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

*Computer Engineering Group*
*Prof. Lothar Thiele*

# Role and Link-State Selection for Bluetooth Scatternets

**Master thesis MA-2006-06, winter semester 05/06**

By:

**Mustafa Yücel**

Advisors:

**Prof. Lothar Thiele, Matthias Dyer, Jan Beutel**

2006-06-27

# Abstract

Distributed, wireless sensor networks captivate by their scalability and the tight integration into their environment. Large numbers of self-organising, autonomous sensor nodes are envisioned to ease observations in wide-ranging, hardly accessible terrains.

Low-power is a primary concern in the field of wireless sensor networks. Bluetooth has often been labelled as an inappropriate technology in this field due to its high power consumption. However, Luca Negri presented in a previous work a power model for Bluetooth including scatternet configuration and low-power sniff mode.

The BTnode platform is used as a demonstration platform for research in mobile and ad-hoc connected networks (MANETs) and distributed sensor networks. Some hardware components provide low-power modes but until now there are no implementations available. The BTnode will be analysed with the main focus on the power consumption. Furthermore, some low-power implementations are delivered, and experimentally validated on the BTnode platform.

Most of the wireless sensor networks provide a host-based command interface over the serial cable. Bluetooth offers with RFCOMM an alternative. A satisfying solution for the BTnode platform is delivered to eliminate the dependency on the wire.

# Acknowledgements

First of all I would like to thank Prof. Dr. Lothar Thiele for giving me the opportunity to write this thesis in his research group at the Computer Engineering and Networks Lab of the Swiss Federal Institute of Technology (ETH) Zurich.
Most of all I wish to thank Matthias Dyer for his constant support throughout the whole project. Without his valuable input and advice this work would have never been possible. In particular, I would like to thank him and Dr. Jan Beutel for giving me the opportunity to prepare a demonstration of my last term thesis and take part in the 3rd European Workshop on Wireless Sensor Networks (EWSN 2006) at ETH Zurich.
I am also grateful to Kevin Martin, Ernesto Wandeler, and Bernhard Distl for their help during this thesis work.
Finally, my sincere thanks go to my parents and my brother for their support and sponsorship during my studies.

Zurich, June 2006

Mustafa Yücel

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Distributed, wireless sensor networks captivate by their scalability and the tight integration into their environment. Large numbers of self-organising, autonomous sensor nodes are envisioned to ease observations in wide-ranging, hardly accessible terrains. Communication among sensors can be implemented with custom solutions or standardised radio interfaces. If on one hand custom solutions carry the greatest optimisation potential, the choice of widespread wireless communication standards guarantees interoperability as well as ease of connection with existing commercial devices.

Bluetooth is a leading standard for short-range ad-hoc connectivity in the PAN (Personal Area Network) field. Although initially designed for simple point-to-point cable replacement applications, Bluetooth has proved very appealing also to build multihop ad-hoc networks called scatternets.

Low-power is a primary concern in the field of wireless sensor networks. Bluetooth has often been labelled as an inappropriate technology in this field due to its high power consumption. However, Bluetooth provides low-power modes (hold, sniff, park) which trade throughput and latency for power. The papers [5] and [6] from Luca Negri present a power model of Bluetooth including scatternet configuration and low-power sniff mode, and experimental validation on a real device, the BTnode platform.

The BTnode is used as a demonstration platform for research in mobile and ad-hoc connected networks (MANETs) and distributed sensor networks. Some hardware components provide low-power modes like the microprocessor or the Bluetooth module. But until now, there are no low-power implementations available. The whole BTnode has to be analysed with the main focus on the power consumption and the best power saving proposals will be implemented.

## 1.2 Contributions

- An analysis of the power consumption of the BTnode is presented. The power consumers and the voltage converters are examined separately.
- An integration of the sleep modes from the microprocessor into the thread management of BTnut.
- The Bluetooth stack of BTnut is extended by the Bluetooth low-power commands. A terminal provides a human-friendly control interface for these commands.
- A Connection Power Manager which automatically puts connections in a greedy fashion into the low-power mode sniff.
- A Connection Manager which can handle the limitations by the Bluetooth module on the BTnode.
- Several measurements about the sleep modes, the voltage converter and the Bluetooth low-power modes.
- A RFCOMM device driver for the operating system of the BTnode platform, BTnut, is implemented. The RFCOMM device driver provides an elegant way to eliminate the dependency on attaching a cable to the BTnode

## 1.3 Overview

The thesis is organised as follows. The second chapter "Fundamentals“ introduces the basics of this thesis and covers the BTnode, the operating system, the microprocessor and Bluetooth. In the third chapter "Cable replacement using RFCOMM“ the device driver for RFCOMM will be presented. In the fourth chapter "Applying the sleep modes“ the power save potential of the microprocessor ATmega128L will be evaluated. In the fifth chapter "Bluetooth Link Power Control“ Bluetooth measurements are performed and a power manager for Bluetooth connections will be implemented. The sixth chapter "A low-power connection manager for JAWS“ describes the limitations by the Bluetooth module on the BTnode platform and a new connection manager will be implemented. The last chapter includes the conclusion and perspectives for future work.

## 1.4 Related Work

Bluetooth provides three low-power modes to applications: hold, sniff, and park. To take advantage of these features, applications need a power model of the device, describing power behaviour in all possible states (number of links, active, sniff, etc.). There is indeed a lack of such a model in the literature to date. Many Bluetooth power-related proposals like [12] and [13] are based on oversimplified power models, not considering number and role (master and slave) of links. Such models are normally not based on experimental measurements, but rather on theoretical assumptions. Even worse, other Bluetooth-related studies employ rather old and inadequate power models that were derived for other wireless systems [14]. Finally, the few power measurements for Bluetooth in the literature like [15] and [16] do not cover Bluetooth low-power modes and scatternet configurations.

However, [5] and [6] from Luca Negri present a high-level power model of Bluetooth in a generic piconet or scatternet scenario, including the low-power sniff mode. Unlike most Bluetooth power abstractions employed in the literature, the model has been experimentally validated on a real device, the BTnode platform.

# Chapter 2

# Fundamentals

This chapter serves as an approach to the basics of this thesis. In the first subchapter, there is a description about the hardware components of the BTnode and the operating system running on it. The second subchapter lists the options to save power on the microprocessor. In the third subchapter, there is an overview about the protocol stack of Bluetooth, followed by some Bluetooth communication principles in the fourth subchapter. The fifth subchapter declares the significance of a connection and transport manager to render a multihop communication. At last, the power saves modes for Bluetooth connections are introduced in the sixth subchapter.

Some knowledge is expected about the programming languages C and Java, and about the basics of operating systems, embedded systems and communication systems. Terms like object file, thread management, flash and multihop turn up in this thesis and the meanings should be familiar to the reader without further explanations.

## 2.1 The BTnode

The BTnode mainly consists of an Atmel AVR microcontroller and two wireless communication modules. The first one is a Bluetooth module from Zeevo, the second one is an ultra low-power radio module from Chipcon. Figure 2.1 shows a simple component scheme of the BTnode. The BTnode is a very compact, programmable platform, deployed e.g. as a mobile sensor node. It is used as a demonstration platform for research in mobile and ad-hoc connected networks (MANETs) and distributed sensor networks. The BTnode has been jointly developed at the ETH Zurich by the Computer Engineering and Networks Laboratory (TIK) and the Research Group for Distributed Systems (DSG).

The hardware specifications of the BTnode are listed as follows:

**Microcontroller** Atmel ATmega 128L (7.3728 MHz @ 8 MIPS)

**Memory** 4 KB RAM, 4 KB EEPROM, 128 KB Flash-ROM, 256 KB external SRAM

**System clock** 32.768 kHz real time clock and 7.3728 MHz system clock

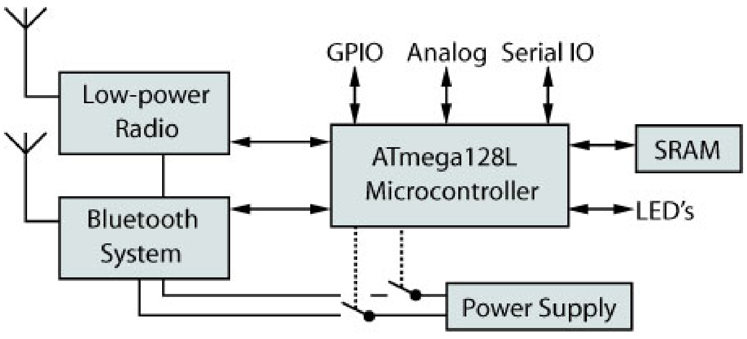


Figure 2.1: A component scheme of the BTnode

**Bluetooth** Zeevo ZV4002, supports scatternets with max. 4 piconets/7 slaves, compatible with Bluetooth Specification 1.2

**Low-power radio** Chipcon CC1000, operating in ISM band 433-915 MHz

**External Interfaces** ISP, UART, SPI, I2C, GPIO, ADC, 4 LEDs

The Bluetooth module provides a relatively high data throughput whereas the radio module consumes less power. Both radios may be operated simultaneously, or independently powered off when not in use.

The BTnode has a couple of power supply options. 2-cell AA batteries are the standard power supply with a common range of 1.2-3.2 V. The primary boost converter for the batteries has a nominal input range of 0.5-3.3 V. Alternatively 3.6-5.0 V can be supplied through the external connector. As a last option the BTnode can be directly connected to a 3.3 V regulated power supply.

The BTnodes run a real-time operating system called Nut/OS[1] for the ATmega128 microcontroller. It is freely available as open source[2] on the Internet. Nut/OS provides the following features:

- Nonpreemtive cooperative multithreading
- Events
- Thread synchronisation mechanisms
- Periodic and asynchronous one shot timers
- Dynamic heap memory allocation
- Interrupt driven streaming I/O

BTnut[3] is a heavyset system software for the BTnodes, consisting of the Nut/OS, specific drivers for the BTnode and a partial implementation of the Bluetooth stack.

[1]http://www.ethernut.de/en/software.html
[2]http://en.wikipedia.org/wiki/Open_source
[3]http://www.btnode.ethz.ch/static_docs/doxygen/btnut/index.html

## 2.2 Power save modes for ATmega128L

The current consumption of the ATmega128L is a function of several factors such as: operating voltage, operating frequency, loading of I/O pins, switching rate of I/O pins, executed code, and ambient temperature. The dominating factors are operating voltage and frequency. Unfortunately the voltage cannot be changed because some components demand 3.3 V. But the frequency remains adjustable even while the microprocessor is running. Figure 2.2 shows almost a linear behaviour between current consumption and frequency: $I_{Supply} \sim f$. According the figure 2.2 the supply current is located around 9.5 mA when the ATmega128L runs at 8 MHz.

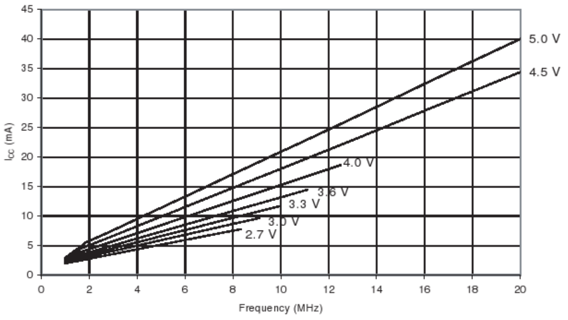
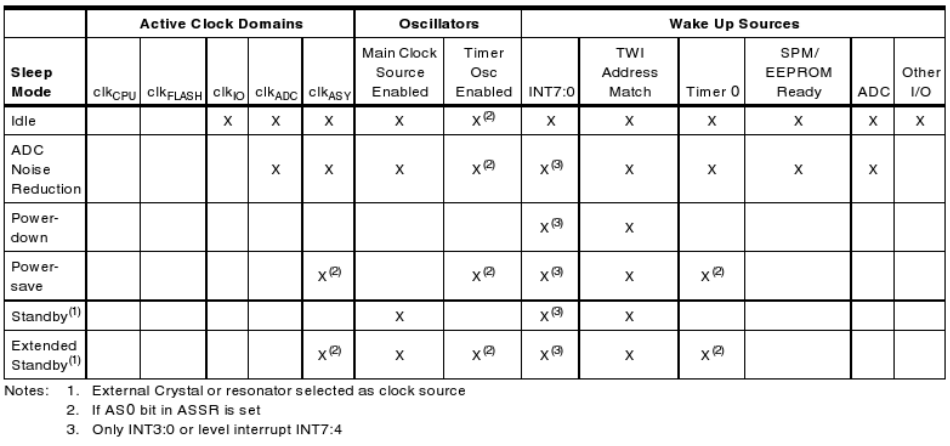


Figure 2.2: Supply current vs. frequency (ATmega128L) [3]

Beside the frequency scaling the sleep modes can be enabled by the application to shut down unused modules in the ATmega128L to save power. The ATmega128L provides various sleep modes allowing the user to tailor the power consumption to the requirements. In all sleep modes the CPU clock halts. When a wake up source is triggered the CPU clock continues and the program execution is resumed. The more clocks are halted the more power will be saved but lesser wake up sources are available. Figure 2.3 shows a table with active clock domains and available wake up sources in different sleep modes.



| | Active Clock Domains | | | | | Oscillators | | Wake Up Sources | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sleep Mode** | $clk_{CPU}$ | $clk_{FLASH}$ | $clk_{IO}$ | $clk_{ADC}$ | $clk_{ASY}$ | Main Clock Source Enabled | Timer Osc Enabled | INT7:0 | TWI Address Match | Timer 0 | SPM/ EEPROM Ready | ADC | Other I/O |
| Idle | | | X | X | X | X | X(2) | X | X | X | X | X | X |
| ADC Noise Reduction | | | | X | X | X | X(2) | X(3) | X | X | X | X | |
| Power-down | | | | | | | | X(3) | X | | | | |
| Power-save | | | | | X(2) | | X(2) | X(3) | X | X(2) | | | |
| Standby(1) | | | | | | X | | X(3) | X | | | | |
| Extended Standby(1) | | | | | X(2) | X | X(2) | X(3) | X | X(2) | | | |

Notes: 1. External Crystal or resonator selected as clock source
2. If AS0 bit in ASSR is set
3. Only INT3:0 or level interrupt INT7:4

Figure 2.3: Wake up sources in different sleep modes [3]

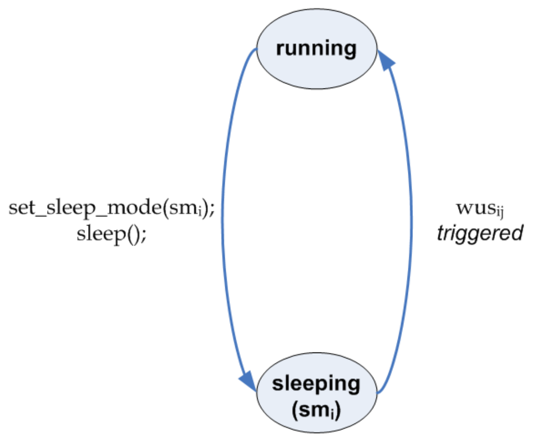


Figure 2.4: State transition model for sleep modes

Figure 2.4 illustrates possible sleep mode transitions. In order to change the sleep mode $sm_i$, the wake up source $wus_{ij}$ has first to be triggered.

## 2.3 Overviewing the Bluetooth stack

The Bluetooth stack is a standardised implementation of a protocol stack. The architecture model illustrated in figure 2.5 makes up the basis for Bluetooth communication. The stack contains several layers where to each layer certain functions and tasks are assigned. The interfaces between the layers as well as the protocols are standardised, manufacturer-independent and publicly available.

The stack also provides with HCI an interface between controller (the Bluetooth module) and host (the microcontroller). Most Bluetooth modules come with HCI out of the box. This means that all layers below HCI are implemented and provided with the appropriate firmware. All layers above HCI will usually not be implemented in the module. To be precise, they will be implemented in the operating system and are usually running as a background daemon.

*Bluetooth Radio* provides the FHSS[4] (Frequency Hopping Spread Spectrum) system which sends and receives packets in determined time slots and defined frequencies.

*Baseband* controls the radio and works up the data for higher layers. The Baseband layer manages the physical radio channels and connections. It is also responsible for error correction and the determination of the hopping sequence.

*Link Manager Protocol* (LMP) is responsible for establishing and terminating connections. LMP also manages the power saving modes on connections.

[4]http://en.wikipedia.org/wiki/Frequency-hopping_spread_spectrum

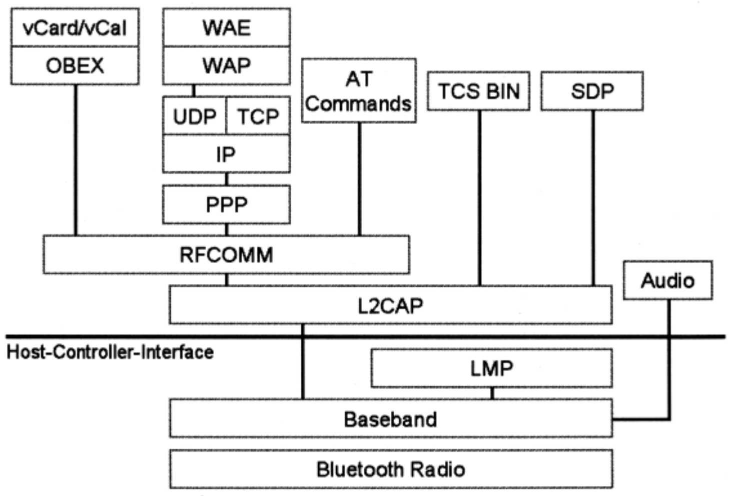


Figure 2.5: The architecture model of the Bluetooth stack [1]

*Host Controller Interface* (HCI) is the command interface to the baseband controller and link manager. The host (the microcontroller) communicates over HCI with the Bluetooth module. The most important function is the discovery of Bluetooth devices in range. This function will be called Inquiry or Device Discovery.

*Logical Link Control and Adaptation Protocol* (L2CAP) provides connection-oriented and connectionless data services by exchanging packets. Multiplexing, segmentation and reassembling are such services. L2CAP allows higher protocols and applications to send and receive L2CAP data packets with a payload right up to 64 kilobytes. L2CAP is channel-based. A channel is a logical connection within a physical baseband connection. Hence several channels may share the same baseband connection. The channels are distinguished and classified with a *Protocol Service Multiplexer* (PSM) number.

*Radio Frequency Communication Protocol* (RFCOMM) emulates a serial interface RS-232[5] between two Bluetooth devices. RFCOMM is a stream-based transport protocol and supports applications which usually act on a serial port. RFCOMM multiplexes up to 60 concurrent connections between two Bluetooth devices. These connections are addressed with a RFCOMM channel number between 1 and 30. Additionally all status and control signals of RS-232 are provided.

*Service Discovery Protocol* (SDP) allows client applications to request information of the running services on the device. SDP indicates the sort and classification of the services as well as the protocols in use. But SDP, in fact, does not provide any mechanisms to use these services. The SDP client generally searches for services which achieve the desired attributes. This function is called Service Discovery. But it is also possible to perform a general search which would list all provided services. This function is called Service Browsing.

*Object Exchange Protocol* (OBEX) is a common protocol for the file transfer

[5]http://en.wikipedia.org/wiki/RS-232

between mobile devices. At first, it was used for infrared communication, nowadays OBEX is applicable to Bluetooth as well.

The Bluetooth module on the BTnode accepts HCI commands over UART. Currently HCI, L2CAP and RFCOMM protocols are implemented in BTnut. An implementation of the SDP layer is planned.

## 2.4 Principles in Bluetooth communication

Bluetooth is based on 79 independent channels where a Bluetooth connection employs a frequency hopping algorithm among those channels. The MAC[6] (Media Access Control) layer is based on a TDMA[7] (Time Division Multiple Access) scheme using slots of 625 $\mu$s each.

The particular feature of Bluetooth is the manner in which networks are constructed autonomously. Such networks are called ad-hoc networks. This sort of network formation is decentrally organised. For that, no special device or infrastructure is needed.

#### Inquiry

The device seeks for other Bluetooth devices. The inquiring process returns all Bluetooth hardware addresses from devices with Inquiry Scan enabled.

#### Inquiry Scan

With Inquiry Scan enabled the device itself spots for inquiry attempts. When an Inquiry is detected an inquiry response packet is sent back to the source containing its own Bluetooth hardware address. The private state *discoverable* may be controlled by enabling and disabling the Inquiry Scan.

#### Piconet

When at least two Bluetooth devices establish a connection, a so-called piconet is built up. Due to the frequency hopping technique the hops have to be coordinated. This means that parameters like sequence and phase must be prearranged. This process is governed by the device which initiates the connection. The same device plays an important role in the piconet and it is called master. Each piconet must have one and exactly one master. The involved communication partners which follow the instructions from the master are called slaves. Master/slave communication is handled in a TDD[8] (Time Division Multiplexing) fashion. Each slave gets time slots assigned by the master. In these time slots the master and the associated slave may exchange data. Altogether the master manages up to seven slaves. Figure 2.6

[6]http://en.wikipedia.org/wiki/Media_Access_Control
[7]http://en.wikipedia.org/wiki/TDMA
[8]http://en.wikipedia.org/wiki/Duplex_(telecommunications)

shows an example of a piconet with one master and four slaves. If necessary the role of a connection may be changed between master and slave at any time.

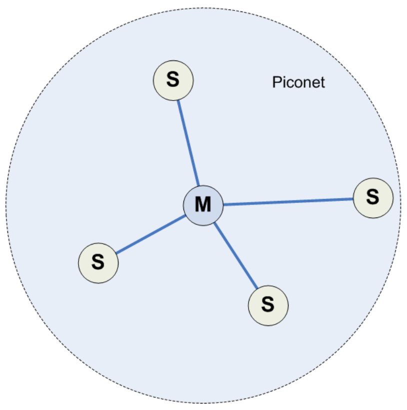


Figure 2.6: A single Bluetooth piconet with four slaves

### Page

If a Bluetooth device with a known hardware address is in range and with Page Scan enabled, a connection may be established. In order to reach the connected state, the paging process synchronises the slave device with the master device.

### Page Scan

Similar to the Inquiry Scan the device replies to page attempts. The private state connectable may be controlled by enabling and disabling the Page Scan.

### Scatternet

In various situations it is required for Bluetooth devices from different piconets to communicate with each other. A Bluetooth device can participate concurrently in several piconets. The network established by piconets is called scatternet which is illustrated in figure 2.7. Scatternet allows some sort of intercommunication between piconets and provides the basis for a multihop communication. The different piconets act in individual frequency hopping sequences. Thus the collision probability is marginal.

## 2.5 Connection and transport manager

The Bluetooth standard contains no specification for the formation and control of multihop topologies or for the data transport across multiple hops. An additional functional layer must provide these services, i.e. for a modular structure, one layer

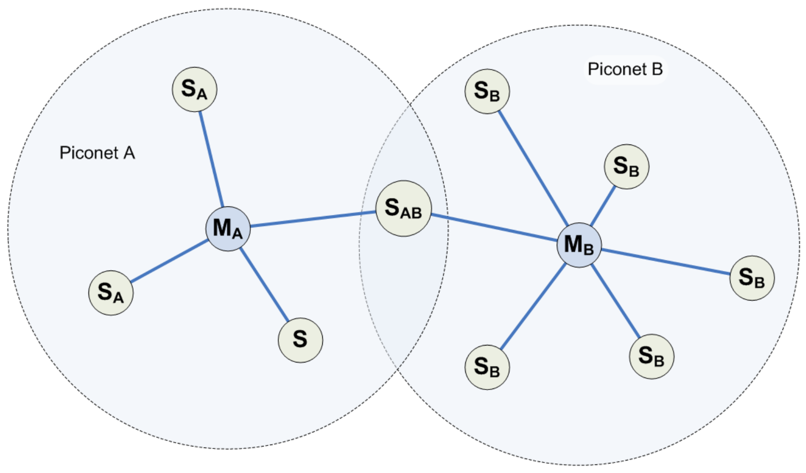


Figure 2.7: A Bluetooth Scatternet containing two piconets

for the topology control and one layer for the data transport.

The connection manager constructs and maintains a multihop network of several nodes. The basic principle is simple: every node periodically searches for other nodes in its range and connects to the discovered devices based on certain conditions. A robust algorithm should be used that automatically takes care of nodes that join or leave the network. This algorithm should provide self-healing topologies in a completely distributed fashion.

The transport manager takes care of multihop packet forwarding. It receives information about available connections from the connection manager. The transport manager provides a connectionless transport type. It can be specified whether all or only a certain device receives the packet. Finally the packets will be passed to an upper layer.

## 2.6 Power save modes for Bluetooth connections

Due to the primary target to integrate Bluetooth generally into mobile devices, various low-power modes are defined. They guarantee a lower current consumption and longer operational lifetime. The Bluetooth Specification [2] defines four power modes: *Active*, *Hold*, *Sniff*, and *Park*. These power modes are applied to single connections. If a connection is established, it starts in the standard power mode *Active* and may be changed during the connected stage.

Besides the low-power modes the Bluetooth Specification [2] describes a power control mechanism to minimise the average current consumption. Whenever the received signal strength is outside of a determined range, the sender can be invited to increase or decrease his transmit power. This be will done for each connection separately. Especially if both devices are very close superfluous current will be saved.

### Active

In active mode slaves wait for transmissions from the master. While the slave is in active mode, all packets from his master have to be received and evaluated. With this method the slave receives own packets. If the master commits a clear to send, the slave can start the data transmission. In active mode the shortest reaction time is awaited. Unfortunately the current consumption is the highest due to the persistent send/receive readiness.

### Hold

In hold mode the whole communication activity within the connection will be stopped for an adjustable time. After the hold time the slave takes again part in the previous piconet.
The *Hold Interval* defines the number of slots in which the connection lingers in hold mode.

### Sniff

In sniff mode the slave will be active periodically for a short time. The master only communicates to the slave during a prearranged active time window. When more send data is available within the window, it is possible to enlarge the window size one-time. The time ratio between inactive and active stage can be adjusted in a flexible fashion. Thus, the current consumption is indirectly affected. If the active stage is shortened, the current consumption is decreased but increases the reaction time.
The *Sniff Attempt* (SA) defines the size (in the number of receive slots) of the window. The *Sniff Interval* (SI) defines the period (in the number of slots) between the windows. The *Sniff Timeout* (ST) defines the number of receive slots in which the sniff partner might resume the data stream.

### Park

The lowest possible current consumption is reachable in park state. The slave is still synchronised with the master but no more graded as active. Data communication is not feasible. The park state allows the master to increase the piconet members. Hence a master maintains up to seven slaves and 255 parked slaves. The beacon time window is used to reactivate a slave.
The *Beacon Interval* defines the period (in the number of slots) between the windows.

# Chapter 3

# Cable replacement using RFCOMM

In my last term thesis [17] I worked on the communication between BTnode and PDA using the standard Bluetooth API JSR 82 for Java. The task was to realise wireless interactions like upload a flash image to the BTnode or retrieve informations about the connected BTnode. The current work in this chapter is thought as a preliminary task. The knowledge from the last term was applied to eliminate the dependency on attaching a cable to the BTnode.

The first subchapter explains why RFCOMM is used, followed by some implementation design thoughts in the second subchapter. On the basis of an example, the usage of the RFCOMM device driver is described in the third subchapter. The fourth subchapter lists some annotations and limitations by the RFCOMM device driver. In order to reduce the size requirements when using the device driver, the Bluetooth stack is slimmed down in the fifth subchapter. The sixth subchapter is concerned about the Bluetooth API for Java: JSR 82. Additionally there is a market analysis about available JSR 82 implementations in J2SE for various operating systems. At last, the seventh subchapter lists the options to communicate with a BTnode over RFCOMM.

## 3.1 Why RFCOMM?

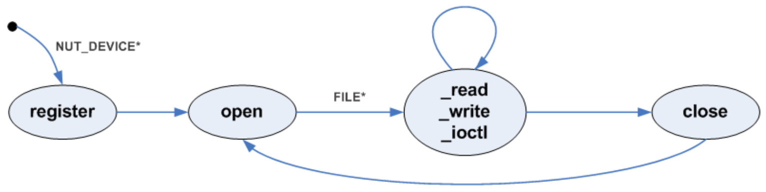
On different computer and telecommunication devices the standardised serial interface RS-232 remains indispensable. PDAs, notebooks, mobile phones and other devices like our BTnode use the serial interface for wired communication. Bluetooth contains the function to replace the wire. For that reason the RFCOMM layer is embedded in the Bluetooth stack. It is also named *Serial cable emulation protocol* in the Bluetooth Specification [2].

As the RFCOMM layer is designed to replace a cable, it is good idea to base on this protocol. Various operation systems also offers virtual serial ports for RFCOMM. Additionally the operating system BTnut comes with a stream interface for standard I/O. This simplifies the RFCOMM integration and reduces changes in existing applications.

## 3.2 The implementation design for RFCOMM

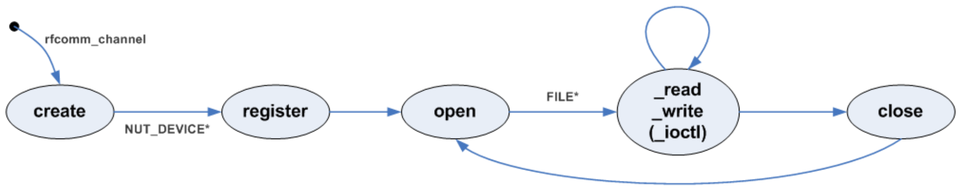
The RFCOMM layer was already implemented in BTnut and provides a simple API. But employing this API is an expensive way to extend existing software with RFCOMM. In BTnut all I/O runs over a standard stream interface. Thus extending the stream interface to handle and maintain RFCOMM channels is a transparent and fast working solution.

Device drivers in BTnut use an abstraction layer with a predefined API to access devices. Generally a device driver goes through the following sequence.



`register` calls some initialisation functions. `_ioctl` is used to configure parameters while the device is opened. But this sequence raises one problem. To establish a RFCOMM connection, the Bluetooth hardware address and the RFCOMM channel have to be known. Thereby `_ioctl` cannot be used to commit the destination parameters. Without specifying both parameters, a RFCOMM channel cannot be opened.

This problem will be solved by using a skeleton device driver for RFCOMM channels. First, for each channel, the appropriate device has to be created. The driver just accepts connections resp. listen on channels to omit the declaration of a hardware address. The RFCOMM device driver goes through the following, slightly modified sequence:



`_ioctl` is no more used but may be extended to specify some read or write timeouts. `open` listens on the determined channel. `close` disconnects the channel and stops listening. To have some feedback when a connection is established resp. the channel is opened, a connection callback may be passed with `create`.

## 3.3 Usage of the RFCOMM device driver

The application `btnut/app/bt-cmd` from the BTnut software repository, which offers a HCI command terminal over the serial interface, will be extended with the

RFCOMM device driver. This subchapter includes a step-by-step guide to integrate the RFCOMM device driver into existing applications. This application will also be used in subchapter 3.5 to determine the additional size gap caused from including the device driver. The extended application will be saved under `btnut/app/bt-cmd-over-rfcomm`.

First the L2CAP and RFCOMM stack have to be initialised. After initialising the stacks, the RFCOMM device is created using `RFCommCreate`:

```
NUTDEVICE * devRFComm01 = RFCommCreate(rfcomm_stack, 1, con_cb);
```

The RFCOMM channel is delivered as second parameter. The third parameter is the connection callback, referring to a following function:

```
void con_cb(u_char channel, u_char connected)
```

Whenever a RFCOMM channel opens or closes, this function will be called after. The first parameter is the channel number (in this case 1). The second parameter presents the current state of the channel. In this application the function is used to print out debug messages on the serial line as well as welcome messages on the RFCOMM channel.

Next the RFCOMM device `devRFComm01` is registered with `NutRegisterDevice`:

```
NutRegisterDevice(devRFComm01, 0, 0);
```

The last two parameters are not used in the RFCOMM device driver and should be always zero.

Finally the RFCOMM device is opened using `fopen`:

```
FILE * rfcomm_terminal = fopen(devRFComm01->dev_name, "r+");
```

The second parameter specifies the access mode: `"r"` is for read-only, `"w"` for write-only, and `"r+"` for read/write access. Because a RFCOMM channel cannot be trimmed to unidirected communication, the device driver undertakes this function. Thus e.g. no receive buffer is build up in write-only mode.

Now the file `rfcomm_terminal` can be used to read from the RFCOMM channel or write to the RFCOMM channel. All stream I/O functions from `stdio.h`[1] library are ready to perform over RFCOMM. At least, to handle the whole terminal communication over RFCOMM, this file is passed when initialising the terminal:

```
btn_terminal_init(rfcomm_terminal, TERM_PREFIX);
```

With previously presented, minimalistic changes all reads and writes by the terminal application run over RFCOMM. As already mentioned this application is saved under `btnut/app/bt-cmd-over-rfcomm` into the repository.

[1] http://www.btnode.ethz.ch/static_docs/doxygen/btnut/group__xgCrtStdio.html

## 3.4 Limitations by the RFCOMM device driver

The RFCOMM device driver is primary designed to replace a cable. Thus not all possible functions are implemented. Keep in mind that the device driver should be as small as possible because the whole Bluetooth stack is used too and this stack already uses up much space.

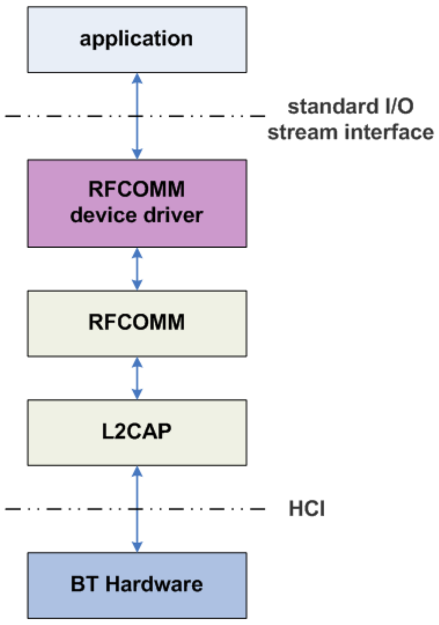


Figure 3.1: Overviewing the RFCOMM device handling

**only passive connections** As said the device driver cannot establish connections actively. If the device is opened, it only listens and accepts connections. However it provides still some sort of connection feedback.

**one connection per channel** The current implementation of the RFCOMM layer cannot handle multiple connections on one channel. At least one connection is feasible per channel.

**no timeouts** There is no timeout implementation. Thus any read access will block until at least one byte is received.

**no cooked mode** By default, the cooked mode is enabled in a serial port implementation. The cooked mode will map characters `\n` and `\r` to `\r\n`. Most of the terminal clients expect `\r\n` to display a new line correctly. So far the RFCOMM device driver only supports the raw mode.

**synchronous send** All writes to the devices are directly sent over RFCOMM. No send buffer is provided. This may lead to unwanted effects. A single `fprintf` function may send several RFCOMM packets at once. E.g. the format string `"uptime: %d s\n"` results to three RFCOMM packets with the payload `"uptime: "`, `"13"`, and `" s\n"`.

## 3.5 Slim down the Bluetooth stack of BTnut

With bootloader, the flash memory on the microprocessor ATmega 128L is limited to 120 KB. When the RFCOMM device driver is used, the Bluetooth stack takes up a great part of the flash. Figure 3.2 exposes the maximum size requirements of the Bluetooth stack layers determined by the object file sizes in BTnuts Bluetooth library `btnut/lib/btnode3/libbt.btnode3.a` (with disabled logging).

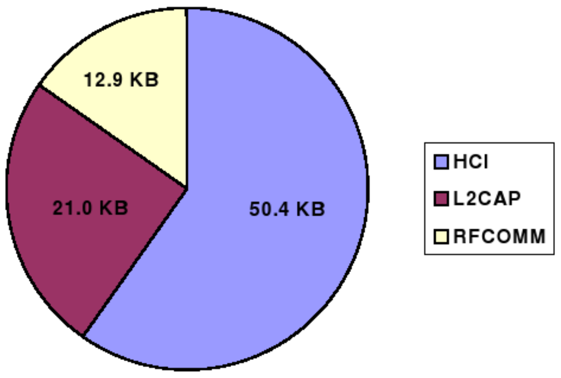


Figure 3.2: Size of the Bluetooth stack layers (2006-05-17)

But these numbers only represents the size if all object files are linked. As practical reference the application sizes of `btnut/app/bt-cmd` and of the extended version `btnut/app/bt-cmd-over-rfcomm` from subchapter 3.3 will be compared. The size grows due to the including of the RFCOMM device driver from 64.2 KB to 84.1 KB. Remember that the HCI layer was already linked in `bt-cmd`. Only the object files of L2CAP and RFCOMM are additionally included. The size difference is 19.9 KB. Due to the limitation to 120 KB such a high difference may not be tolerated. Thus reducing the size requirements of L2CAP and RFCOMM will be the next step.

In certain situations the BTnode never creates L2CAP or RFCOMM connections actively. It has only to accept remote initiated connections. In this case, connect commands are never used and some responses never turn up. The two macros `BT_L2CAP_SLIMDOWN` and `BT_RFCOMM_SLIMDOWN` are added to the source code. These macros disables the appropriate ability to create connections. These macros can be enabled in `btnut/Makedefs`. If both macros are enabled the size of the application `bt-cmd-over-rfcomm` is going down to 80 KB. It is not much but this method is still compatible with the Bluetooth Specification [2]. Other functions like defragmenting and assembling of L2CAP packets are not used in certain cases and can be disabled but breaks the Bluetooth Specification [2]. With such methods a minimum size of 75 KB is reachable.

To summarise the progress: L2CAP, RFCOMM and the appropriate device driver costs 19.9 KB. If both `SLIMDOWN` macros are enabled the value will be reduced to 15.8 KB. With some other invasive modifications (CVS revision 1.45 of `btnut/btnode/bt/bt_l2cap.c` and 1.14 of `btnut/btnode/bt/bt_rfcomm.c`) a minimum value of 10.8 KB is reachable.

## 3.6 The Bluetooth API for Java: JSR 82

JSR 82[2] provides a standard API for the development of Bluetooth applications and profiles in Java and allows a consistent access to the Bluetooth functionality and abilities. The API was primarily designed for Bluetooth mobile phones running *Java Micro Edition*[3] (J2ME). But JSR-82 itself does not offer any implementation. There are preinstalled implementations on mobile phones as well as several implementations for the *Java Standard Edition* (J2SE) on varied operating systems. For further information about JSR 82 the book ”Bluetooth Application Programming with the Java APIs“ [4] is highly useful.
JSR 82 maintains Device Discovery as well as Service Discovery and provides among others an interface for connection-oriented L2CAP, RFCOMM and OBEX.

### 3.6.1 The hunt for a JSR 82 implementation on J2SE

Currently, the main implementation under Linux is *avetanaBluetooth* from Avetana. *avetanaBluetooth*[4] is available for Windows, MacOS X and Linux. The Linux version is open source (licensed under GPL) and can be downloaded from Sourceforge[5]. On Linux, *avetanaBluetooth* accesses the BlueZ libraries. The official Bluetooth subsystem for Linux BlueZ[6] provides an open and stable implementation of the Bluetooth stack.

Under Windows there are three implementations: *avetanaBluetooth*, *BlueCove* and *aveLink Bluetooth SDK for Java* from Atinav. The main difference between them is the conformity to different Bluetooth stack: *avetanaBluetooth* to the Widcomm stack, *BlueCove* to the Windows XP SP2 Bluetooth stack and *aveLink* provides an own stack. The next difference is the price resp. the licensing policy: the Windows version of *avetanaBluetooth* is available to a low fee, *BlueCove*[7] is open source (licenced under LGPL) and *aveLink*[8] can be purchased to a high fee. *aveLink* is expensive because of the own implementation of the Bluetooth stack. It is essential to know that *BlueCove* does not provide a L2CAP interface.

## 3.7 Accessing the BTnodes with Bluetooth APIs

Several software APIs exist to access the BTnode over RFCOMM. Three options are listed to accomplish the access: a RFCOMM-only, a native and a platform independent solution.

[2]http://www.jcp.org/en/jsr/detail?id=82
[3]http://en.wikipedia.org/wiki/J2ME
[4]http://www.avetana-gmbh.de/avetana-gmbh/produkte/jsr82.eng.xml
[5]http://sourceforge.net/projects/avetanabt
[6]http://www.bluez.org
[7]http://sourceforge.net/projects/bluecove
[8]http://www.avelink.com/Bluetooth/Products/JSR-82

### The RFCOMM-only solution

Many operating systems like Windows, MacOS X and Linux provides virtual serial ports for RFCOMM. Unfortunately they will only work if the SDP layer is implemented on the communication partner side. As long as the Service Discovery is not replied by the BTnode, the creation of a virtual port will fail.

But there is a workaround solution on Linux. A virtual serial port is created when the appropriate entry exists in `/etc/bluetooth/rfcomm.conf`. By adding following lines

```
rfcomm69 {
  # Automatically bind the device at start-up
  bind yes;
  # Bluetooth address of the device
  device 00:04:3F:00:00:69;
  # RFCOMM channel for the connection
  channel 1;
  # Description of the connection
  comment "BTnode running bt-cmd-over-rfcomm";
}
```

a new device `/dev/rfcomm69` is created which can be accessed in the same manner as the serial port. This workaround also works well with the Java package `javax.comm`.

The benefit of this solution is that existing software, which used the serial port before, must not be rewritten. It is platform independent as well but works only with RFCOMM.

### The native solution

The API from the Bluetooth stack is directly used. Like all applications running on the BTnode the running program accesses the stack directly. This solution is platform dependent and Bluetooth stack dependent.

### The platform-independ solution

The standard Bluetooth API JSR-82 for Java provides a stream interface for RFCOMM. The following short code establishes a RFCOMM connection:

```
// connect to the Bluetooth hardware address 00:04:3F:00:00:69
//      at RFCOMM channel 1
StreamConnection conn = (StreamConnection)
                        Connector.open("btspp://00043F000069:1");
```

In order to begin the communication the appropriate streams have first to be opened.

```
InputStream in = conn.openInputStream();
OutputStream out = conn.openOutputStream();
// in.read() and out.write() is ready to use
```

A small Java program was written to access a BTnode over RFCOMM. It serves a simple command-line interface:

```
java RFCOMMTerminal <addr> <channel>
```

Data from the RFCOMM channel will be printed out and keyboard entries on the command-line are sent over RFCOMM. This Java program may be used alternatively to the virtual RFCOMM ports on Windows and MacOS X, as long as the SDP layer is not implemented in BTnut. The source code is stored under `proj/yuecelm/RFCOMMTerminal` into the BTnut software repository.

The benefit of this solution is the platform-independence. All devices running J2ME/J2SE and supporting JSR-82 are capable to execute the Java program with the same source code.

Chapter 4

# Analysing the power consumption

In order to get inside into the whole system, the BTnode will be analysed with the main focus on the power consumption. Considering the datasheets of the hardware components the power consumers and the voltage converters are examined separately. In the first subchapter the power consumers and their current flow will be characterised. The second chapter lists the available supply options. At last there is a setup description of the measurement system which will be used in the next chapters.

Some labels which appears in the text are taken 1:1 from the BTnode rev3.22 schematics. These schematics are attached in the appendix on page 61.

## 4.1 Current consumption of the microprocessor and other components

The BTnode has three main power consumers: the microprocessor ATmega128L, the Bluetooth module ZV4002, and the low-power radio module CC1000. Figure 4.1 shows the current consumption while the ATmega128L runs in the idle thread, the ZV4002 lingers in the standby mode and the CC1000 listens to a certain frequency.

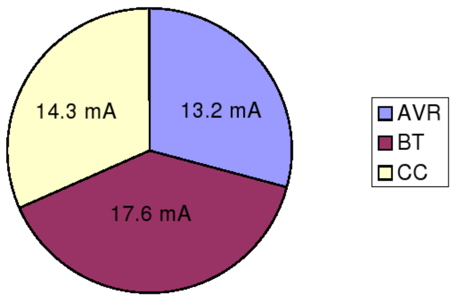


Figure 4.1: Current consumption of microprocessor and communication modules

Remember that the current consumption of all components, like the external SRAM and the flash memory for the ZV4002, are included in one section. Except the power latches they are supplied from one of these power lines: VCC_AVR, VCC_BT or VCC_CC. The CC1000 and any sensors, which are powered from the line VCC_IO, will not be further considered in this thesis.

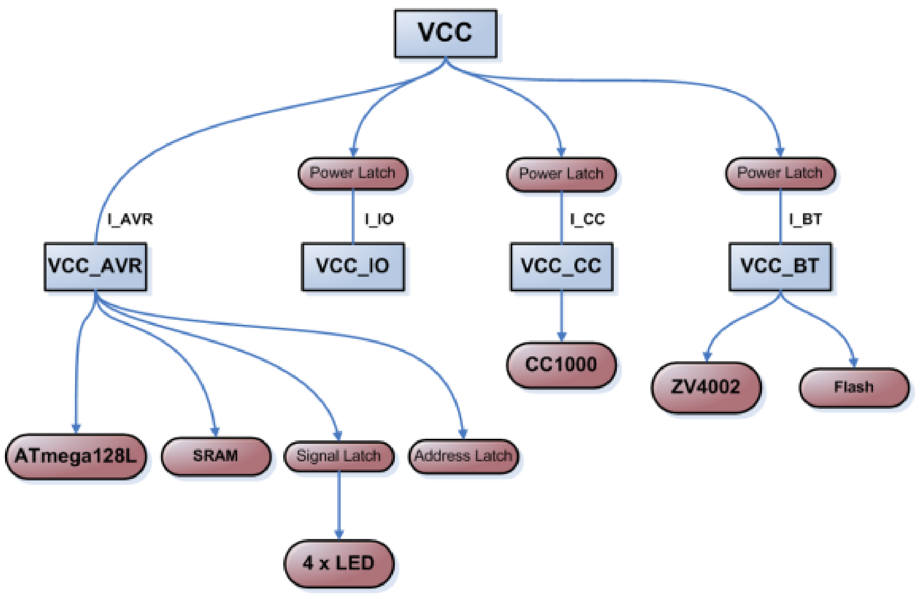


Figure 4.2: Voltage domains and power consumer components

Figure 4.2 shows the division of all components into one voltage domain whereas the voltage converters are covered separately in the next subchapter. So the current flow is clearly arranged. If e.g. the current I_AVR is measured, keep in mind that other components than the ATmega128L may affect an additional current flow. When a LED is powered it pulls 4 mA. The external SRAM for ATmega128L and the flash memory for ZV4002 consume each maximal 30 mA on read/write access.

## 4.2 Efficiency of voltage converters

The components have to be supplied by a voltage at 3.3 V. As illustrated in figure 4.3 there are three options to supply the BTnode, depending on the available voltage. The only option that does not pass a voltage converting process is possible with a 3.3 V regulated power supply through the VCC and GND pins. Thus no power loss occurs.

Alternatively a power source with a voltage between 3.6-5.0 V can be supplied through the VDC_IN and GND pins. The dropout regulator LT1962 from Linear lowers the voltage to 3.3 V. Some specific data about the efficiency will be delivered in subchapter 5.3 on page 33.

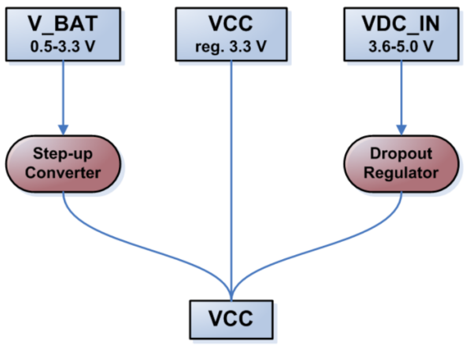


Figure 4.3: Available supply options

The last option is a power supply with a voltage between 0.5-3.3 V. This voltage range is suitable for batteries. Figure 4.4 shows the voltage progression on a battery-supplied BTnode with the Bluetooth module in standby mode.

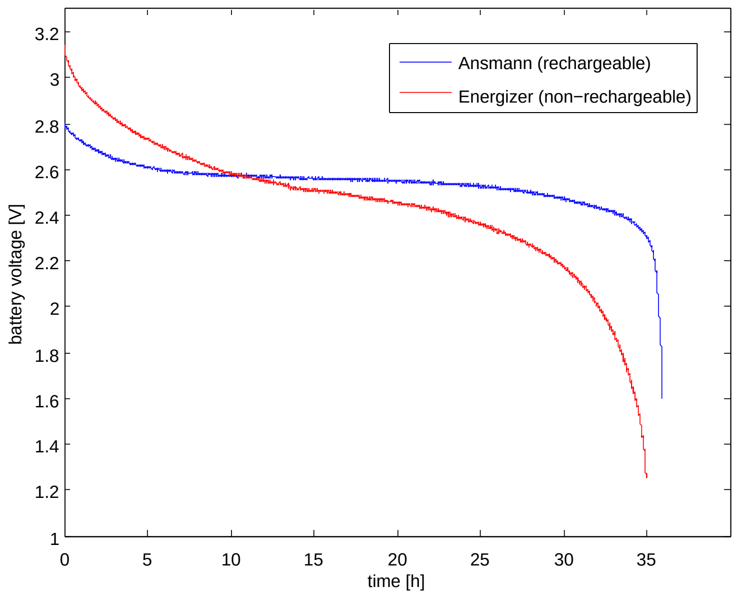


Figure 4.4: The voltage progression with battery supply

The step-up DC/DC converter LTC3429 from Linear raises the voltage to 3.3 V. The converter LTC3429 has an efficiency curve $\eta$ as showed in figure 4.5. The converter follows the formula $\eta \cdot P_{in} = P_{out}$ which has a bad behaviour by batteries with low voltage. Some specific data about the efficiency will be delivered in subchapter 5.3 on page 33. Remember that the batteries cannot be emptied out completely because they do not deliver arbitrary high currents on low voltage. If e.g. the battery voltage reaches 1.6 V and 60 mA are required on start-up, the converter wants to pull around 155 mA to reach the voltage 3.3 V. Because the batteries cannot deliver high currents the voltage converter compensates this issue by decreasing the output voltage. If the voltage goes below 2.7 V the Brown-out detector will be triggered and the microprocessor resets. Thus the current capacity indicated on batteries should never be used directly in lifetime calculations.

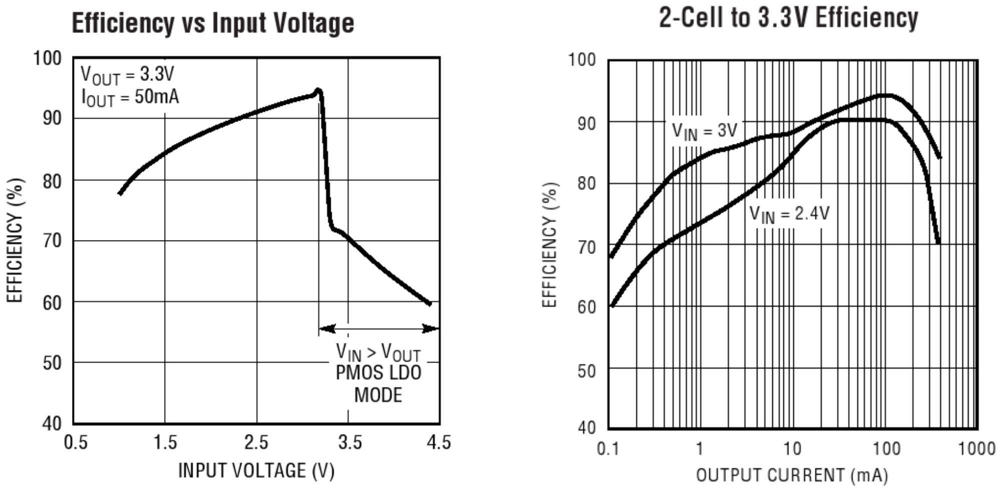


Figure 4.5: Efficiency of the step-up converter LTC3429

## 4.3 Measurement setup

The sleep modes on the microprocessor ATmega128L and the low-power modes on the Bluetooth module ZV4002 will be measured. The primary objective is to determine accurate data about the current consumption, the secondary is to detect a priori potential issues. Thus these measurement positions are of interest: close-by the power supply to calculate the overall power consumption, after the voltage converter to estimate the power loss, and after each power latch to measure the main power consumers separately. The measurement of the voltage and the current from the power supply is uncomplicated. Considering the BTnode schematics, the current after the voltage converter cannot be measured but the voltage VCC. Due to the measuring points R2, R17 and R26 it is possible to measure the voltages VCC_AVR, VCC_BT and VCC_CC and the currents I_AVR, I_BT and I_CC. Figure 4.6 shows a photo from the whole measurement system. To simplify and automate the measurement process a small program was written using LabVIEW. LabVIEW provides a GPIB driver to interface the measurement instrument. The user interface as illustrated in figure 4.7 should be self-explanatory. Last but not least figure 4.8 presents a detailed, labelled photo from the modified BTnode. This BTnode allows to measure also the currents I_AVR, I_BT and I_CC.

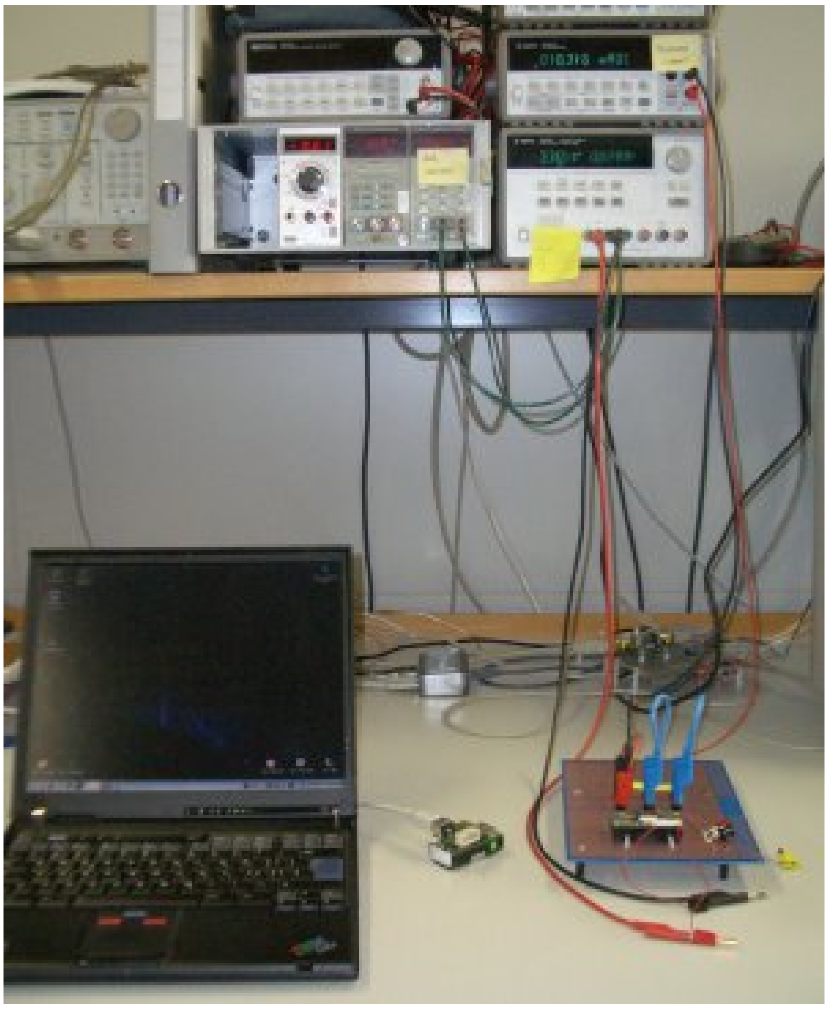
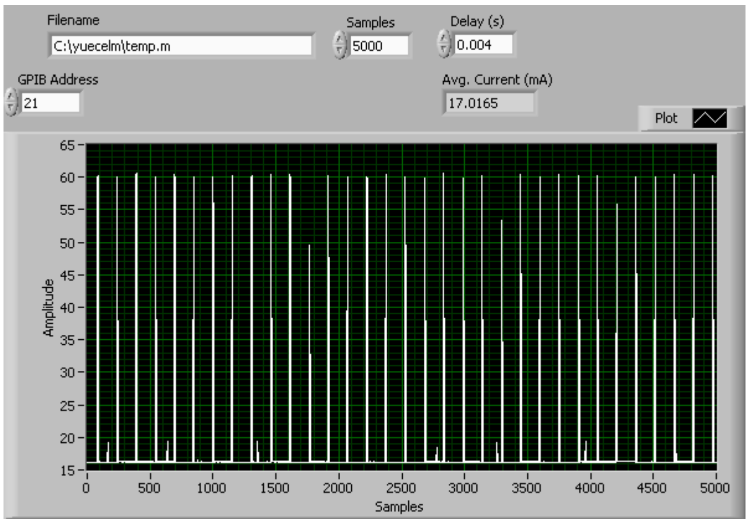


Figure 4.6: A photo from the measurement system



Filename: C:\yuecelm\temp.m
Samples: 5000
Delay (s): 0.004
GPIB Address: 21
Avg. Current (mA): 17.0165

Figure 4.7: The measurement user interface using LabVIEW

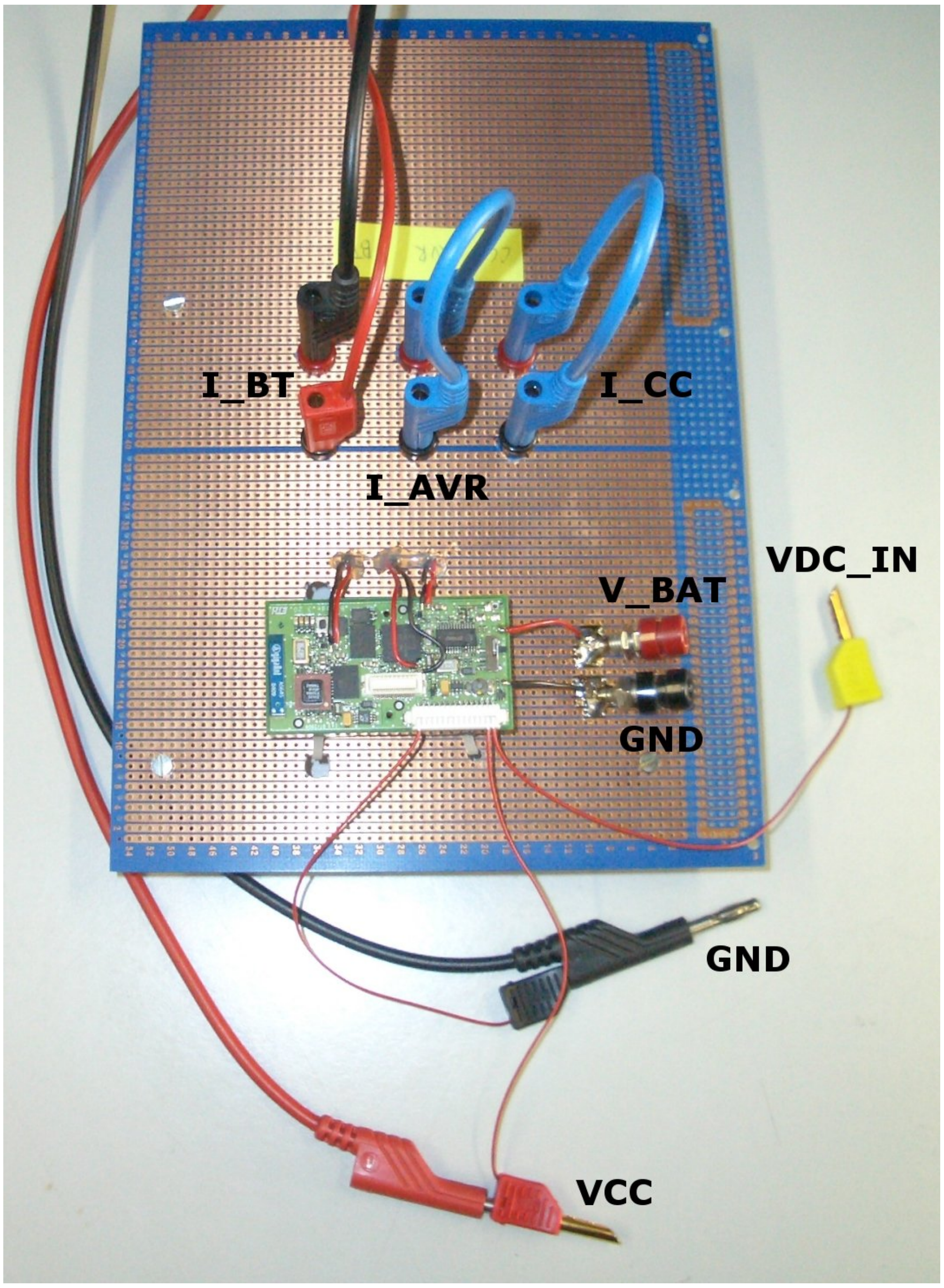


Figure 4.8: A labelled photo from the modified BTnode

# Chapter 5

# Applying the sleep modes

Before starting with the Bluetooth low-power modes, the power save potential of the microprocessor ATmega128L will be evaluated. The first subchapter covers the measurements about the current consumption of the ATmega128L in each sleep mode. The second subchapter describes how the sleep modes can be integrated into BTnut. Furthermore, the application range for some sleep modes will be listed. Gathered from the sleep mode measurements, the last subchapter includes some data about the efficiency of the voltage converters.

## 5.1 The measured current consumption in each mode

A small application `btnut/app/bttest/pm-test.c` was written to put the BTnode into a sleep mode. Both communication modules are explicitly powered off and the appropriate pins on the ATmega128L are tristated. During the start-up the actual sleep mode is indicated by a LED code. By resetting the BTnode the next sleep mode will be entered. The measurements are made with a couple of BTnodes from rev3.20 and rev3.22, and with different fuse settings. The supply voltage and current is measured. Figure 5.1 shows the measurement results with the three supply options as described in subchapter 4.2 on page 24. Figure 5.1 highlights the power consumption in each sleep mode with unpowered communication modules.

The data from the VCC 3.3V series is gathered to build a table with the current and power consumption in each sleep mode. The delays are taken from the ATmega128L datasheet [3].

| sleep mode | current | power | delay |
|---|---|---|---|
| | [mA] | [mW] | [# clock cycles] |
| NONE | 13.6 | 44.85 | 0 |
| IDLE | 4.97 | 16.39 | 0 |
| ADC | 2.42 | 7.989 | 0 |
| EXT_STANDBY | 1.56 | 5.135 | 6 |
| PWR_SAVE | 1.40 | 4.635 | 16K (+ 65 ms) |
| STANDBY | 0.812 | 2.6811 | 6 |
| PWR_DOWN | 0.183 | 0.6040 | 16K (+ 65 ms) |

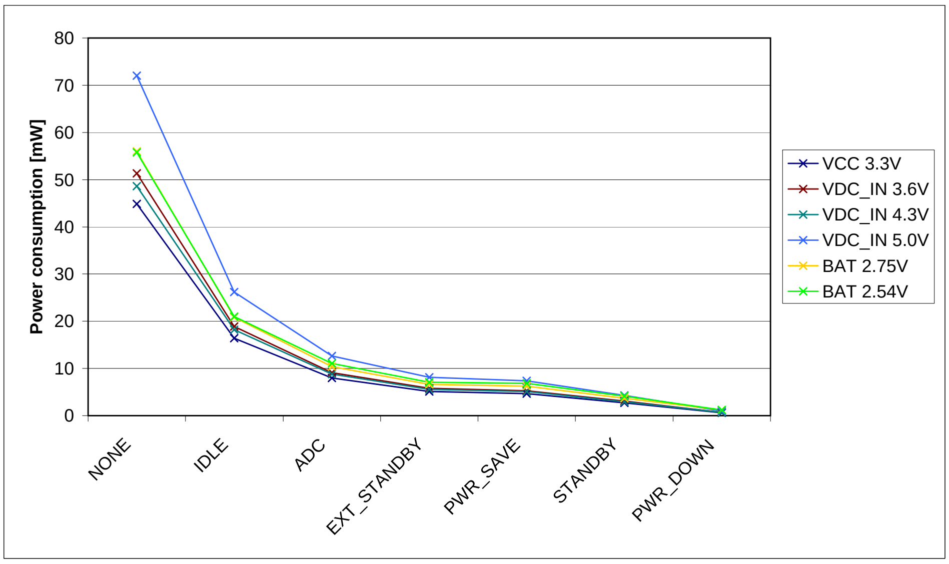


Figure 5.1: Power consumption in different sleep modes

## 5.2 Unleash the sleep modes in BTnut

The AVR Libc package provides a subset of the standard C library for AVR microcontrollers. Within the package, there is a library[1] to handle the sleep modes. Functions to set the sleep mode or put the microcontroller into the sleep state are supplied. The AVR Libc package is already included in BTnut. But BTnut offers a more elegant way to manage the sleep modes. To understand how it works, the thread management of BTnut will be first introduced.

BTnut implements cooperative multithreading. That means that threads are not bound to a fixed time slice. Thread changes only occur if the running thread explicitly yields the CPU or a hardware interrupt is triggered. The scheduler works on the principle that the thread with the highest priority always runs. The priority ranges from 0 to 254 where the lowest value indicates the highest importance.

BTnut provides a terminal command `nut threads` to list all threads. At least two threads are always printed: `main` and `idle`. In the `main` thread the main function of the application is running. Additional threads may be created with `NutThreadCreate`. A particular role takes the `idle` thread. It is assigned with the lowest possible priority. Thus it is runs only if no other threads are runnable. The `idle` thread has two main tasks. In an endless loop, it calls `NutThreadYield` and `NutThreadDestroy`. The first function tries to give up the CPU to another thread. The second function frees the memory of the previously killed threads. Afterwards the `idle` thread may put the CPU into a sleep mode. The sleep mode can be set with `NutThreadSetSleepMode`. By default the sleep mode is set to `SLEEP_MODE_NONE`. Additionally the following sleep modes are supported:

- `SLEEP_MODE_IDLE`

[1]http://www.nongnu.org/avr-libc/user-manual/group__avr__sleep.html

- SLEEP_MODE_ADC
- SLEEP_MODE_EXT_STANDBY
- SLEEP_MODE_PWR_SAVE
- SLEEP_MODE_STANDBY
- SLEEP_MODE_PWR_DOWN

In order to determine which sleep modes are feasible within the thread management, a small application `sleepandwork.c` was written. The source code is attached in the appendix on page 61. First it disables the LEDs and both communication modules. As next the sleep mode within the idle thread is setted. Then the application enters an endless loop with a sleep and a work task. Each task will take around one second. These both tasks results that the `main` and the `idle` threads are running alternatively.

This application is used by the next measurements. The BTnode is directly supplied over VCC with 3.3 V and the current through is recorded. One measurement cycle is done in each sleep mode. Figure 5.2 compares the current consumption between SLEEP_MODE_IDLE, SLEEP_MODE_ADC and SLEEP_MODE_EXT_STANDBY.

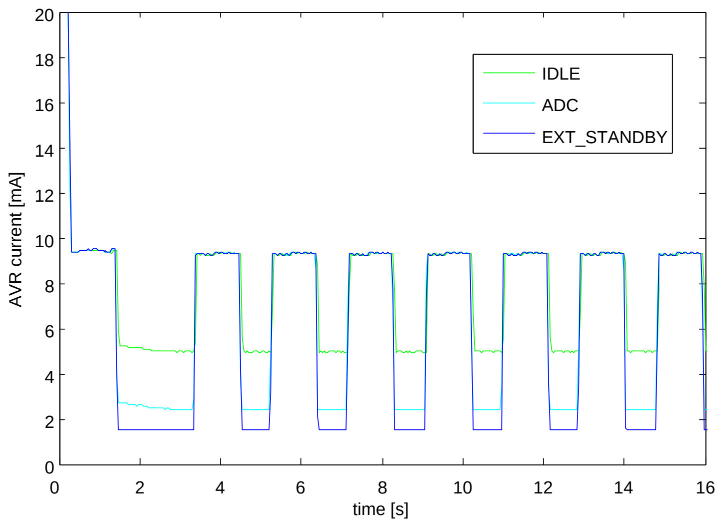


Figure 5.2: Comparing sleep modes IDLE, ADC and EXT_STANDBY

After a start-up time of circa 2.5 seconds, both tasks are running alternatively. If the CPU is in sleep state, it is clearly visible in the current curve.

Figure 5.3 compares the current consumption between SLEEP_MODE_IDLE and SLEEP_MODE_NONE. If the sleep mode is set to SLEEP_MODE_NONE, the current is much higher while residing in the sleep task. The `idle` thread accesses excessively the heap resp. the SRAM. These accesses considerably increase the current demand. This unwanted behaviour can be avoided by putting the CPU into a sleep state.

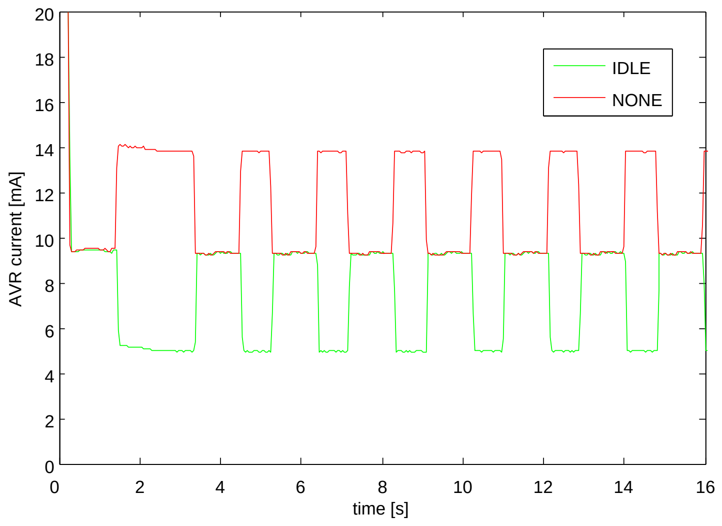


Figure 5.3: Comparing sleep modes IDLE and NONE

Figure 5.3 compares the current consumption between `SLEEP_MODE_IDLE` and `SLEEP_MODE_PWR_SAVE`. `SLEEP_MODE_PWR_SAVE` has a timing problem and is not ready to use in the thread management.

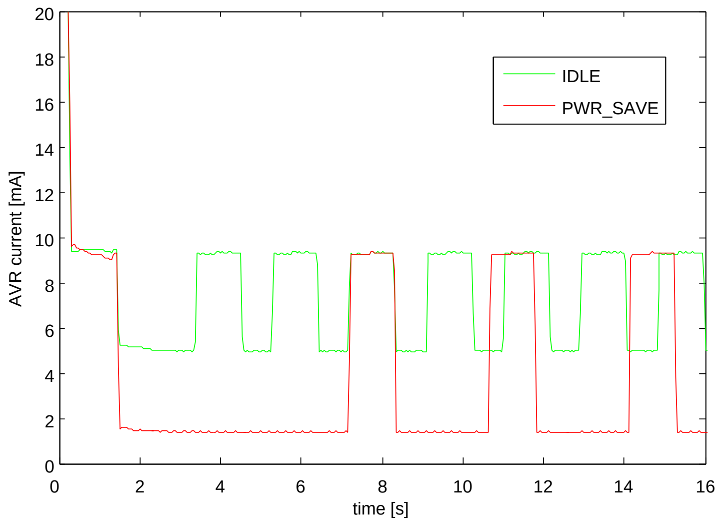


Figure 5.4: Comparing sleep modes IDLE and PWR_SAVE

The sleep modes `SLEEP_MODE_STANDBY` and `SLEEP_MODE_PWR_DOWN` are not applicable due to the lack of a suitable wake up source.

If the serial port or the Bluetooth module is used, `SLEEP_MODE_IDLE` is the only option. The CPU will wakes up by any UART communication. If the serial port and the Bluetooth module is not used, `SLEEP_MODE_ADC` and `SLEEP_MODE_EXT_STANDBY` come into further consideration.

If the flow control on UART is implemented, operative and enabled, other sleep modes are feasible. External interrupts, like on the RTS/CTS lines, are available in each sleep mode as a wake up source.

Because the sleep mode `SLEEP_MODE_IDLE` is applicable in each situation, it becomes to the standard in the thread management of BTnut. While the CPU is running in the `idle` thread, the power consumption is reduced down to 36.5 % comparing to `SLEEP_MODE_NONE`.

## 5.3 The measured voltage converter efficiency

Figure 5.1 on page 30 also indicates the efficiency of the voltage converters. The data from the VCC 3.3V series corresponds the power consumption without loss affected by the voltage converters. Thus it is taken as reference to determine the efficiency $\eta = P/P_{VCC}$.

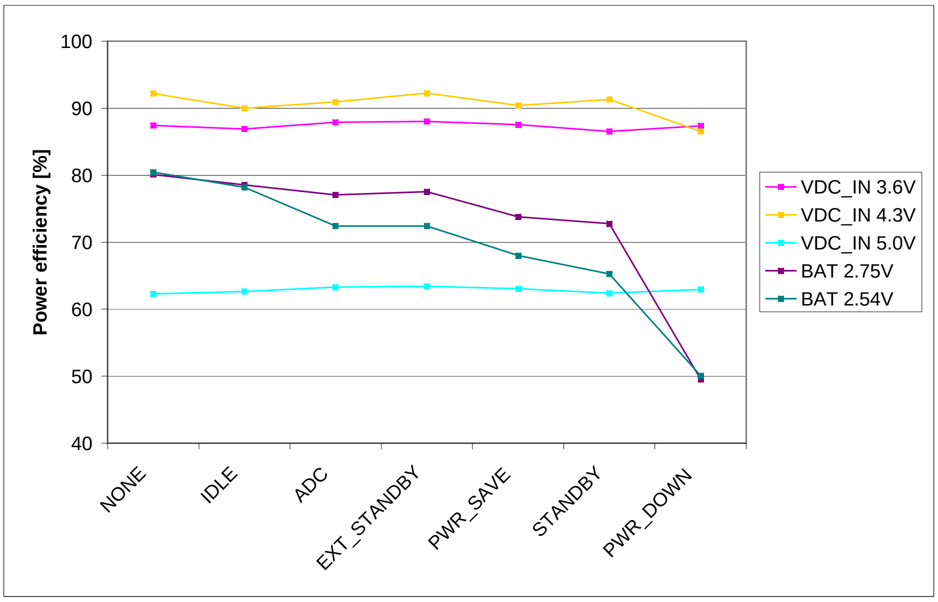


Figure 5.5: Efficiency curves of different supplies

As expected from a dropout regulator, the efficiency curve of VDC_IN is nearly constant. (5.1) is expected from a voltage divider but it cannot be applied to the efficiency curves of VDC_IN 3.6V and VDC_IN 4.3V. Input voltages close to the lower limit at 3.6 V seem to get a efficiency breakdown.

$$VDC\_IN_1 > VDC\_IN_2 \longrightarrow \eta_1 < \eta_2 \tag{5.1}$$

Considering the efficiency curves in figure 4.5 on page 26, the efficiency of the step-up converter in figure 5.5 is about 5-10 % lower than expected from the datasheet.

# Chapter 6

# Bluetooth link power control

Bluetooth provides low-power modes (hold, sniff, park) which trade throughput and latency for power. Because the Bluetooth stack of BTnut does not handle the low-power modes, the HCI layer will be extended in the first subchapter. In the second subchapter, these low-power mode commands are added to the terminal to provide a human-friendly control interface. In the third subchapter the results of [5] and [6] will be verified with additional measurements. Furthermore, these measurements may detect potential issues. At last, a library application in BTnut, the Connection Power Manager, will be presented which autonomously puts already established connections into sniff mode.

## 6.1 Extending the HCI layer of BTnut

The HCI commands for the low-power modes are not implemented in the Bluetooth stack. All the low-power commands are an essential part of the link policy commands. According to the Bluetooth Specification [2] the file in BTnut `btnut/btnode/bt/bt_hci_cmd_link_policy.c` will be extended with the following commands:

- HCI_Hold_Mode
- HCI_Sniff_Mode
- HCI_Exit_Sniff_Mode
- HCI_Park_State
- HCI_Exit_Park_State

But just implementing the HCI commands is not sufficient. Generally, after sending a HCI command, the controller responds with a HCI event. In context to the low-power commands the Bluetooth module returns a mode change event with the current mode. Thus after extending the file `btnut/btnode/bt/bt_hci_event.c` the Bluetooth stack handles the mode change events, and is able to track the mode on each connection.

## 6.2 Extending the Bluetooth command terminal

BTnut includes libraries to offer a terminal on the serial port or — since the work described in the third chapter — on RFCOMM. `btnut/btnode/terminal/bt-cmds.c` already contains Bluetooth commands e.g. to inquiry or connect other Bluetooth devices. A new file `btnut/btnode/terminal/bt-extra-cmds.c` is created to prevent a blow up of the size of existing applications. The following commands are added to the terminal library:

```
ebt scan <inquiry_scan> <page_scan>
ebt hold <con_handle> <interval>
ebt sniff <con_handle> [<interval> <attempt> <timeout>]
ebt park <con_handle> [<beacon_interval>]
```

The first command is used to enable `1` or disable `0` both scans. `con_handle` is an identifier of the connection. All other parameters are adopted from the Bluetooth Specification [2]. They specify either the number of receive slots or the number of slots. If only `con_handle` is specified, the appropriate low-power mode will be exited. Last but not least `bt_extra_cmds_register_cmds()` must be called in the application to provide these commands on the terminal.

The next change is applied on the terminal command `bt contable` which prints out a list of open connections. It will be a valuable feature to additional indicate the actual mode. Thus a new function `bt_hci_local_mode_discovery` is implemented to poll the Bluetooth stack about the mode. This function is added to `btnut/btnode/bt/bt_hci_local_cmds.c`. A print out sample of the extended terminal command follows:

```
[bt-cmd@btnode]$ bt contable
Number of open connections: 2
Handle Addr              State        Mode
    01 00:04:3f:00:00:65 my slave     sniff (2048)
    04 00:04:3f:00:00:ad my slave     active
```

## 6.3 Bluetooth link measurements

In order to measure the current consumption behaviour of the ZV4002, the Bluetooth terminal application `btnut/app/bt-cmd` is used to perform the Bluetooth functions. The measurements are only done with one modified BTnode as showing in figure 4.8 on page 28. The Bluetooth firmware version 6.3 from Zeevo runs on the ZV4002 where the BTnut terminal command `bt version` returns `2 00C9 2 0012 003D`. The voltage, which will be assumed constant during the experiments, was previously measured at 3.3 V. The multimeter is set to operate at 250 samples/s which measures the current I_BT.

### Standby state

After the Bluetooth module ZV4002 is powered, the module will reach the standby state. Per default, the inquiry scan and the page scan are enabled. Thus the module is ready to be discovered and connected from other Bluetooth devices. When both scans are disabled (`ebt scan 0 0`), another current baseline can be observed. Figure 6.1 shows the ZV4002-specific current consumption of a connection with both scans enabled and disabled.

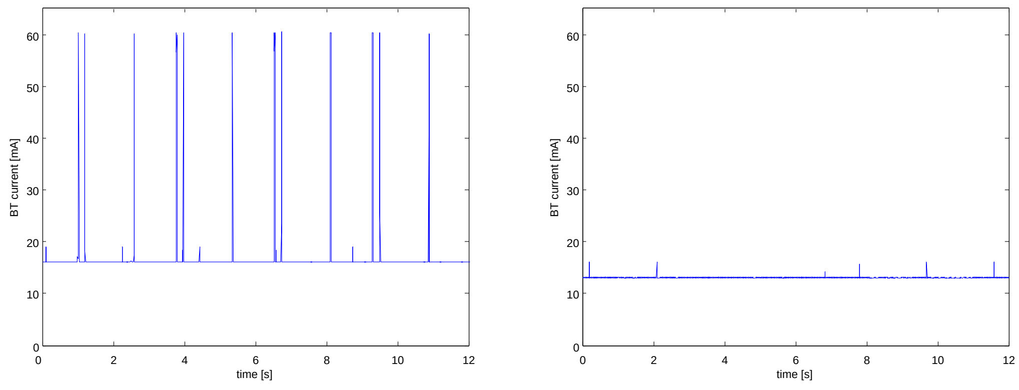


Figure 6.1: Bluetooth current consumption in standby state, both scans enabled (l.) and both scans disabled (r.)

The two periodical peaks at 60 mA are the currents from the scans. But one scan has the halved frequency. If the peaks are not counted the baseline current of 16.0 mA before is reduced to 13.1 mA with disabled scans. Additionally figure 6.2 shows the current consumption of a connection with only one scan enabled.

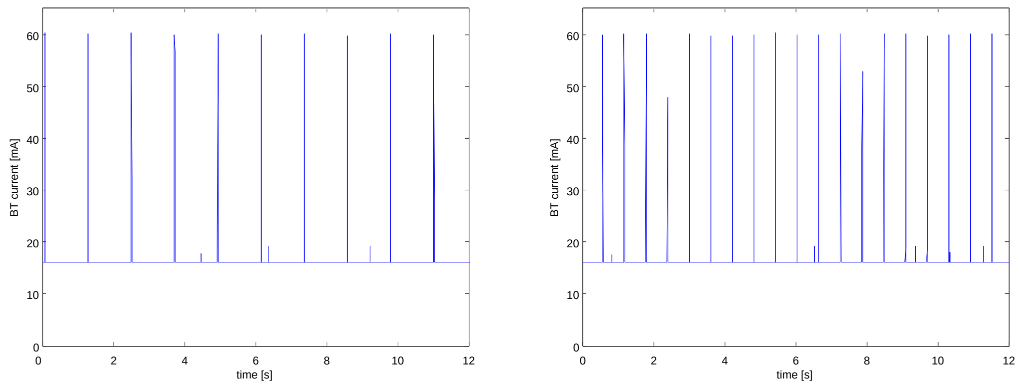


Figure 6.2: Bluetooth current consumption in standby state, only inquiry scan enabled (l.) and only page scan enabled (r.)

According the default of the Bluetooth specifications [2] the inquiry scan has a period of 2.56 seconds whereas the page scan has a period of 1.28 seconds. Disabling one scan reduces the current consumption by 0.5 mA. Observed by the two different current baselines at 16.0 mA and 13.1 mA, the ZV4002 seems to go into a sleep state if both scans are disabled. Because it is desired to be the most time discoverable and/or connectable, the inquiry and page scan are left enabled in the next measurements.

### Connected state

For clarification: we denote with *master role connection* a connection to a slave and with *slave role connection* a connection to a master. Figure 6.3 compares the current consumption of a slave role connection and a master role connection. The slave role current baseline at 30.7 mA is much higher than the master one at 18.0 mA.

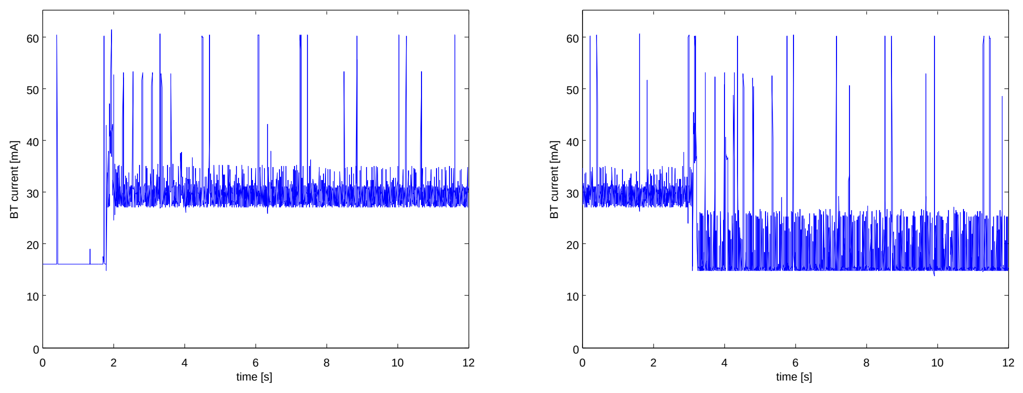


Figure 6.3: Bluetooth current consumption, slave role connection established (l.) and role change to master role connection (r.)

### Sniff mode

Figure 6.4 highlights the bursty behaviour of sniff mode, with periodical peaks around 40 mA every *Sniff Interval* (SI) slots. With at least one scan enabled two current baseline are observed. These values are lower than the active ones and they are equal to one standby current baseline (16.0 or 13.1 mA). In sniff mode, there is no significant difference in the current consumption curve between master and slave role connections.

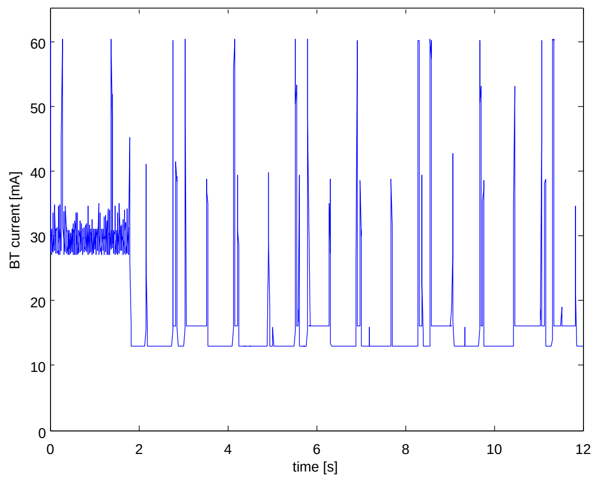


Figure 6.4: Bluetooth current consumption, change mode of a slave role connection to the sniff mode

### Distance

Figure 6.5 shows the current consumption while two connected devices are moving away. Figure 6.5 exposes that the ZV4002 provides a power control which is affected by the link quality. Hence connections across long distances should be avoided. Otherwise the current goes up to 45 mA on a master role connection.

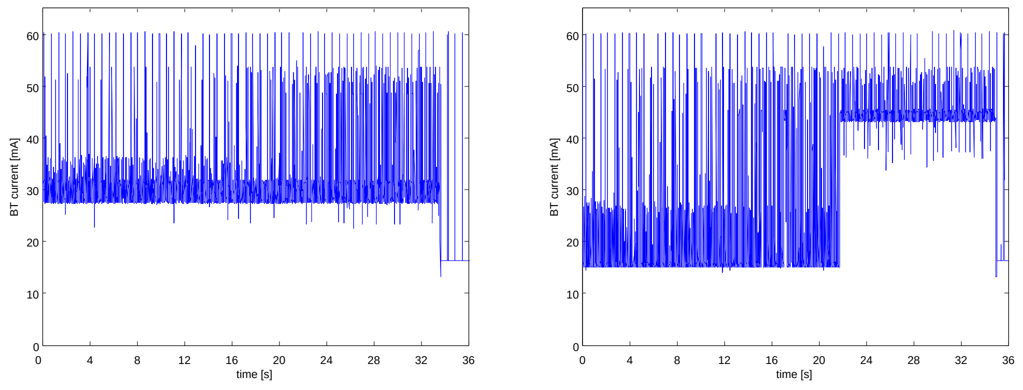


Figure 6.5: Bluetooth current consumption, impact of increasing distance, slave role connection (l.) and master role connection (r.)

### Park state

Figure 6.6 shows the current consumption of a slave connection during the mode change from active to park. The park state should present the lowest current consumption. Enter the park state seems to work fine but unparking is not feasible. After enter the park state, the ZV2004 lingers about one minute in a blocked and undefined state. Thereafter, the connection is lost by returning an meaningless error message "Unknown Error“. This issue was not observed with the older Bluetooth firmware version 6.2.

Figure 6.7 and the following table presents the current consumption in different states or modes. Each value is the average of 2-5 measurement passes.

| mode/state (default: both scans on) | average current consumption [mA] |
|---|---|
| standby | 17.0 |
| standby (page_scan off) | 16.5 |
| standby (inq_scan off) | 16.5 |
| standby (inq_scan off, page_scan off) | 13.1 |
| connected (master role) | 18.0 |
| connected (slave role) | 30.7 |
| sniff (SI 256, SA 4, ST 4, Slave) | 16.1 |
| sniff (SI 1024, SA 14, ST 14, Slave) | 15.9 |
| sniff (SI 256, SA 4, ST 4, Master) | 16.2 |
| sniff (SI 1024, SA 14, ST 14, Master) | 15.5 |

As far as the current consumption is concerned, the specifications in the ZV4002 datasheet are unreliable. Zeevo also claims in the datasheet to provide a deep sleep

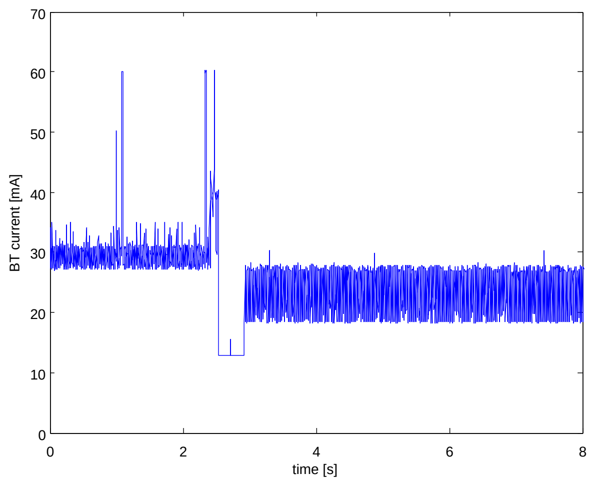


Figure 6.6: Bluetooth current consumption, change mode of a slave role connection to the park state

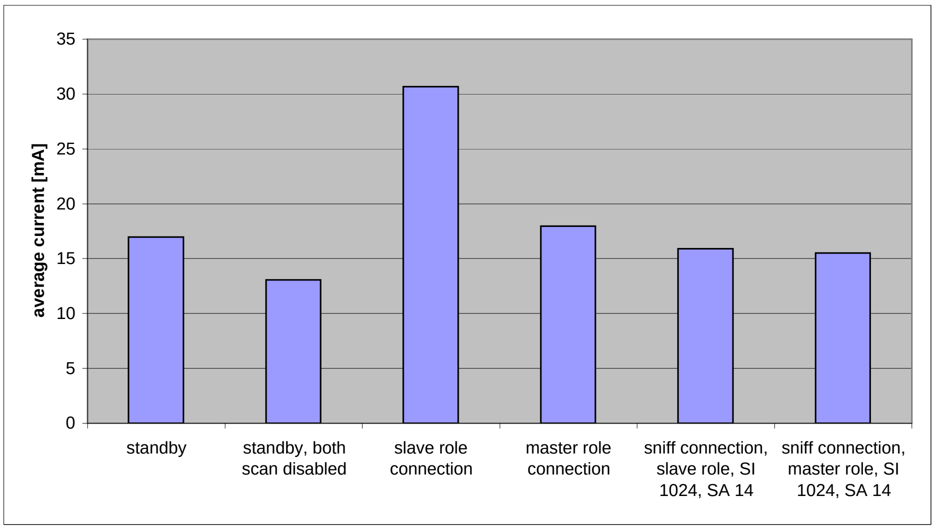


Figure 6.7: Average current consumption in different states/modes

state but a current consumption in $\mu$A scope was never observed.

If both scan are disabled, the current consumption of one sniff link is proportional to $(SA/SI)$. If at least one scan is enabled, there is a certain probability that the Bluetooth module goes into the sleep state between the scans and sniff intervals. The higher probability, the more power can be saved. Figure 6.9 on page 42 shows the average current consumption with fixed sniff interval and varied sniff attempt values.

## 6.4 Implementing the Connection Power Manager

The Bluetooth stack of BTnut tracks each connection. Open connections are listed with the terminal command `bt contable`. The work routine of the *Connection Power Manager* (CPM) is very simple. On the basis of periodical accesses to the connection table of the stack, the manager tries to put active links to the sniff mode. Because the sniff mode change can be initiated on master and slave side, the CPM puts either slave role connections or master role connections to the sniff mode. Additionally the maximum number of sniff links can be passed as well as the sniff parameters SI, SA and ST. Figure 6.8 shows a state model of the CPM.

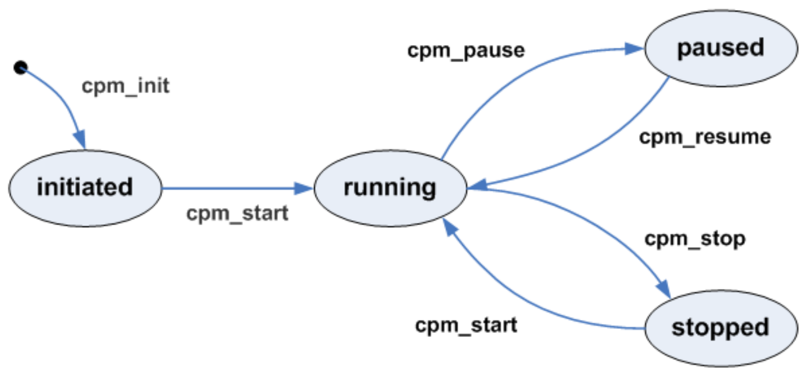


Figure 6.8: State transition model of the Connection Power Manager (CPM)

According to figure 6.8 the CPM provides five external functions `cpm_init`, `cpm_start`, `cpm_pause`, `cpm_resume` and `cpm_stop` and has four functional states:

**initiated** the CPM stack is allocated and ready

**running** autonomously puts active links into sniff mode

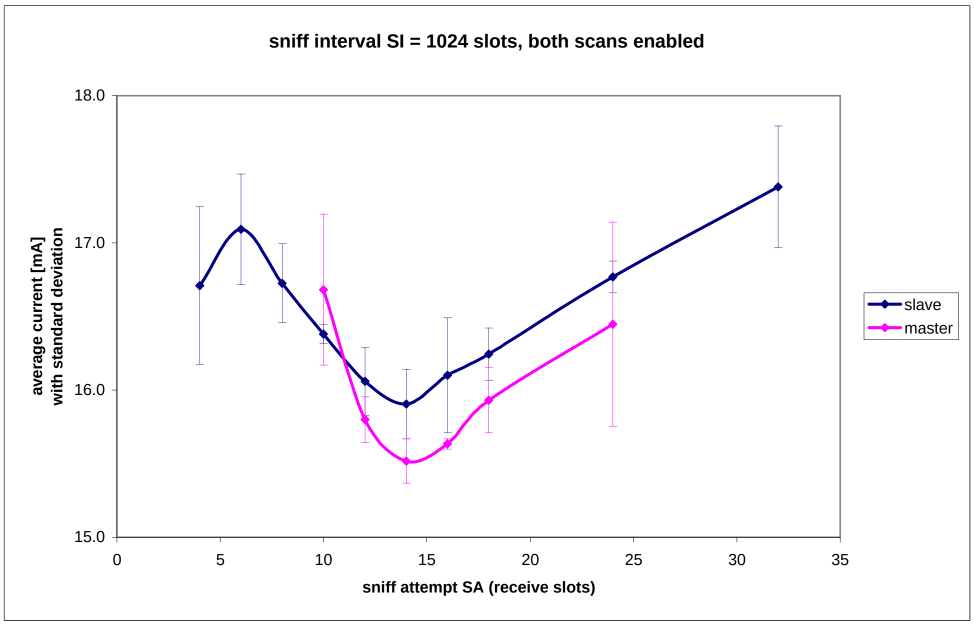
**paused** pauses the mode changes

**stopped** returns all sniff links back to the active mode

The *Class of Device* (CoD) may be passed with `cpm_init` to filter the sniff candidates. There are no checks about valid transitions inside the CPM. Thus it is strongly recommended to preserve the state transitions as shown in figure 6.8.

In order to control the CPM in the same fashion as the low-power commands described in subchapter 6.2, the following command is added to the Bluetooth terminal library `btnut/btnode/terminal/bt-extra-cmds.c`:

```
ebt cpm <start: 1, stop: 0>
```



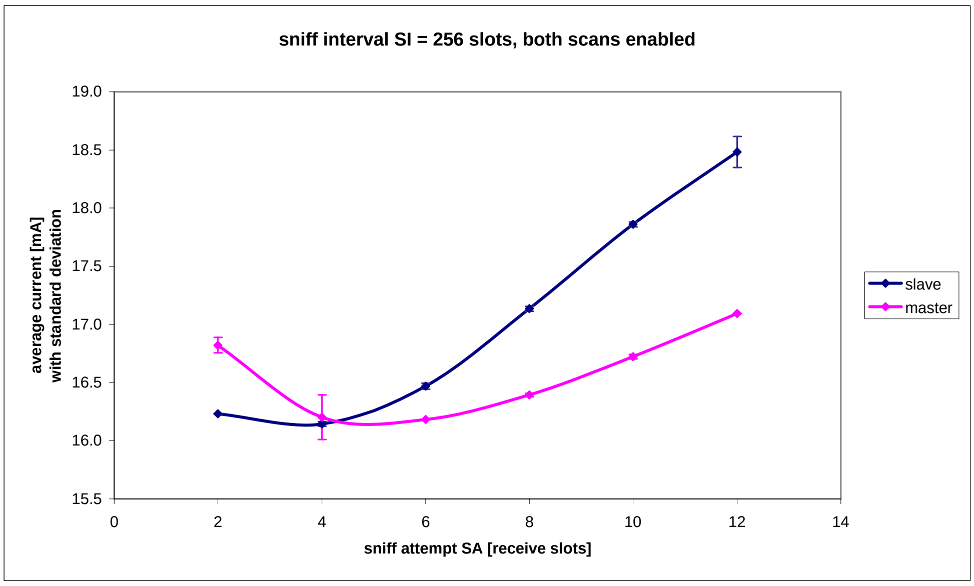


Figure 6.9: Average current consumption with fixed sniff intervals, SI = 1024 slots (t.) and SI = 256 slots (b.)

# Chapter 7

# A low-power connection manager for JAWS

In [7] an implementation called JAWS is described based on the BTnode platform. JAWS provides a connection and transport manager to automatically build up a topology and offers multihop communication within the Bluetooth scatternet. With both managers JAWS delivers multihop services such topology display or code distributing. JAWS will be extended to maintain Bluetooth connections in low-power mode sniff.

In the first subchapter the limitations by the Bluetooth module ZV4002 are shown. The second subchapter examines the connection managers delivered by JAWS. Due to the strong limitations of ZV4002, the new connection manager is described in the third subchapter which can handle the limitations. The fourth subchapter presents current measurements from a JAWS node. At last, the fifth subchapter shows stability tests about the sniff connections.

## 7.1 Limitations by the Bluetooth module ZV4002

The Bluetooth module ZV4002 from Zeevo has strong limitations. Considering the HCI reference guide from Zeevo, the following restrictions are present in the Bluetooth firmware implementation:

- The Zeevo device can only sniff in one piconet. Hence if a sniff link already exists in another piconet the new proposed sniff will be rejected.
- If the Zeevo device has multiple sniffed links they must have the same sniff interval.

With other words the ZV4002 supports only one slave role connection in sniff mode. This is a significant restriction because at least one slave role connection in active mode pulls the current of the ZV4002 to the maximum at 30.7 mA. In respect to minimise the current consumption this situation should be, if possible, avoided.

Additionally there are more limitations of the ZV4002 which are not denoted in any document. During the sniff mode a role change is not possible. The connection has to exit the sniff mode, thereafter the role can be changed. After changing a fourth master connection to the sniff mode, the ZV4002 will remain in a blocked state.

These strong limitations, the restriction to one slave and three master role connections for the sniff mode, complicates the power saving task. In order to put all connections to the sniff mode the following topologies are targeted:

- a chain
- a circle
- a tree
- a circle with one or more appended chains or trees

If each node has only one slave role connection there is a role polarisation in one direction. Figure 7.1 shows some examples. A node denotes a Bluetooth device, the arrow source the master role of a connection, the arrow head the slave role of a connection.

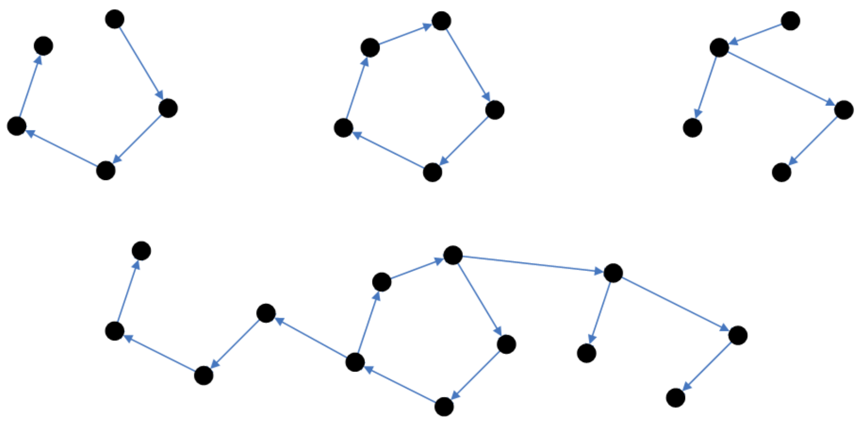


Figure 7.1: Topology examples which fits to the ZV4002 restrictions

## 7.2 Why not using an existing connection manager?

Due to the strong limitations of ZV4002 described in last subchapter, the existing connection managers are examined to their adaptability. There are two connection managers contributed by JAWS: the *Tree Connection Manager* (TCM), and the *XTC Connection Manager* (XCM).

The TCM builds up a tree topology as showed in figure 7.2. The TCM has the advantage that no loops are possible. The upperlaying transport manager does not receive any broadcast messages twice. However, this lets appear one of the disadvantages. If one link is lost the whole subtree behind the link is temporally unreachable. The TCM seems to be a suitable candidate for the sniff task. But the tree is not polarised, the parent node is either master or slave. The tree algorithm does not consider the role of the parent and child nodes. Thus it is not easily adaptable to the

ZV4002 limitations. More information about the TCM can be found in the paper [7] and the master thesis [8].

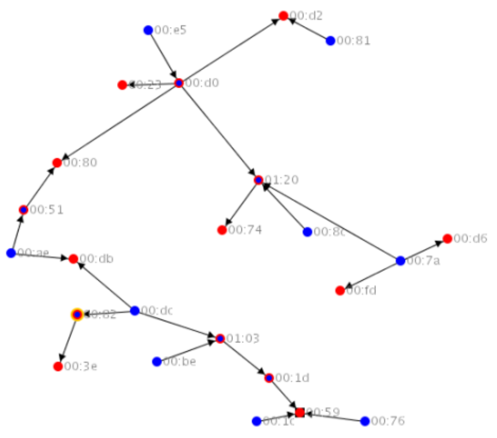


Figure 7.2: Topology example from the Tree Connection Manager

The XCM builds up a mesh topology. Even in dense environments the XTC algorithm promises to establish a sparse network graph as showed in figure 7.3, avoiding long distance communication links. This seems to be a good tradeoff between connectivity and spareness which makes this algorithm very attractive. Unfortunately the XTC algorithm provides no option to either set the maximum grade of a node or limit the number of each role connections. The XTC algorithm does not consider the role of the neighbour nodes. Thus, the XCM is also no candidate for the ZV4002. More information about the XCM can be found in the master thesis [9] from Kevin Martin.

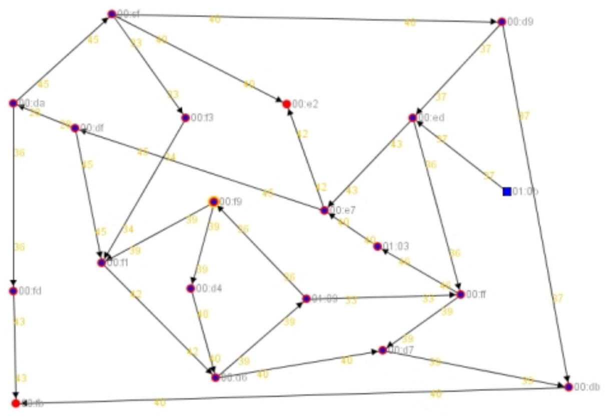


Figure 7.3: Topology example from the XTC Connection Manager

Due to the used algorithms both connection managers require some data exchange with the neighbours. It is unavoidable that many connections are opened, and closed right after. Both algorithms are also not considering the valuable Bluetooth properties discoverable and connectable. By enabling or disabling the inquiry scan or the page scan, the properties discoverable and connectable may be changed at any time. They might be powerful resources to optimise the construction of an ad-hoc network.

## 7.3 Implementation of a new connection manager

A new connection manager has to be implemented which handles the limitations described in subchapter 7.1. The best solution will be if the number of slave and master role connections can be limited in a flexible fashion. In this thesis, a connection-slot based solution is implemented which allows the user to set the appropriate number of slave and master role connections.

If no role changes occur, a slave role connection can be viewed as an incoming connection, a master role connection as an outgoing connection. These connections fit in a sight of connection slots which have to be filled for each node. The *Local Connection Manager* (LCM) adapts the principle of connection-slots. The work routine of the LCM is simple. One task establishes the connections to other JAWS nodes until all outgoing slots are filled. The other task is responsible for incoming connections. The reject policy for incoming connections is realised over two mechanisms: disable the scans or reject the connection within the paging stage. E.g. if all slots for incoming connections are filled, the inquiry scan will be disabled and the JAWS node will no more discoverable by the neighbours. A topology example of the LCM with the limitations of the ZV4002 is illustrated in figure 7.4. Figure 7.9 on page 50 shows a topology example of the LCM using the GUI interface of JAWS.

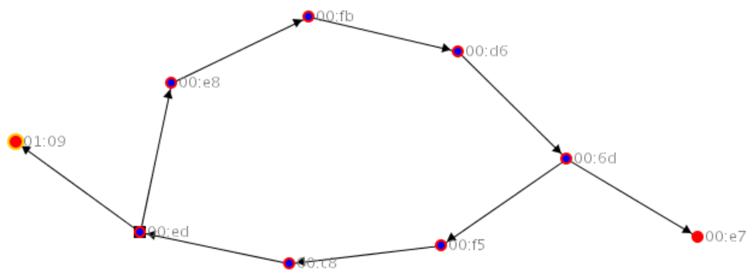


Figure 7.4: Topology example from the Local Connection Manager (LCM)

The simple algorithm as showed in the appendix on page 63 is used to establish connections to other JAWS nodes. In contrast to the other two connection managers, this algorithm requires no communication with its neighbours. The inquiry and page process is sufficient to construct a topology. A blacklist is maintained to temporally prevent the establish of connections to the listed devices. A device is blacklisted if e.g. the device is paged too much without success. In order to reduce the power consumption, the CPM described in subchapter 6.4 on page 41 is included in the LCM and the LCM provides terminal commands to start or stop the CPM.

The LCM is also capable to permit Bluetooth devices which are not operating as JAWS nodes the access to the JAWS network. Thus devices like PDAs or mobile phones can benefit from the multihop network. The LCM can also limit the connections to these devices. This feature is not explicitly provided by the other connection managers TCM and XCM. The LCM will try a role change from a slave to a master connection. Hence the high power consumption of a slave role connection is shifted to the communication partner.

The LCM is primary designed to test the stability and behaviour at the specified limit number of connections. In opposition to the TCM and the XCM, the LCM cannot guarantee a safe and parallel topology construction. Especially, if the limit is set to one slave role connection, the LCM cannot avoid the construction of two or more circles. Additionally, the current implementation of LCM does not track or indicate the link quality (RSSI) of a connection.

## 7.4 Current measurements on JAWS nodes

The current I_BT is measured to determine the power consumption benefit of sniff connections. The LCM is limited to one master role connection and two slave role connections. Figure 7.5 shows the current consumption with and without CPM. In the first case, all three connections are in sniff mode. In the second case, all three connection are in active mode.

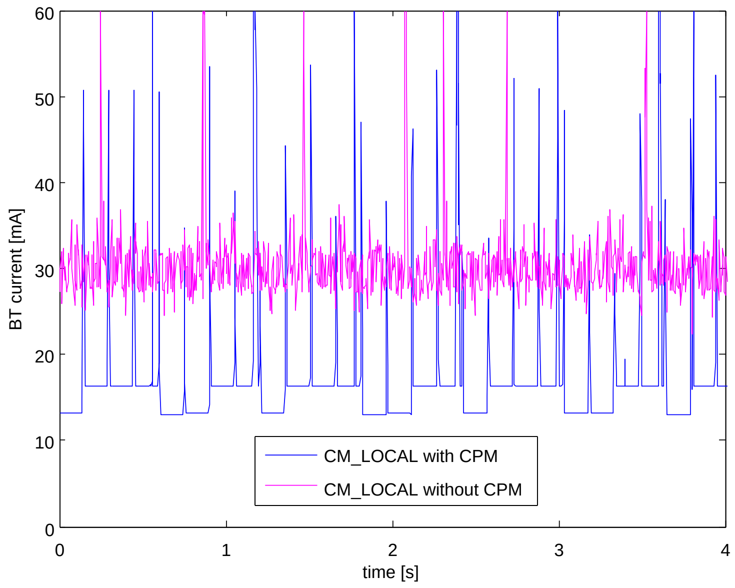


Figure 7.5: Comparing the BT current consumption on a JAWS node with and without the Connection Power Manager (CPM)

The average current without CPM is located at 30.3 mA whereas the average current with CPM is located at 17.2 mA. There is a power save potential around 57 %.

The current curve without CPM is characteristical if the Bluetooth device maintains at least one slave role connection. A similiar curve was observed in figure 6.3 on page 38 with the slave role connection.

Next, the current I_AVR is measured to determine the power save potential on the microprocessor with JAWS and the sleep mode IDLE. Figure 7.6 shows the current consumption of JAWS and compares the behaviour between sleep mode NONE and IDLE.

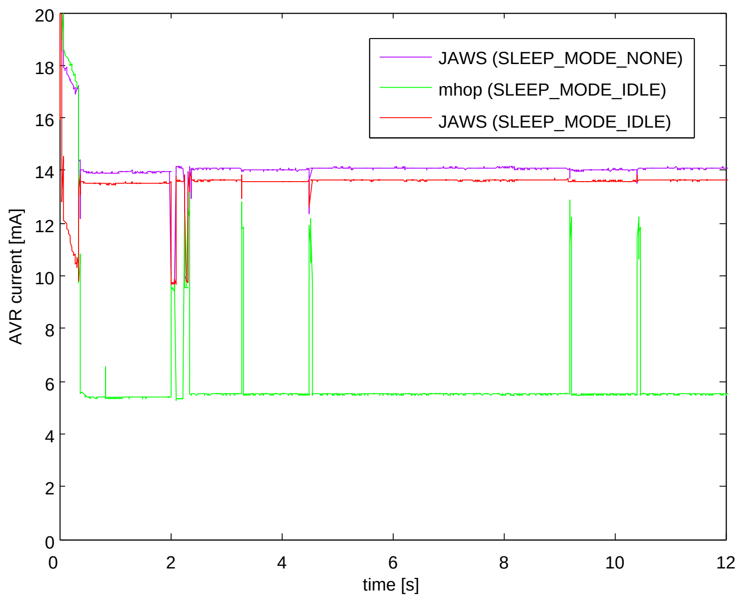


Figure 7.6: Comparing the AVR current consumption between JAWS and mhop application

Unfortunately, JAWS does not present the expected results. JAWS will not enter the idle thread. As reference, the application `btnut/app/mhop` is also included in the measurement as shown in figure 7.6. `btnut/app/mhop` provides a subset of the functionality of JAWS and only offers a connection and transport manager with a multihop blink command. The behaviour from JAWS is no more observed in `btnut/app/mhop`.

## 7.5 Stability tests

The small Java program `StabilityLog.java` is used to determine the stability of the Bluetooth network in sniff mode. The source code of `StabilityLog.java` is attached in the appendix on page A. The terminal command `coninfo` returns the connection table from each node in the network. Nine JAWS nodes are positioned for the measurements. Figure 7.7 shows one measurement without CPM and three measurements with CPM. The CPM will try to put all connections to the sniff mode.

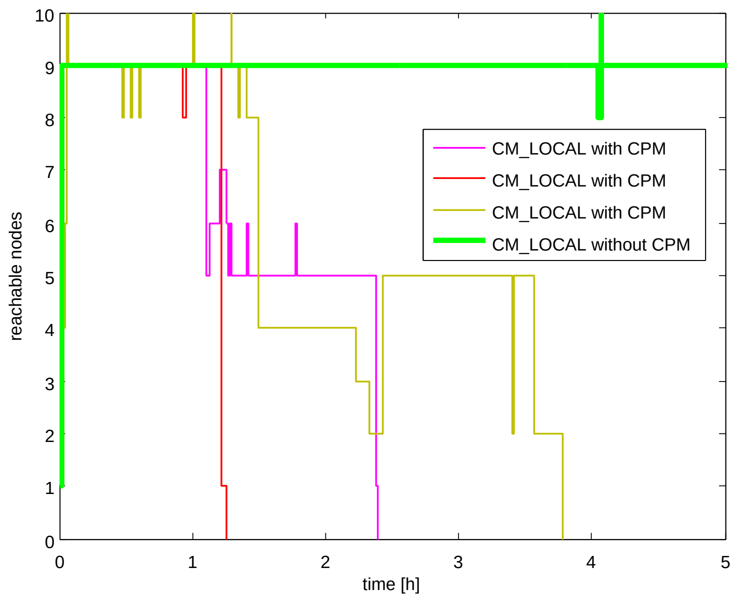


Figure 7.7: Stability tests of a JAWS network with and without CPM

The JAWS nodes with CPM enabled are no more reachable after a few hours. Further investigations arises that the ZV4002 no longer responds to the Bluetooth stack.

If these measurements are examined more precisely, an interesting fact can be observed. Figure 7.8 shows a detailed view, and additionally the number of sniff connections.

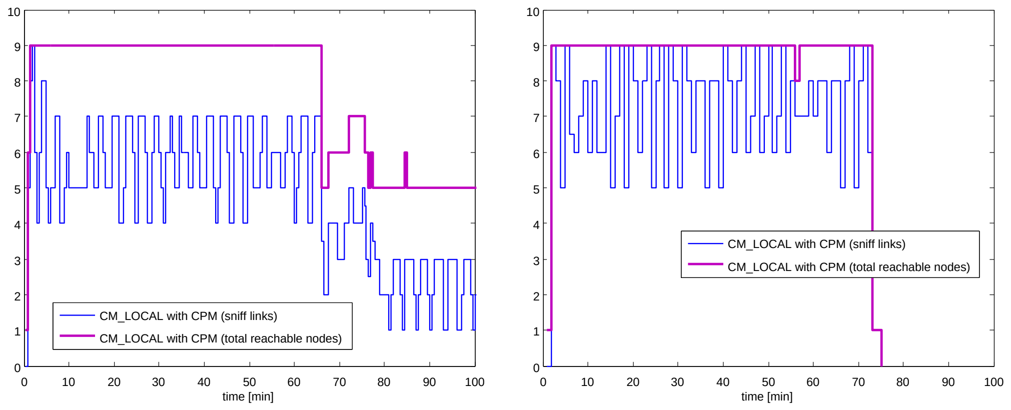
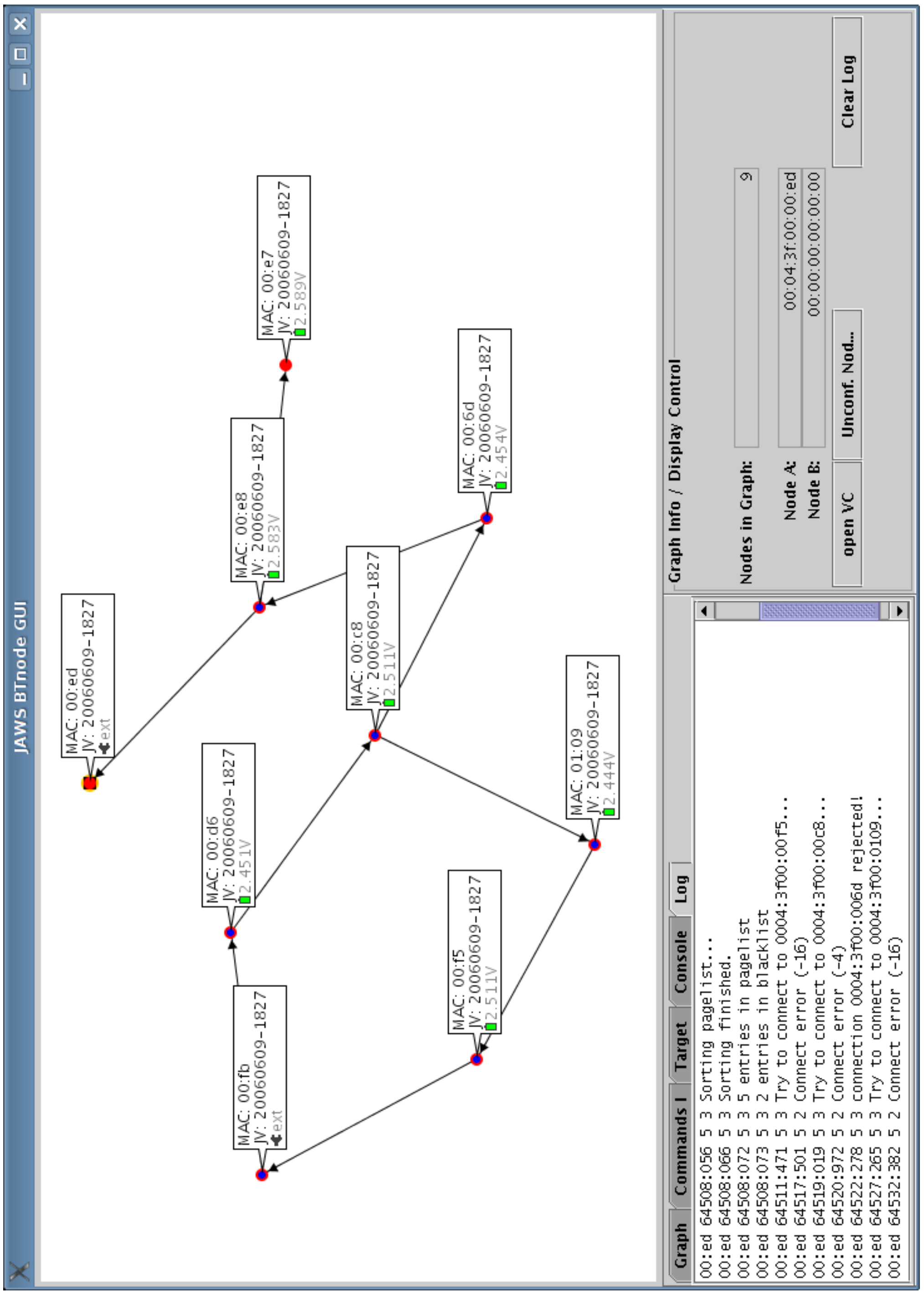


Figure 7.8: Stability tests of a JAWS network with CPM

An unstable behaviour is observed. The ZV4002 cannot steady hold the sniff connections. The left graph is generated with `coninfo` requests each 30 seconds. The right graph is generated with `coninfo` requests each 60 seconds.



Nodes in Graph: 9
Node A: 00:04:3f:00:00:ed
Node B: 00:00:00:00:00:00

```
00:ed 64508:056 5 3 Sorting pagelist...
00:ed 64508:066 5 3 Sorting finished.
00:ed 64508:072 5 3 5 entries in pagelist
00:ed 64508:073 5 3 2 entries in blacklist
00:ed 64511:471 5 3 Try to connect to 0004:3f00:00f5...
00:ed 64517:501 5 2 Connect error (-16)
00:ed 64519:019 5 3 Try to connect to 0004:3f00:00c8...
00:ed 64520:972 5 2 Connect error (-4)
00:ed 64522:278 5 3 connection 0004:3f00:006d rejected!
00:ed 64527:265 5 3 Try to connect to 0004:3f00:0109...
00:ed 64532:382 5 2 Connect error (-16)
```

Figure 7.9: Screenshot from the JAWS-GUI with nodes using the Local Connection Manager (LCM)

# Chapter 8

# Conclusions

## 8.1 Conclusions

As first, the RFCOMM device driver is delivered which trades wireless connectivity with flash memory space. Depending which Bluetooth stack layers are used, the additional space requirement is located between 4 and 20 KB. Because the RFCOMM functionality is embedded in the standard I/O stream interface, the usage of RFCOMM is as simple as interfacing the serial port.

Different strategies are presented to reduce the power consumption. Depending on the selected sleep modes, there is a power save potential between 50 and 98 % on the side of the microprocessor. In which time ratio the program will spend in the idle thread, and how much the external SRAM is accessed, are essential questions to evaluate more accurate data. The different sleep modes trades flexibility and reactivity for power.

The Bluetooth module provides low-power modes for connections which trade reactivity for power. Especially the sniff mode are examined. But the Bluetooth module on the BTnode has strong restrictions and limits the connections in sniff mode to one slave role connection and three master role connections. Due to these restrictions only a subset of topologies are targeted to put all connections to the sniff mode. If all three connections are in sniff mode, there will be a power consumption reduction about 43 % on the side of Bluetooth. But sniff connections in scatternets have a negative impact to the stability of the Bluetooth module. The Bluetooth module crashes in a while and renders the Bluetooth stack unusable. So far, sniff connections should be avoided in scatternets.

## 8.2 Outlook

The Bluetooth module on the BTnode is the greatest power consumer. If low-power becomes more importance on the BTnodes, it will be unavoidable to look at another Bluetooth module. Besides the gains in the stability, Bluetooth modules of the newest generation support a new feature from the Bluetooth specification 2.0. This feature is called *Enhanced Data Rate* (EDR) which triples the data rate. The higher data rate shortens the transmission time. Hence the overall power consumption will be reduced.

As long as the old Bluetooth module is used, the Bluetooth stack should be extended to be prepared to a crash on the part of the Bluetooth module. So far the crash causes a blocked Bluetooth stack. All threads which accesses the Bluetooth stack are affected in the same fashion. A timeout implementation for each HCI command can be used to notice about the crash, and appropriate countermeasures can be taken.

In this thesis, the low-power custom radio CC1000 is disregarded completely. But the CC1000 has a great potential to reduce the power consumption. In order to put the whole BTnode in a very low-power mode, the Bluetooth module is powered down, the microcontroller is put into a deep sleep mode and the CC1000 is used as a wake-up radio. This approach reduces the power consumption considerably if the BTnode is not used of an indeterminate duration.

# Bibliography

[1] Bluetooth Protocol Architecture Version 1.0 Document No. 1.C.120/1.0, Bluetooth Special Interest Group (SIG).

[2] Bluetooth Core Specification v1.2, Bluetooth Special Interest Group (SIG), http://bluetooth.org/spec.

[3] Atmel ATmega128L Processor Manual 2467N-AVR-03/06, Atmel Cooperation, http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf.

[4] **B. Kumar, P. Kline, T. Thompson**, "Bluetooth Application Programming with the Java APIs“, 2003, Morgan Kaufmann, ISBN 1-55860-934-2.

[5] **L. Negri, L. Thiele**, "Power Management for Bluetooth Sensor Networks“, Proc. 3rd European Workshop on Wireless Sensor Networks (EWSN 2006), Springer Verlag, Berlin, No. 3868, pages 196-211, February 2006.

[6] **L. Negri, L. Thiele**, "The Power Consumption of Bluetooth Scatternets“, Proc. IEEE Consumer Communications and Networking Conference (CCNC 2006), pages 519-523, January 2006.

[7] **J. Beutel, M. Dyer, L. Meier, L. Thiele**, "Scalable Topology Control for Deployment-Support Networks“, Fourth International Symposium on Information Processing in Sensor Networks (IPSN 2005), ACM, pages 359-363, April 2005.

[8] **D. Hobi, L. Winterhalter**, "Large-scale Bluetooth Sensor-Network Demonstrator“, Master Thesis, ETH Zurich, Summer term 2005, http://www.tik.ee.ethz.ch/∼beutel/projects/sada/2005ss_hobiwinterhalter.pdf.

[9] **Kevin Martin**, "Adaptive XTC on BTnodes“, Master Thesis, ETH Zurich, Winter term 2005, http://www.tik.ee.ethz.ch/∼beutel/projects/sada/2004ws_martin_xtc_report.pdf.

[10] **K. Römer, F. Mattern**, "The Design Space of Wireless Sensor Networks“, IEEE Wireless Communications, Vol. 11, No. 6, S. 54-61, December 2004.

[11] **J. Beutel, M. Dyer, L. Meier, M. Ringwald, L. Thiele**, "Next-Generation Deployment Support for Sensor Networks“, TIK Report No. 207, ETH Zurich, November 2004.

[12] **I. Chakraborty, A. Kashyap, A. Rastogi, H. Saran, R. Shorey, A. Kumar**, "Policies for increasing throughput and decreasing power consumption in bluetooth mac“, Proc. IEEE intl. conf. on Pers. Wirel. Comm., pages 90-94, December 2000.

[13] **H. Zhu, G. Cao, G. Kesidis, C. Das**, ”An adaptive powerconserving service discipline for bluetooth“, IEEE intl. conf. on Comm., volume 1, pages 303-307, 2002.

[14] **R. L. Ashok, R. Duggirala, D. P. Agrawal**, ”Energy efficient bridge management policies for inter-piconet communication in bluetooth scatternets“, Proc. Vehicular Tech. Conf., 2003.

[15] **M. Leopold, M. B. Dydensborg, P. Bonnet**, ”Bluetooth and sensor networks: a reality check“, SenSys ’03: Proceedings of the 1st international conference on Embedded networked sensor systems, pages 103-113, New York, NY, USA, 2003.

[16] **K. Y. Lee, J. W. Choi**, ”Remote-controlled home automation system via bluetooth home network“, SICE 2003 Annual Conference, volume 3, pages 2824-2829, August 2003.

[17] **Mustafa Yücel**, ”Inspektion von Sensornetzen per PDA“, Term Thesis, ETH Zurich, Summer term 2005, http://people.ee.ethz.ch/∼yuecelm/reports/sa_pdasensornetz.pdf.

# Appendix A

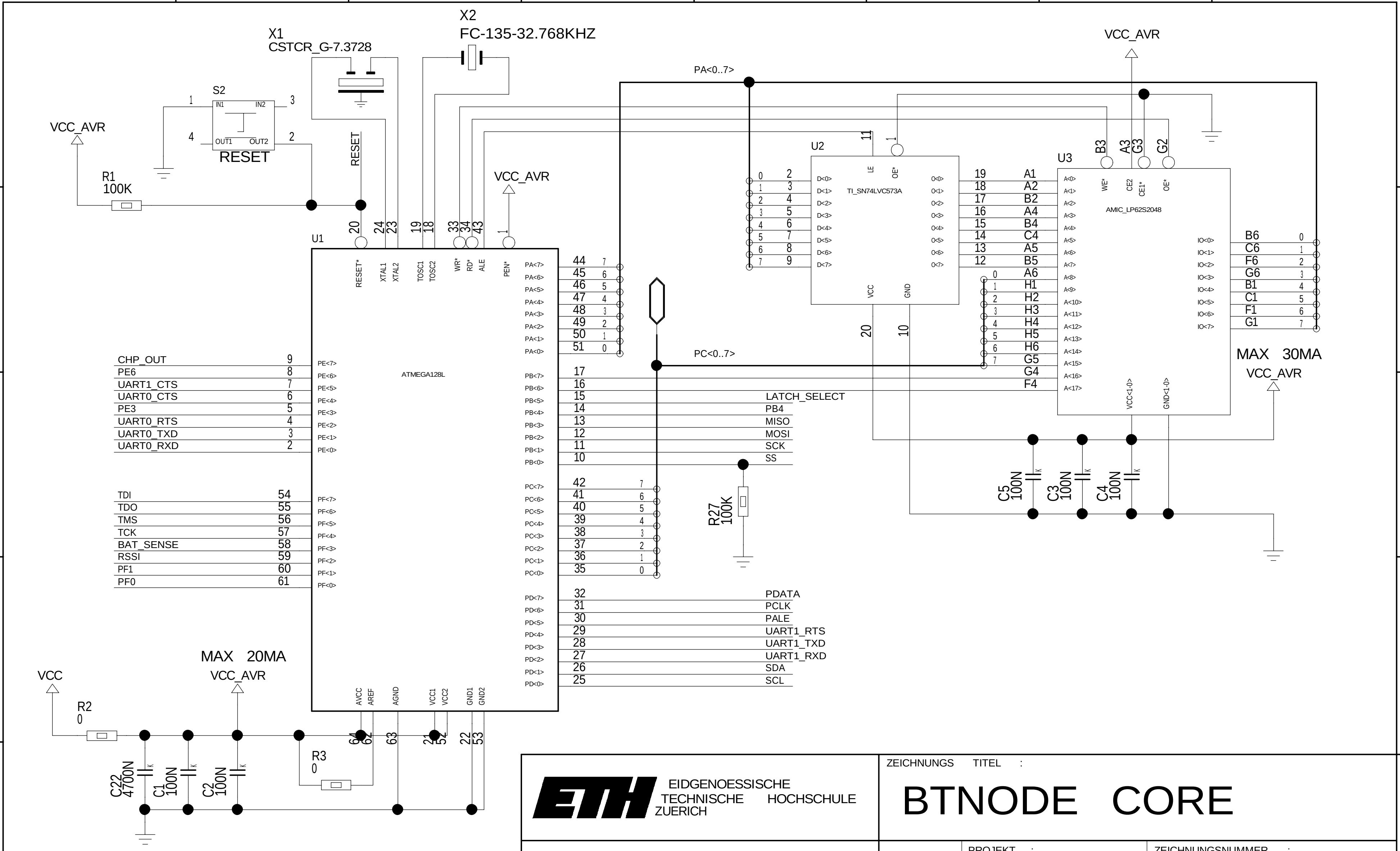
**BTnode rev3.22 schematic**

**BTnut application: sleep-and-work.c**

**Connection algorithm of the LCM**

**Java: StabilityLog.java**

**Task description**

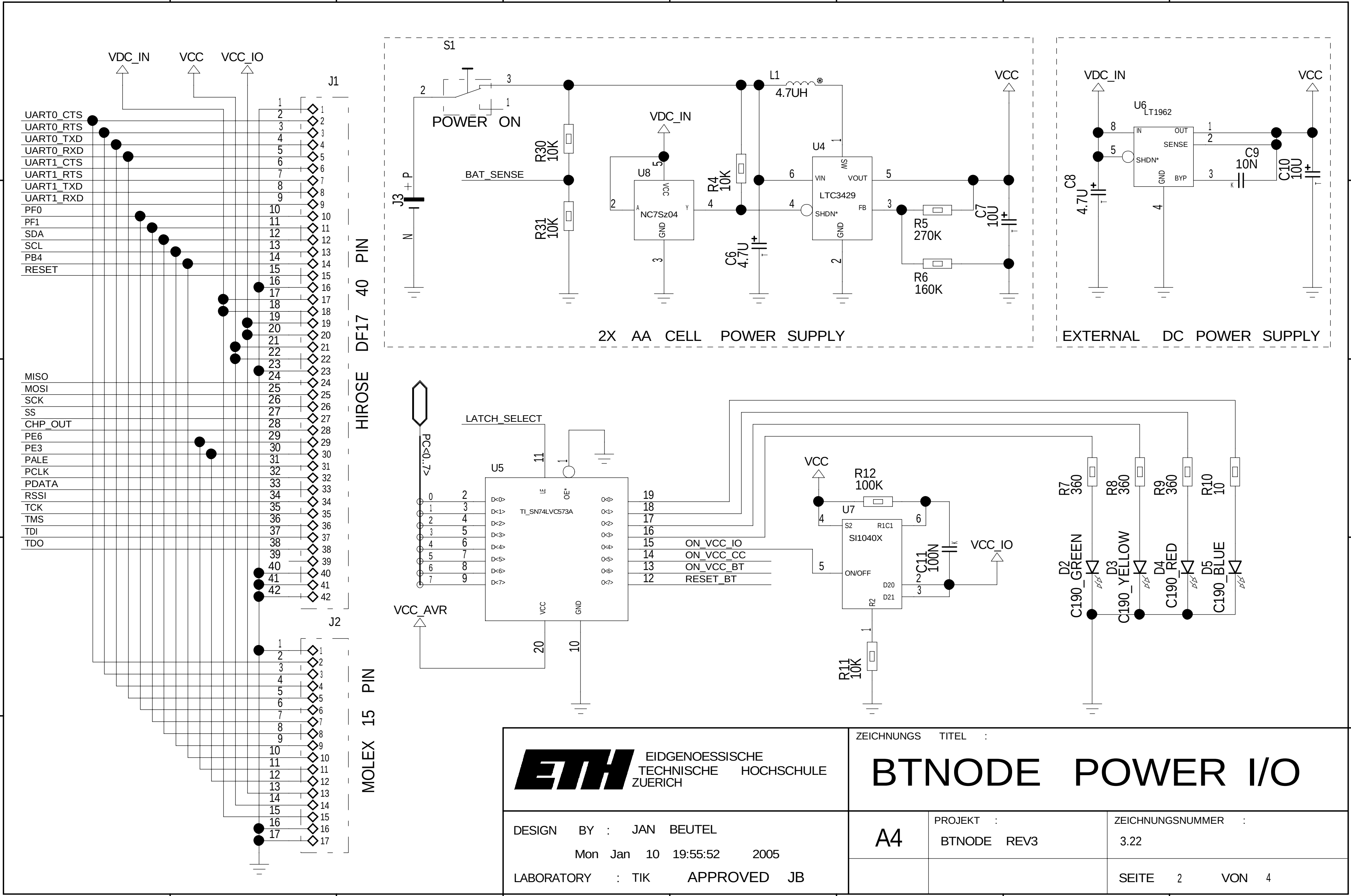


ETH EIDGENOESSISCHE TECHNISCHE HOCHSCHULE ZUERICH

ZEICHNUNGS TITEL :

BTNODE CORE

| | | | |
|---|---|---|---|
| DESIGN BY : JAN BEUTEL | A4 | PROJEKT : BTNODE REV3 | ZEICHNUNGSNUMMER : 3.22 |
| Mon Jan 10 19:55:51 2005 | | | |
| LABORATORY : TIK APPROVED JB | | | SEITE 1 VON 4 |

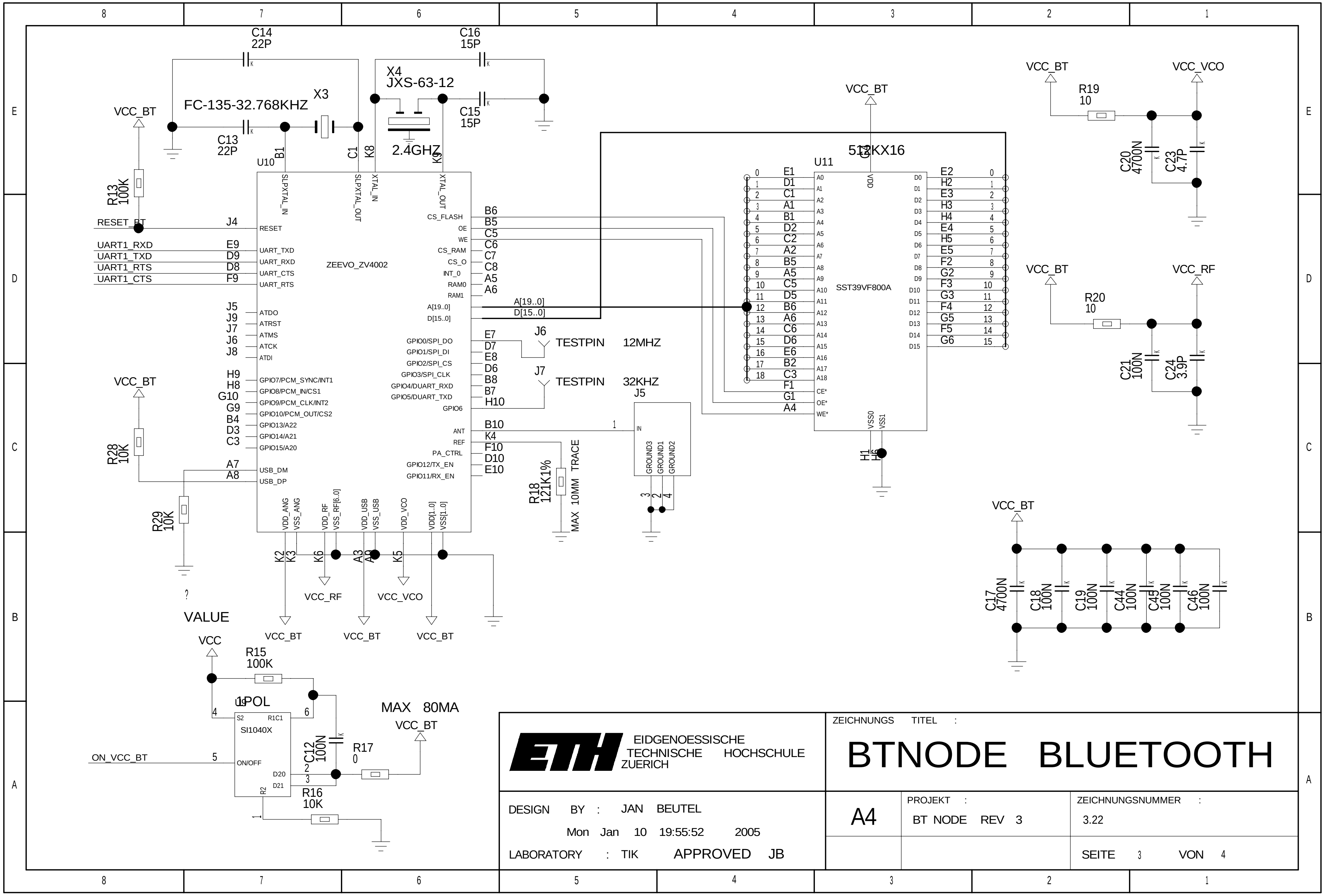


EIDGENOESSISCHE TECHNISCHE HOCHSCHULE ZUERICH

ZEICHNUNGS TITEL :

# BTNODE POWER I/O

| | | |
|---|---|---|
| DESIGN BY : JAN BEUTEL | A4 | PROJEKT : BTNODE REV3 |
| Mon Jan 10 19:55:52 2005 | | ZEICHNUNGSNUMMER : 3.22 |
| LABORATORY : TIK APPROVED JB | | SEITE 2 VON 4 |



EIDGENOESSISCHE TECHNISCHE HOCHSCHULE ZUERICH

ZEICHNUNGS TITEL :

BTNODE BLUETOOTH

DESIGN BY : JAN BEUTEL

Mon Jan 10 19:55:52 2005

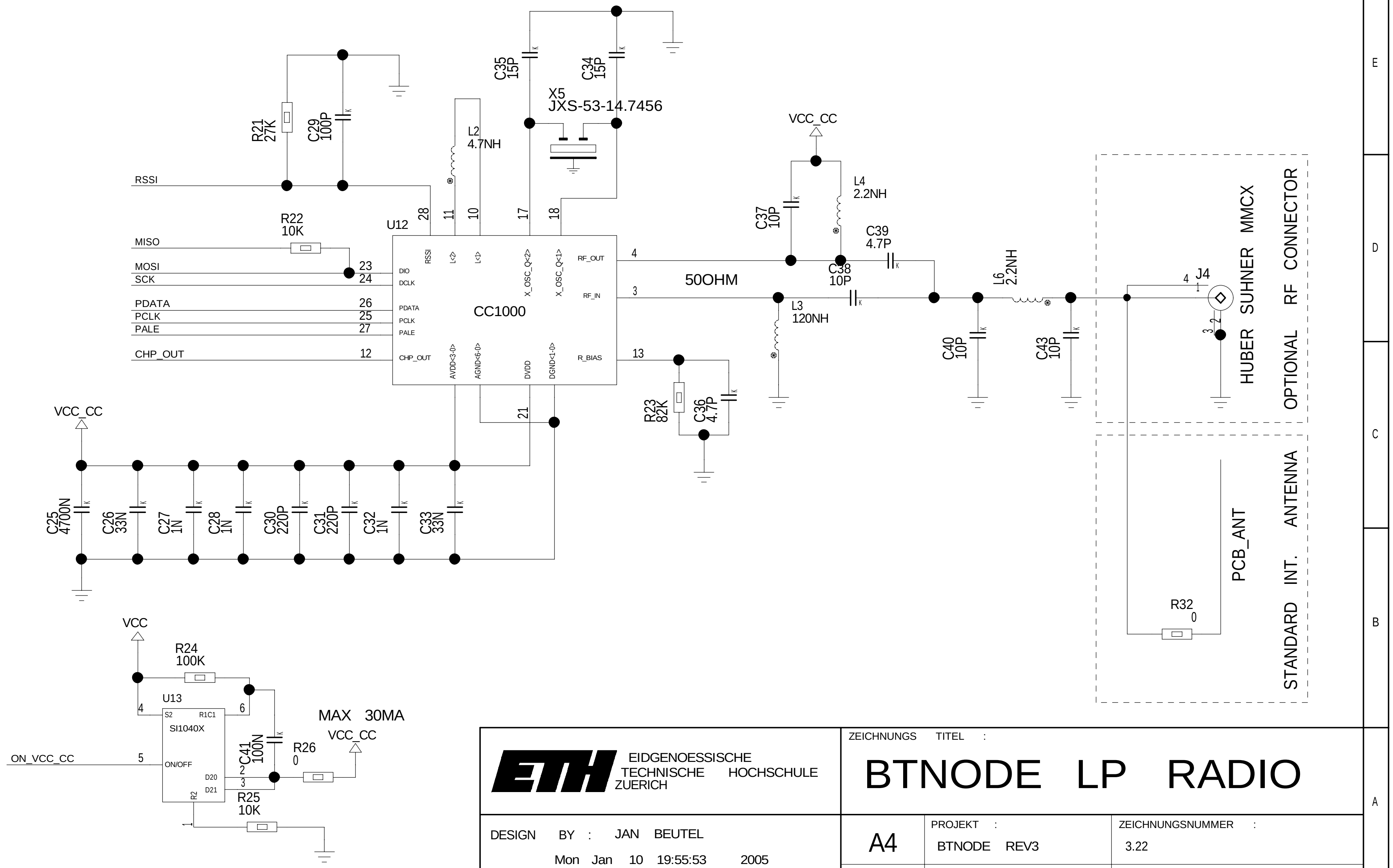
LABORATORY : TIK APPROVED JB

A4

PROJEKT : BT NODE REV 3

ZEICHNUNGSNUMMER : 3.22

SEITE 3 VON 4



| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|

E D C B A

EIDGENOESSISCHE TECHNISCHE HOCHSCHULE ZUERICH

ZEICHNUNGS TITEL :

BTNODE LP RADIO

DESIGN BY : JAN BEUTEL

Mon Jan 10 19:55:53 2005

LABORATORY : TIK

APPROVED JB

A4

PROJEKT : BTNODE REV3

ZEICHNUNGSNUMMER : 3.22

SEITE 4 VON 4

```
//
//   sleep-and-work application
//   (to test the sleep modes)
//

#include <hardware/btn-hardware.h>
#include <led/btn-led.h>
#include <sys/thread.h>
#include <sys/timer.h>

// incrementing variables for ~ 1 sec
void work(void)
{
    u_long i;
    for (i=0;i<30000000; ) { i++; }
}

int main(void)
{
    btn_hardware_init();

    // turn off LEDs
    btn_led_init(0);

    // turn off both communication modules
    btn_hardware_bt_power(0);
    btn_hardware_cc1000_power(0);

    // set the sleep mode for the idle thread
    // SLEEP_MODE_{NONE|IDLE|ADC|EXT_STANDBY|PWR_SAVE|STANDBY|PWR_DOWN}
    NutThreadSetSleepMode(SLEEP_MODE_IDLE);

    while (1)
    {
        // runs the idle thread for 1 sec
        NutSleep(1000);
        // do something for 1 sec
        work();
    }

    return -1;
}
```

```
//
//   Connection algorithm in pseudo code
//      of the Local Connection Manager (LCM)
//   (establishes connections to other JAWS nodes)
//

for (;;)
{
  start_async_inquiry();

  cleanup_page_blacklist();
  cleanup_page_list();
  bubblesort_page_list();

  result = wait_response_start_async_inquiry();

  foreach result_i
  {
    if (check_mac_prefix(result_i)) else continue;
    if (check_cod(result_i)) else continue;
    if (check_page_blacklist(result_i)) else continue;
    if (check_contable(result_i)) else continue;

    update_pagelist(result_i);
  }

  bubblesort_page_list();

  sleeptime += sleeptime_step;

  foreach page_list_entry_i
  {
    if (check_connection_limits()) else break;

    sleep(random(sleeptime_min));

    if (check_contable(page_list_entry_i)) else continue;

    create_async_connection(page_list_entry_i);
    success = wait_result_create_async_connection();
    if (success)
      delete_from_pagelist(page_list_entry_i)
  }

  if (no_connections())
    sleeptime = sleeptime_min;

  sleep(sleeptime);
}
```

```
//
// StabilityLog.java
//
// test the *stability* of JAWS nodes
//    using terminal command 'coninfo'
//

import java.io.InputStream;
import java.io.OutputStream;

import gnu.io.CommPortIdentifier;
import gnu.io.SerialPort;

public class StabilityLog
{

  public static void main(String[] args)
  {
    CommPortIdentifier portId = null;
    SerialPort port = null;
    InputStream is;
    OutputStream os;
    String[] lines = null;
    byte[] bytes = new byte[50000];
    long start;
    int i, j, n, devices, sniff_conns, active_conns;

    try
    {
      portId = CommPortIdentifier.getPortIdentifier("COM3");

      port = (SerialPort) portId.open("JAWS", 5000);
      port.setSerialPortParams(57600, SerialPort.DATABITS_8,
                               SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
      port.setFlowControlMode(SerialPort.FLOWCONTROL_NONE);

      // reserves enough receive buffer
      port.setInputBufferSize(50000);

      is = port.getInputStream();
      os = port.getOutputStream();

      start = System.currentTimeMillis();

      System.out.println("STABILITY = [");

      for (i=0;i<100;i++)
      {
        devices = 0;
        sniff_conns = 0;
        active_conns = 0;

        // blink broadcast
        os.write("rpcs 0 blink\n".getBytes()); // blink broadcast
        Thread.sleep(1000);

        // broadcasting coninfo
        os.write("coninfo\n".getBytes());

        Thread.sleep(30000);

        if (is.available() > 0)
        {
          n = is.read(bytes);
          lines = new String(bytes).substring(0,n).split("\n");

          for (j=0;j<lines.length; j++)
          {
            if (lines[j].startsWith(":T "))
            {
              devices++;
            }
            else if (lines[j].startsWith(":TE "))
            {
              if (lines[j].charAt(10) > '4')
              {
                sniff_conns++;
              }
              else
              {
                active_conns++;
              }
            }
          }
        }

        System.out.println((System.currentTimeMillis() - start) +" "+
                           devices +" "+ sniff_conns +" "+ active_conns);
      }

      System.out.println("]");

      port.close();
    }
    catch (Exception e)
    {
      if (port != null)
      {
        port.close();
      }

      System.err.println(e.getMessage());
      System.exit(-1);
    }
  }

}
```

**Institut für Technische Informatik**
**und Kommunikationsnetze**
**Computer Engineering and Networks Laboratory**

Wintersemester 2005/06

MASTERTHESIS MA-2006.06
of

**Mustafa Yücel**

| | | |
|---|---|---|
| Tutor: | Matthias Dyer | (dyer@tik.ee.ethz.ch) |
| Co-Tutor: | Jan Beutel | (beutel@tik.ee.ethz.ch) |
| Professor: | L. Thiele | |

Ausgabe: 12. December, 2005
Abgabe: 12. June, 2006

# Role and Link-State Selection for Bluetooth Scatternets

## Introduction

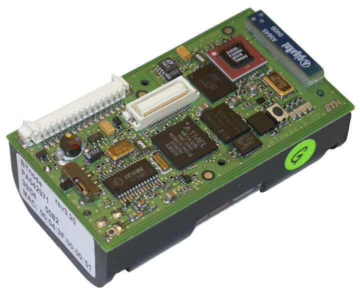


Abbildung 1: BTnode rev.3

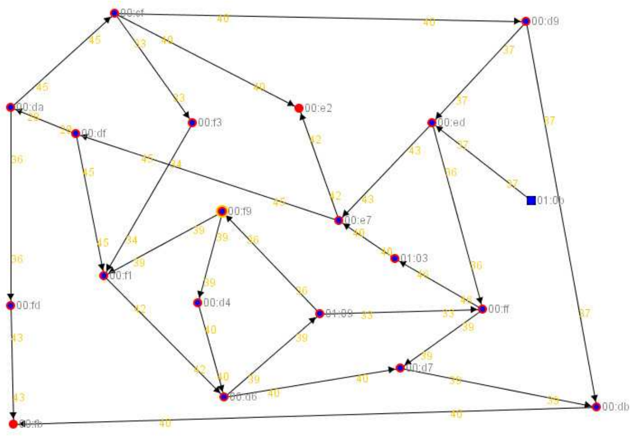


Abbildung 2: Example of a XTC Scatternet

A sensor network is a collection of small, low-resource devices that are distributed in the physical environment. Due to cost and flexibility issues, it is often assumed to be a wireless sensor network (WSN) consisting of a large number of sensor nodes. Each of these nodes collects sensor data, and the network collaboratively provides high-level sensing results.

The *BTnode rev.3* [1] (see Fig.1) is a plattform for the development of sensor–network applications and protocols. It has two radio interfaces: a Bluetooth radio provides relatively high bandwith, while the second radio is for low-rate and low-power operation. Recently, a new system software [2] for the BTnode rev.3 has been released, that is based on the Ethernut embedded OS [3].

Deployment-support networks (DSNs) have been proposed [4] as a non-permanent, wireless cable replacement for the development, testing and debugging of sensor network applications. This approach allows to deploy and test large numbers of devices in a realistic physical scenario. The DSN is transparent, highly scalable, and can be quickly deployed. It does not disturb the target WSN any more than the traditional, cable-based approach. For the engineer, everything actually looks as if the usual cables were in place; he can thus use the same tools. The

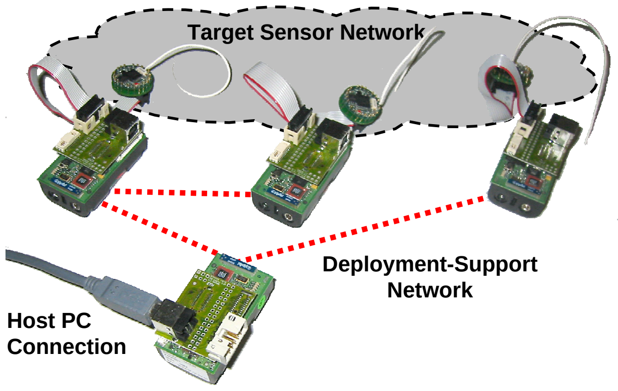


Abbildung 3: Deployment Support Network

DSN nodes are attached to WSN target devices via a programming and debugging cable and form an autonomous network (see Fig. 3). The WSN nodes can then be accessed through serial-port tunnels operated over the DSN. With this tool, the limit for largescale prototyping is pushed from simulation and virtualization to coordinated real-world deployment.

## Problem task (german)

In früheren Arbeiten [5, 6, 9] wurde ein DSN Demonstrator entwickelt, mit den folgenden Eigenschaften:

1. **Topology Control:** Nach dem Einschalten beginnen die BTnodes sich miteinander zu verbinden. Ein Algorithmus entscheidet, welche Verbindungen ausgewählt werden und wie die Netzwerktopology am Ende aussieht. Zur Zeit kann zwischen einem baum-basierten und einem mesh-basierten (XTC) Algorithmus gewählt werden.

2. **Target Monitoring:** An jedem DSN-BTnode kann ein Target Node angeschlossen werden über die serielle Schnittstelle.

3. **Event Logging:** Events vom Target oder vom DSN-Knoten selber können lokal mit Zeitstempel geloggt und ans GUI gesendet werden.

4. **Remote Programming:** Vom GUI aus kann ein neues Software-Image auf die BTnodes geladen werden. Sowohl für die DSN-, als auch für die Target-Knoten.

5. **GUI:** Der GUI-Server ist über ein serielles Kabel mit einem BTnode verbunden. Der GUI-Client verbindet sich mit dem Server und ist das Benutzerinterface zum DSN.

Das Ziel dieser Arbeit ist, den Prototypen in folgenden 2 Bereichen zu erweitern:

1. **L2CAP Connection:** Zur Zeit muss der BTnode über ein serielles Kabel mit dem PC/PDA verbunden werden um die DSN Services zu nutzen. In der Arbeit [8] wurde erfolgreich gezeigt, dass der PC/PDA auch über Bluetooth (L2CAP) mit den BTnodes kommunizieren kann. Diese Funktionalität soll nun für das DSN verfügbar gemacht werden.

2. **Low Power Modes:** Da bis jetzt noch keine energiesparende Funktionen von Bluetooth genutzt werden, ist die Lebenszeit eines batteriebetriebenen Nodes sehr beschränkt. In dieser Arbeit soll in einem ersten Schritt die energiesparenden Modes (v.a. der Sniff-Mode) ausprobiert werden um sie dann in einem zweiten Schritt in den bestehenden Demonstrator zu integrieren. Die gewonnene Energieeffizienz soll gemessen und verglichen werden.

Für das Testen und Messen der erzielten Ergebnisse soll direkt das DSN verwendet werden. D.h. eine Case-Study für das DSN soll ein weiteres Ergebniss dieser Arbeit sein.

### Teilaufgaben

1. Erstellen Sie einen Projektplan und legen Sie Meilensteine sowohl zeitlich wie auch thematisch fest. Erarbeiten Sie in Absprache mit dem Betreuer ein Pflichtenheft.
2. Machen Sie sich mit den relevanten Arbeiten im Bereich Sensornetze vertraut. Führen Sie eine Literaturrecherche durch. Suchen Sie auch nach relevanten neueren Publikationen.
3. Arbeiten Sie sich in die Softwareentwicklungsumgebung der BTnodes ein. Machen Sie sich mit den erforderlichen Tools vertraut und benutzen Sie die entsprechenden Hilfsmittel (online Dokumentation, Mailinglisten, Application Notes).
4. Machen Sie sich mit der JAWS Applikation vertraut. Schauen Sie sich insbesondere die Implementierung des L2CAP-Layers an. Arbeiten Sie sich in die Grundlagen von Bluetooth ein. Wesentlich für diese Arbeit ist vor allem das *HCI Interface* und alles was *L2CAP* und *sniff-mode* betrifft.
5. Erweitern Sie die JAWS DSN Applikation, so dass von einem GUI PC aus eine L2CAP Verbindung aufgemacht werden kann. Die Bluetooth-Verbindung soll die herkömmliche serielle Verbindung zum Terminal ersetzen. Erweitern Sie auch die GUI Applikation (Server), so dass die neue Bluetooth Verbindungsart genutzt werden kann.
6. Erarbeiten Sie ein Konzept, welches beschreibt, wie die low-power modes auf dem DSN genutzt werden können. Lesen sie dazu auch [7]
7. Implementieren und testen sie die low-power modes im DSN.
8. Definieren und messen sie relevante Charakterisiken Ihrer Implementierung.
9. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

## Durchführung der Masterarbeit

### Allgemeines

- Der Verlauf des Projektes Masterarbeit soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.
- Sie verfügen über PC's mit Linux/Windows für Softwareentwicklung und Test. Für die Einhaltung der geltenden Sicherheitsrichtlinien der ETH Zürich sind Sie selbst verantwortlich. Falls damit Probleme auftauchen wenden Sie sich an Ihren Betreuer.
- Stellen Sie Ihr Projekt zu Beginn der Masterarbeit in einem Kurzvortrag vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums Ende Semester.
- Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern. Verfassen Sie dazu auch einen kurzen wöchentlichen Statusbericht (EMail).

### Abgabe

- Geben Sie zwei unterschriebene Exemplare des Berichtes spätestens am *12. June 2006* dem betreuenden Assistenten oder seinen Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.

## Literatur

[1] Btnodes, a distributed environment for prototyping ad hoc networks. http://www.btnode.ethz.ch.

[2] Btnut system software reference. http://www.btnode.ethz.ch/support/btnut_api/index.html.

[3] Ethernut. http://www.ethernut.de/.

[4] J. Beutel, M. Dyer, L. Meier, and L. Thiele. Scalable topology control for deployment-support networks. In *Proc. 4th Int'l Conf. Information Processing in Sensor Networks (IPSN '05)*, pages 359–363. IEEE, Piscataway, NJ, April 2005.

[5] Daniel Hobi and Lukas Winterhalter. Large-scale bluetooth sensor-network demonstrator. Master's thesis, ETH Zurich, Switzerland, 2005. MA-2005-13.

[6] Kevin Martin. Adaptive XTC on BTnodes. Master's thesis, ETH Zurich, Switzerland, 2005. MA-2005-05.

[7] L. Negri, J. Beutel, and M. Dyer. The power consumption of bluetooth scatternets. In *IEEE Consumer Communications and Networking Conference*, page to appear. IEEE, Piscataway, NJ, 2006.

[8] Mustafa Yuecel. Inspektion von Sensornetzen per PDA. Master's thesis, ETH Zurich, Switzerland, 2005.

[9] Sven Zimmermann. Online sensor-network monitoring. Master's thesis, ETH Zurich, Switzerland ETH Zurich, Switzerland, 2005. SA-2005-26.