeprom_A0
Eeprom_A1
Eeprom_A2
S2
SW DIL-8
R20
1k
R19
1k
URXD1
UTXD1
TXD
RXD
SCL
pull_up_scl
SDA
pull_up_sda
Switch to control the software mode in the MSP430 from external side and other features.
Title
MSP430 with JTAG
Size
A4
Number
Revision
Rev1
Date:
18.03.2008
Sheet 4 of 6
File:
C:\Documents and Settings\..\Schematic_4.SchDoc
Drawn By:
Florian Betschart



WP' high would disable write operations to the Eeprom.

These A0..A2 input signals are used to set the value that is to be looked for on the three least significant bits (b3, b2, b1) of the 7-bit device select code in the I2C protocol. These inputs must be tied to VCC or VSS, to establish the device select code. default code: 111, opt. pull-down each.

Note that UDTR# means active low. If no USB is connected, UDTR# is low as well as if a terminal was opened on the PC with USB connection. BSL starts for a minimum of two negative transitions of TCK and TCK low while RST'/NMI rises from low to high:

RST'/NMI (DTR')

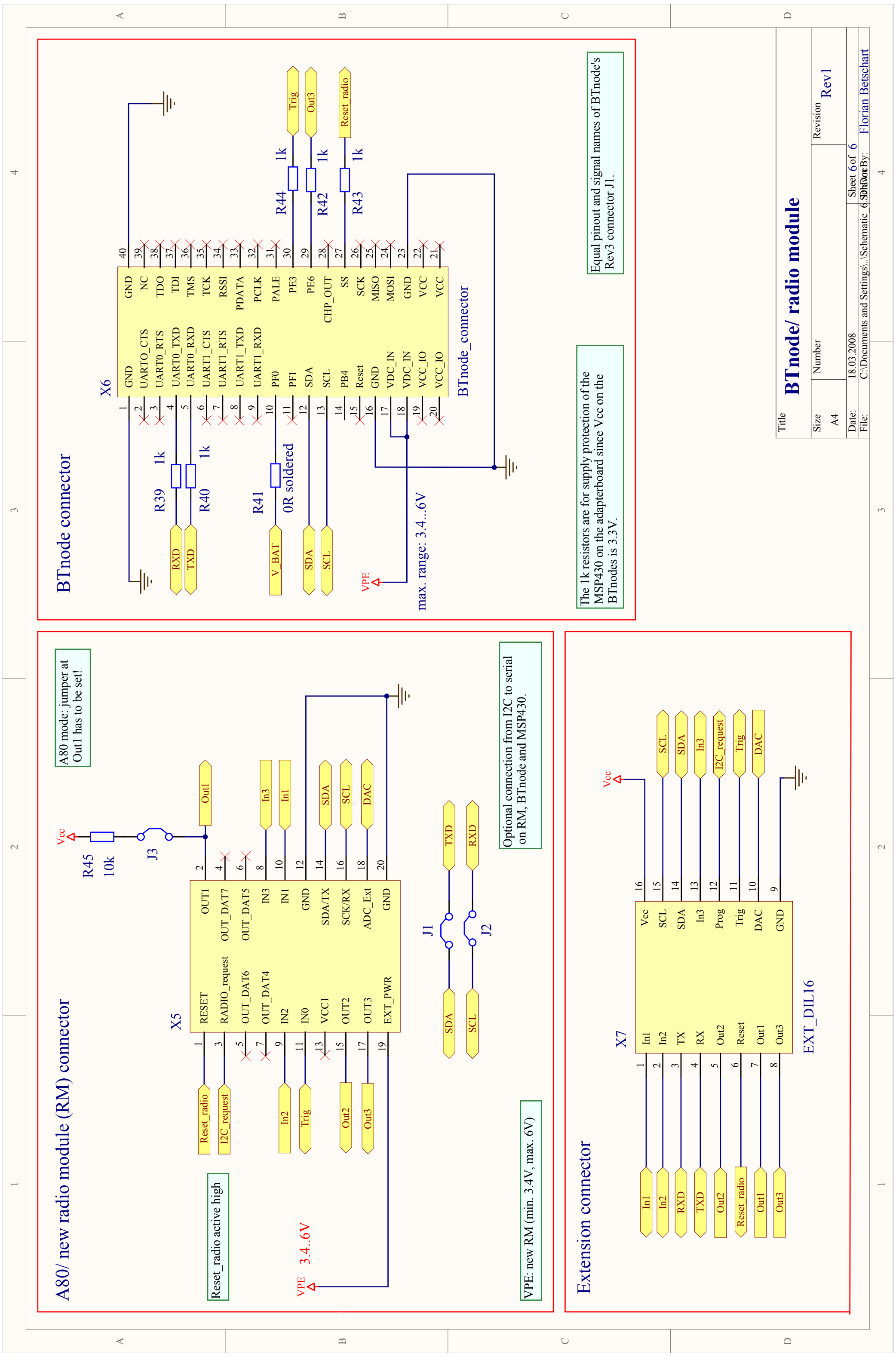
TCK (RTS')

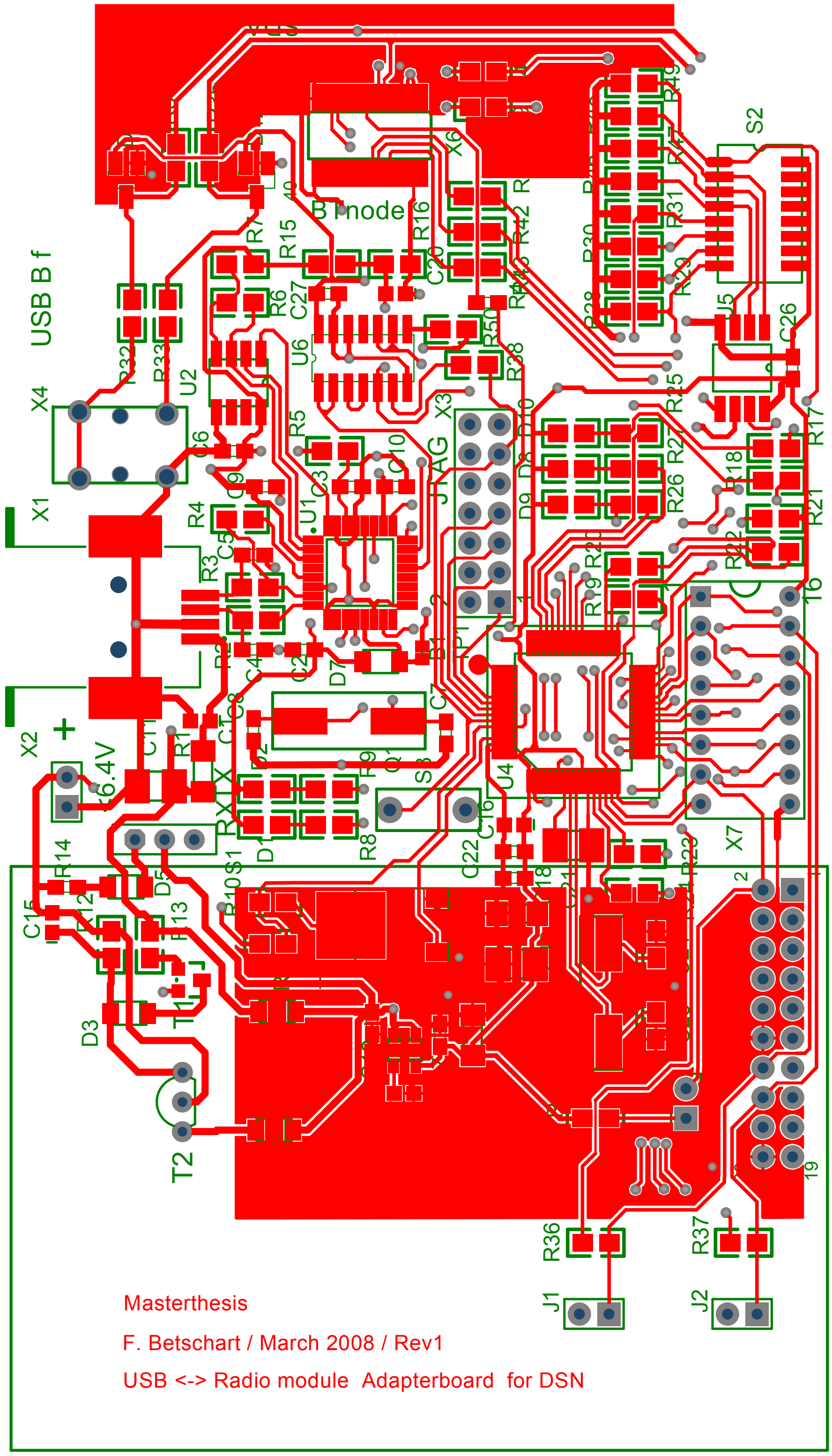
The SCL clock frequency is 100 kHz (standard mode). The 1,5k pull-up resistors build with the bus capacitance of about 10 pF per pin a low pass filter (max. 400pF bus capacitance). The cut-off frequency should not be less than 100 kHz, which is here about 4..5 MHz. See I2C bus spec (eventually there are 10k resistors soldered).

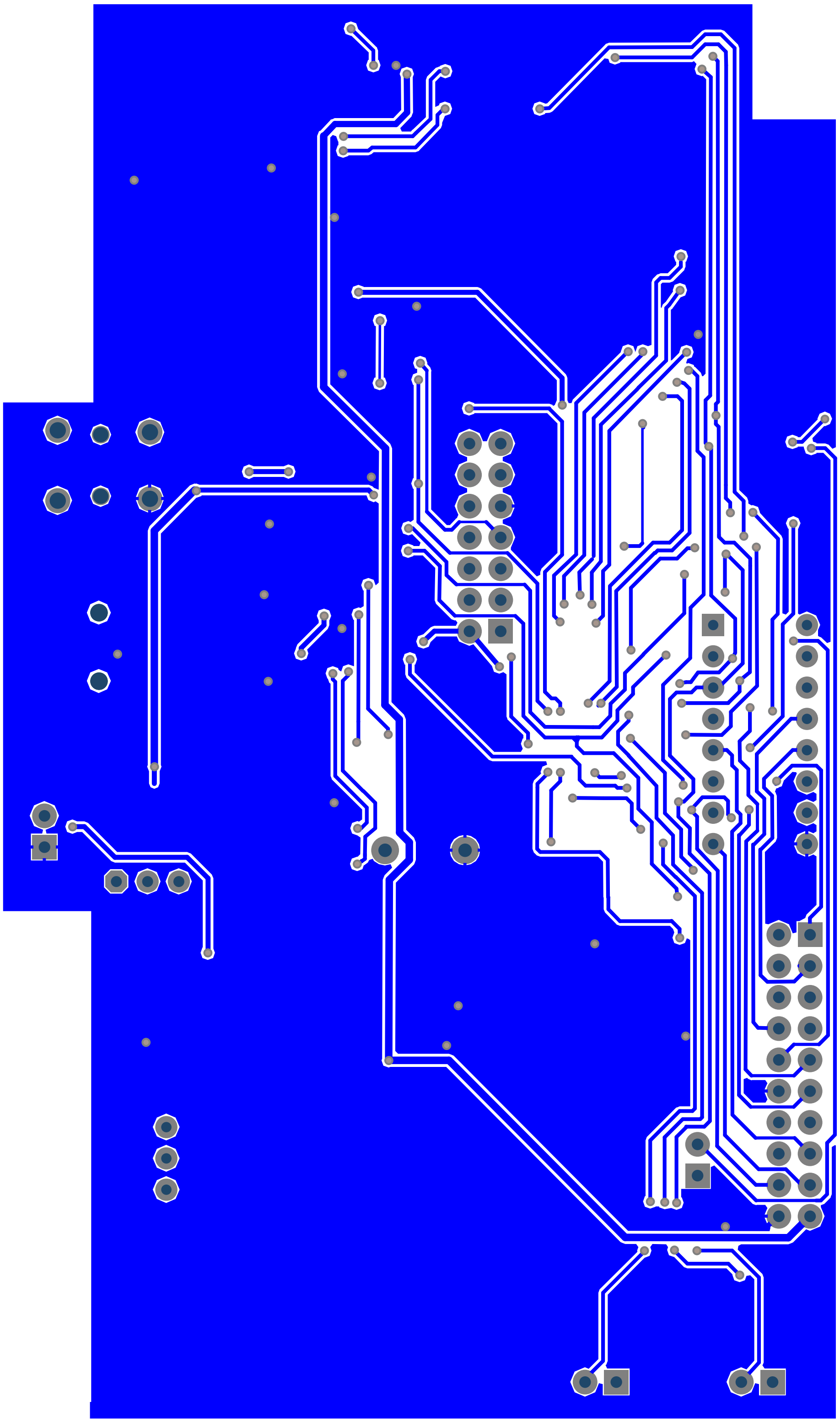
The I2C bus is idle high. The I2C bus can be activated in SW, if radio module not connected and/or Adapterboard is Master.

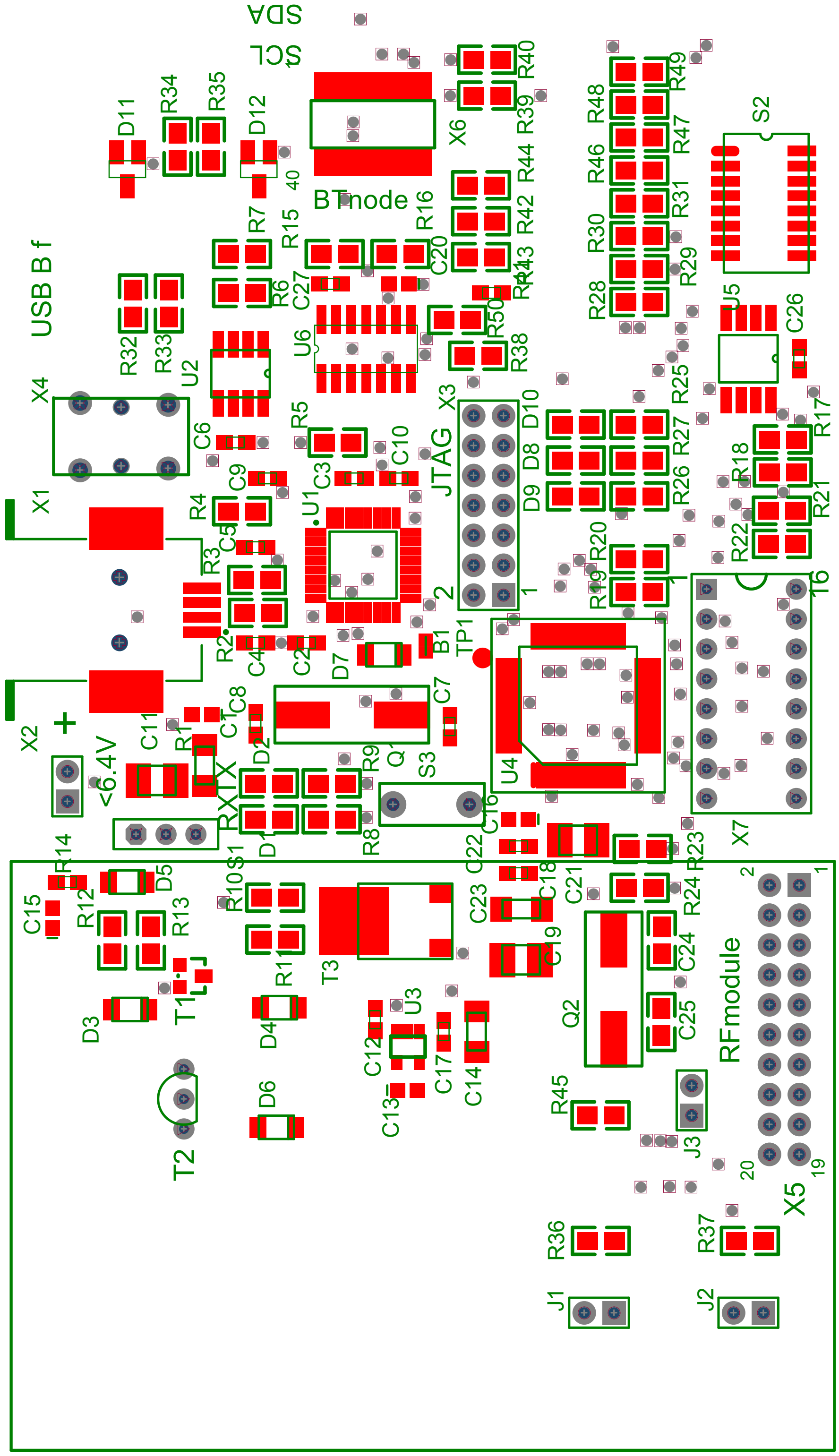
There will not be serious reflexions on the bus, since the minimal tr (rise time) of both the SDA, SCL is 20ns at a minimum (I2C fast mode).
That results in a maxium allowed propagation delay of 4 ns over the longest path, which is equal to
a maximum track length of about 60 cm. This requirement is met in this project, so no active termination is needed.

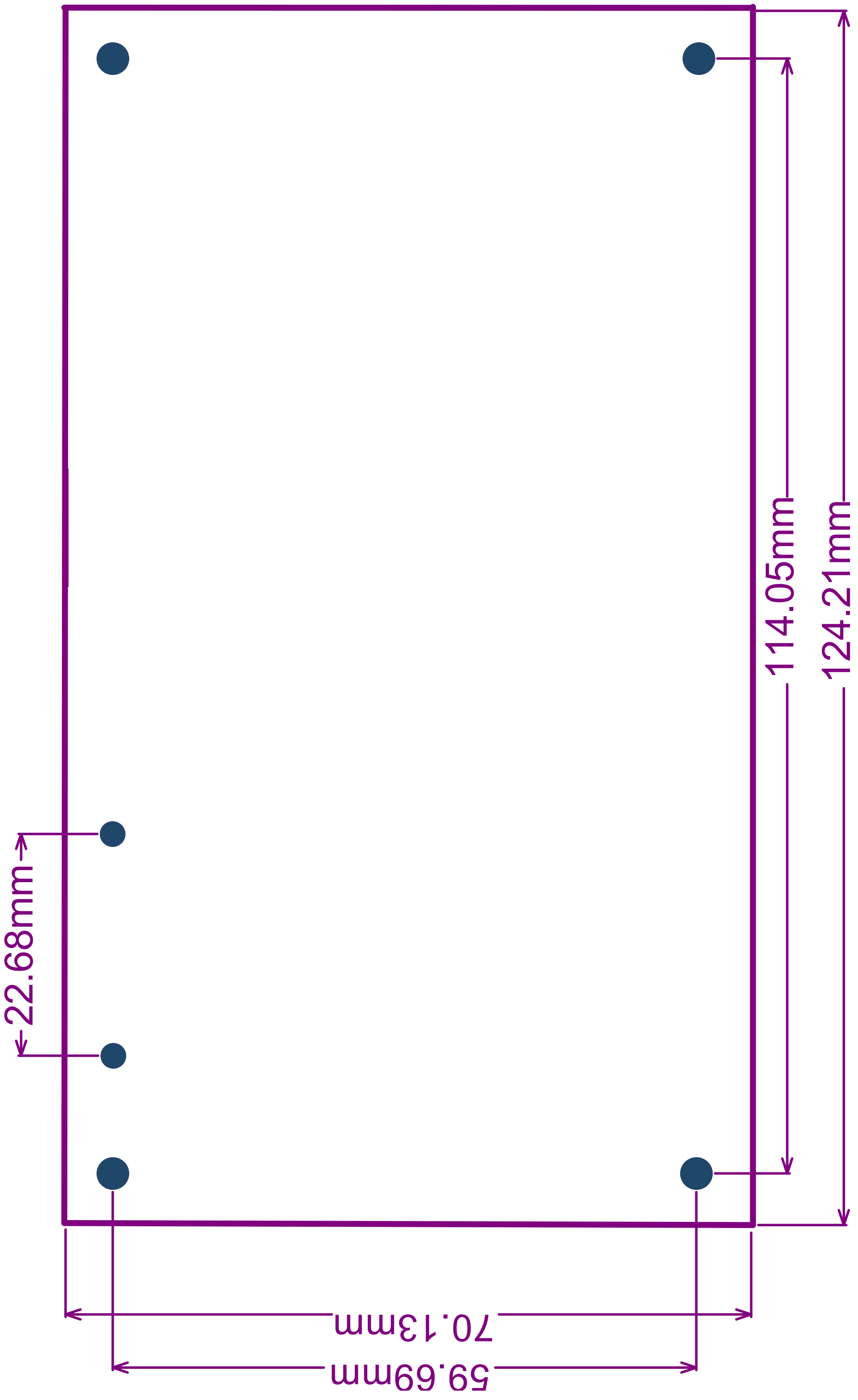
Diodes for overload protection of the bus wires.


BTnode connector
X6
BTnode_connector
GND
UART0_CTS
UART0_RTS
UART0_TXD
UART0_RXD
UART1_CTS
UART1_RTS
UART1_TXD
UART1_RXD
PF0
PF1
SDA
SCL
PB4
GND
VDC_IN
VDC_IN
VCC_IO
VCC_IO
GND
NC
TDO
TDI
TMS
TCK
RSSI
PDATA
PCLK
PALE
PE3
PE6
CHP_OUT
SS
SCK
MISO
MOSI
GND
VCC
VCC
Reset
R39 1k
R40 1k
R41 0R soldered
R42 1k
R43 1k
R44 1k
RXD
TXD
V_BAT
SDA
SCL
VPE
max. range: 3.4...6V
Trig
Out3
Reset_radio
The 1k resistors are for supply protection of the MSP430 on the adapterboard since Vcc on the BTnodes is 3.3V.
Equal pinout and signal names of BTnode's Rev3 connector J1.
A80/ new radio module (RM) connector
A80 mode: jumper at Out1 has to be set!
X5
RESET
RADIO_request
OUT_DAT6
OUT_DAT4
IN2
IN0
VCC1
OUT2
OUT3
EXT_PWR
OUT1
OUT_DAT7
OUT_DAT5
IN3
IN1
GND
SDA/TX
SCK/RX
ADC_Ext
GND
R45 10k
J3
Vcc
Out1
In3
In1
SDA
SCL
DAC
Reset_radio
I2C_request
In2
Trig
Out2
Out3
Reset_radio active high
VPE 3.4..6V
J1
J2
TXD
RXD
SDA
SCL
Optional connection from I2C to serial on RM, BTnode and MSP430.
VPE: new RM (min. 3.4V, max. 6V)
Extension connector
X7
EXT_DIL16
In1
In2
TX
RX
Out2
Reset
Out1
Out3
Vcc
SCL
SDA
In3
Prog
Trig
DAC
GND
In1
In2
RXD
TXD
Out2
Reset_radio
Out1
Out3
SCL
SDA
In3
I2C_request
Trig
DAC
Title
BTnode/ radio module
Size
A4
Number
Revision
Rev1
Date: 18.03.2008
Sheet 6of 6
File: C:\Documents and Settings\..\Schematic_6.SchDoc
Drawn By: Florian Betschart
A
B
C
D
1
2
3
4


USB B f
X4
X1
X2
+
6.4V
R32
R33
U2
R15
R16
B mode
X6
C27
U6
R5
C9
C3
U1
C10
JTAG
X3
R4
R3
C5
C4
C2
D7
R1
C8
C1
R9
R8
C16
C22
U4
X7
R14
C15
C12
D5
R13
S1
R10
D3
T1
T2
R42
R45
R50
R38
R30
R31
R28
R25
R27
R26
R18
R21
R17
R22
R29
R19
S2
J5
C26
16
2
19
R36
R37
J1
J2
Masterthesis
F. Betschart / March 2008 / Rev1
USB <-> Radio module Adapterboard for DSN




SDA
SCL
BTnode
40
D11
R34
R35
D12
R40
X6
R39
R44
R42
R43
R41
R50
R38
R48
R49
R47
R46
R31
R30
R29
R28
S2
U5
C26
USB B f
R32
R33
U2
R7
R15
R6
C27
R16
C20
U6
X4
C6
R5
C10
C9
C3
X3
JTAG
D10
D8
D9
R25
R27
R26
R17
R18
R21
R22
R20
R19
X1
R4
U1
C5
R3
2
1
T6
R2
C4
C2
D7
B1
TP1
X2
+
C8
C7
C11
R1
C1
D2
R9
Q1
S3
U4
<6.4V
RX
TX
D1
R8
C16
X7
R14
S1
R10
C22
R23
C15
R12
D5
R13
C23
C18
C21
R24
R11
T3
C19
C24
2
1
T1
D3
D4
U3
Q2
C25
RFmodule
C12
C17
C14
C13
D6
T2
R45
J3
20
19
X5
R36
R37
J1
J2


114.05mm
124.21mm
22.68mm
70.13mm
59.69mm

Appendix B

Adapterboard components list

Description	D	Comment	Distributor	Manufacturer
SMD L tbr c e	B1	not soldered		
B pass Capacitor	C1	10nF	Distrelec 830121	Murata GRM188R71 103 A01D
B pass Capacitor	C2	33nF	Distrelec 823496	Murata GRM188R71 333 A61D
B pass Capacitor	C3	100nF	Distrelec 823499	Murata GRM188R71 104 A93D
B pass Capacitor	C4	47pF / 50 V	Distrelec 830710	pcos B37930- 5470-J60
B pass Capacitor	C5	47pF / 50 V	Distrelec 830710	pcos B37930- 5470-J60
B pass Capacitor	C6	100nF	Distrelec 823499	Murata GRM188R71 104 A93D
B pass Capacitor	C7	22pF / 50 V	Distrelec 830734	pcos B37930- 5220-J60
B pass Capacitor	C8	22pF / 50 V	Distrelec 830734	pcos B37930- 5220-J60
B pass Capacitor	C9	100nF	Distrelec 823499	Murata GRM188R71 104 A93D
B pass Capacitor	C10	100nF	Distrelec 823499	Murata GRM188R71 104 A93D
lo SR	C11	10uF	Distrelec 823514	Murata GRM32DR71 106 A12L
B pass Capacitor	C12	2.2uF/ 10V X5R	Distrelec 823506	Murata RM188R61A225 34D
B pass Capacitor	C13	10nF	Distrelec 830121	Murata GRM188R71 103 A01D
B pass Capacitor	C14	lo SR	Distrelec 823523	Murata GRM319R71 475 A88L
B pass Capcacitor	C15	470pF	Distrelec 830716	pcos B37930- 5471-J60
B pass Capacitor	C16	10nF	Distrelec 830121	Murata GRM188R71 103 A01D
B pass Capacitor	C17	100nF	Distrelec 823499	Murata GRM188R71 104 A93D
B pass Capacitor	C18	100nF	Distrelec 823499	Murata GRM188R71 104 A93D
lo SR	C19	10uF	Distrelec 823514	Murata GRM32DR71 106 A12L
B pass Capacitor	C20	470pF	Distrelec 830716	pcos B37930- 5471-J60
lo SR	C21	10uF	Distrelec 823514	Murata GRM32DR71 106 A12L
B pass Capacitor	C22	100nF	Distrelec 823499	Murata GRM188R71 104 A93D
B pass Capacitor	C23	lo SR	Distrelec 823523	Murata GRM319R71 475 A88L
Quart Capacitor	C24	33pF	Distrelec 831418	emet
Quart Capacitor	C25	33pF	Distrelec 831418	emet
B pass Capacitor	C26	100nF	Distrelec 823499	Murata GRM188R71 104 A93D
B pass Capacitor	C27	100nF	Distrelec 823499	Murata GRM188R71 104 A93D
USB Receive L D	D1	reen	Distrelec 253187	Stanle PG1112
USB Receive L D	D2	red	Distrelec 253184	Stanle BR1112
Schott Diode 15ns	D3	Schott	Farnell 8734038	Philips 1PS76SB21
Schott Diode 15ns	D4	Schott	Farnell 8734038	Philips 1PS76SB21
Schott Diode 15ns	D5	opt.	Farnell 8734038	Philips 1PS76SB21
Schott Diode 15ns	D6	Schott	Farnell 8734038	Philips 1PS76SB21
Schott Diode 15ns	D7	Schott	Farnell 8734038	Philips 1PS76SB21
USB Receive L D	D8	ello	Distrelec 253186	Stanle P 1112
USB Receive L D	D9	reen	Distrelec 253187	Stanle PG1112
USB Receive L D	D10	red	Distrelec 253184	Stanle BR1112
Dual Protection Diode	D11	BAS40-04	Farnell 8734305	Philips BAS40-04
Dual Protection Diode	D12	BAS40-04	Farnell 8734305	Philips BAS40-04
Jumper and 2-pin Connector	J1	Jumper	Distrelec 122350 / 122228	
Jumper and 2-pin Connector	J2	Jumper	Distrelec 122350 / 122228	
Jumper and 2-pin Connector	J3	Jumper	Distrelec 122350 / 122228	
6 M Quar Cr stal	Q1	Cr stal 6 M	Distrelec 644810	Jauch SMU4-600-30-30/50
Quar Cr stal 4 M	Q2	4 M	Distrelec 644808	Jauch SMU4-400-30-30/50
Chip Ferrite Bead	R1	Ferrit Bead	Distrelec 110802	pcos BLM31AJ260SN1L
Resistor	R2	27R	Distrelec 713256	Visha CRC 0402
Resistor	R3	27R	Distrelec 713256	Visha CRC 0402
Resistor	R4	1 5	Distrelec 713298	Visha CRC 0805
Resistor	R5	470R	Distrelec 713286	Visha CRC 0805
Resistor	R6	2 2	Distrelec 713302	Visha CRC 0805
Resistor	R7	10	Distrelec 713318	Visha CRC 0805
Resistor	R8	100R	Distrelec 713270	Visha CRC 0402
Resistor	R9	limit current	Distrelec 713270	Visha CRC 0402
Resistor	R10	100	Distrelec 713342	Visha CRC 0805

Resistor	R11	300	Distrelec 713353	Visha CRC 0805
Resistor	R12	10M	Distrelec 713390	Visha CRC 0805
Resistor	R13	10	Distrelec 713318	Visha CRC 0805
Resistor	R14	0R opt.	Distrelec 715151	Visha CRC 0603
Resistor	R15	200	Distrelec 713349	Visha CRC 0805
Resistor	R16	300	Distrelec 713353	Visha CRC 0805
Resistor	R17	opt.	Distrelec 713270	Visha CRC 0805
Resistor	R18	opt.	Distrelec 713270	Visha CRC 0805
Resistor	R19	1	Distrelec 713294	Visha CRC 0805
Resistor	R20	1	Distrelec 713294	Visha CRC 0805
Resistor	R21	100	Distrelec 713342	Visha CRC 0805
Resistor	R22	1	Distrelec 713294	Visha CRC 0805
Resistor	R23	1	Distrelec 713294	Visha CRC 0805
Resistor	R24	1	Distrelec 713294	Visha CRC 0805
Resistor	R25	limit current	Distrelec 713270	Visha CRC 0805
Resistor	R26	limit current	Distrelec 713270	Visha CRC 0805
Resistor	R27	limit current	Distrelec 713270	Visha CRC 0805
Resistor	R28	100	Distrelec 713342	Visha CRC 0805
Resistor	R29	100	Distrelec 713342	Visha CRC 0805
Resistor	R30	100	Distrelec 713342	Visha CRC 0805
Resistor	R31	100	Distrelec 713342	Visha CRC 0805
Resistor	R32	100R	Distrelec 713270	Visha CRC 0805
Resistor	R33	100R	Distrelec 713270	Visha CRC 0805
Resistor	R34	100R	Distrelec 713270	Visha CRC 0805
Resistor	R35	100R	Distrelec 713270	Visha CRC 0805
Resistor	R36	1 5	Distrelec 713298	Visha CRC 0805
Resistor	R37	1 5	Distrelec 713298	Visha CRC 0805
Resistor	R38	47	Distrelec 713334	Visha CRC 0805
Resistor	R39	1	Distrelec 713294	Visha CRC 0805
Resistor	R40	1	Distrelec 713294	Visha CRC 0805
Resistor	R41	0R soldered	Distrelec 715151	Visha CRC 0603
Resistor	R42	1	Distrelec 713294	Visha CRC 0805
Resistor	R43	1	Distrelec 713294	Visha CRC 0805
Resistor	R44	1	Distrelec 713294	Visha CRC 0805
Resistor	R45	10	Distrelec 713318	Visha CRC 0805
Resistor	R46	100	Distrelec 713342	Visha CRC 0805
Resistor	R47	100	Distrelec 713342	Visha CRC 0805
Resistor	R48	100	Distrelec 713342	Visha CRC 0805
Resistor	R49	100	Distrelec 713342	Visha CRC 0805
Resistor	R50	2 2	Distrelec 713302	Visha CRC 0805
S itch	S1	S SPDT2 opt.	Distrelec 200180	Ni a
8- a D L s itch SMD	S2	S D L-8	Farnell 9472096	MULT C MP MC DS-08-T
JPM Push Button	S3	User button	Distrelec 200858	T co lectronics
npn bipolar	T1	NPN Bipoloar	Farnell 1081237	Philips BC849C
n-channel JF T	T2	JF T-N	Farnell 9549854	V S A S L C N X J110
n-channel M SF T	T3	RFD14N05LSM	Farnell 1017790	Fairchild SC RFD14N05LSM
Testpun t	TP1	TP		
FTD USB controller	U1	FTD 232BL	Farnell 9519769	FTD Chip
Serial eprom not 2C for USB	U2	eprom	Distrelec 644545	Microchip 93LC46B/SN
volta e re ulator 3V output	U3	LP2985	Farnell 9778250	NAT NAL LP2985A M5-3.0
MSP430F169 uC from T	U4	MSP430F169	Farnell 1172226	Te as nstruments MSP430F169 PM
Serial eprom 64 Bit 8 S -8	U5	24LC512/SM	Distrelec 644238	Microchip 24 LC512/SM
Quad 2 port NAND	U6	74 C00D Schmitt	Farnell 1201310 74 C00D	Philips 74 C00D
SMD USB Connector female B	X1	USB C NN	Farnell 1321918	T co lectronics 1734346-1
Connector	X2	t./4 AA < 6.4V	intern 4170910001 /	Altium Limited
JTAG Connector	X3	JTAG	Distrelec 122511	Preci Dip

phone ac / lin enbuchse	X4	lin enbuchse	intern A5Q00003888	
Connector to radio module	X5	A80/RM header	Distrelec 122513	Preci Dip
40 pol. BTnode Connector	X6	BTnode connector	Farnell 1077743	RS irose DF17C 2.0 -40DP-0.5V 57
16 pin header e t.	X7	XT D L16	Distrelec 650687	Preci Dip

Appendix C

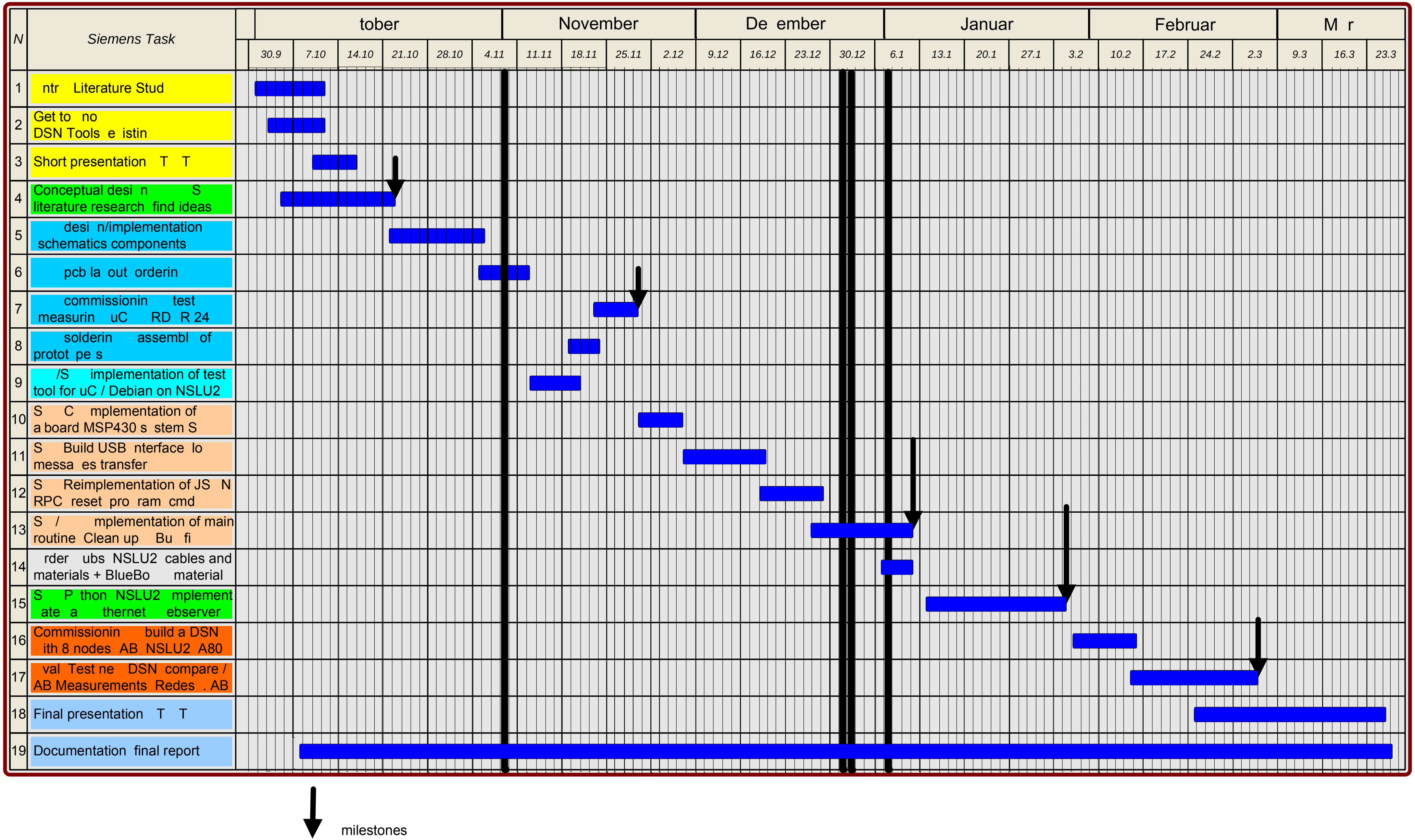
Test DSN components list

Material	Comment	Distributor	Components
Blue Bo	ammond 1591DTBU	Distrelec 300503	12
Batter case	B PLA B 60	Distrelec 970142	12
Distance holder	20 mm plastic M3	Distrelec 340398	48
ma net		.superma nete.ch	48
scre s	M3 plastic	Bossard BN1066	96
USB cable	AB m-m 0.6m	Distrelec 679829	2
USB cable	AB m-m 1.8m	Distrelec 679440	3
USB cable	AB m-m 4.5m	Distrelec 679444	4
USB cable	AB m-m 7.5m	Distrelec 679445	1
USB Stic	SanDis 2 GB	Distrelec 860955	1
USB hub	Ma tro 7-port self po ered	Distrelec 677677	2
Lin s s NSLU2	1 LAN 100 Mbps 2 USB 2.0	Brac	1

Appendix D

Work schedule

Pro ect Plan for Master s Thesis of Florian Betschart at SBT u



AB Adapterboard
A80 Siemens radio module

Appendix E

CD contents

- Thesis pdf, source code, pictures
- Presentations (entry, final)
- All PCB development files (Rev0, Rev1)
- Firmware source code
- Software source code
- Components lists
- Additional documents