ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Daniel Hobi, Lukas Winterhalter

Large-scale Bluetooth Sensor-Network Demonstrator

Master's Thesis MA-2005-13
Summer Term 2005

Tutor: Jan Beutel, Matthias Dyer

Supervisor:
Prof. Dr. L.Thiele

1st October 2005

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

Abstract

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

Both, advances in the fabrication of hardware and the recent progress in wireless network technologies spawned a great variety of impressive, low-cost prototyping platforms for wireless sensor networks, utilized by research groups and industry alike.

Existing development tools however do not scale to the same dimensions. Currently, wired testbeds are used to develop and test small or medium-sized networks. The only option for larger networks, as of today, was simulation in a virtual environment. The problem with both variants is that the sensor network application cannot be tested in real environments.

The *Deployment-Support Network* represents a novel tool which addresses this issue. It is a non-permanent wireless cable replacement that provides services to develop, test, validate and deploy wireless sensor network applications.

In this thesis, the concept of the deployment-support network and a first, exemplary implementation were pursued further. Services to program, control and monitor sensor nodes were specified and implemented.

An extension board for the BTnode platform was designed, manufactured and tested. This board is used to establish a wired connection between BTnodes that are part of the deployment-support network and sensor nodes for which an application is to be developed.

A permanent setup was installed for demonstration purposes. This demonstrator may be used for experiments and measurements. Several stability and performance tests were conducted, and it was shown that the development system presented in this thesis is well suited for long-term applications.

Contents

1

Introduction

1.1 Wireless Sensor Networks

Wireless Sensor Networks (WSN) are typically considered as a large number of small, independent sensor nodes with limited resources and low production costs. Because of their small physical dimensions, they can be used to monitor all kinds of real-world phenomena without disturbing the observed environment. Often, the intention is there to use them in hardly accessible, wide-spread areas. The lack of infrastructure in these scenarios makes it desirable that the sensor nodes are autonomous units. Popular visions suggest that they organize themselves in wireless ad hoc networks to collaboratively forward and even process measurement data in order to provide high-level sensing results. It can be said that the sensor nodes compose in their collectivity a complex distributed system.

As an example, the Argo project [1] is depicted in Figure 1-1. Argo is a global array of 3000 free-drifting sensor floats that will measure the temperature, salinity and velocity of the upper ocean layers. The floats use satellite links to transfer the data. Main goal of the project is to improve the understanding of the ocean's role in climate changes.

Generally, developing and testing of such systems is rather difficult for the following reasons:

- Sensor node complexity. Autonomous behavior, wireless components, sensor integration, robustness, power, actuators and data processing capabilities have to be reconciliated.
- Wireless sensor nodes are often embedded systems with limited debugging possibilities.
- Programming and debugging of many nodes simultaneously requires additional tools and infrastructure. For large numbers of sensor nodes this can lead to a scaling problem of the test equipment and become infeasible.

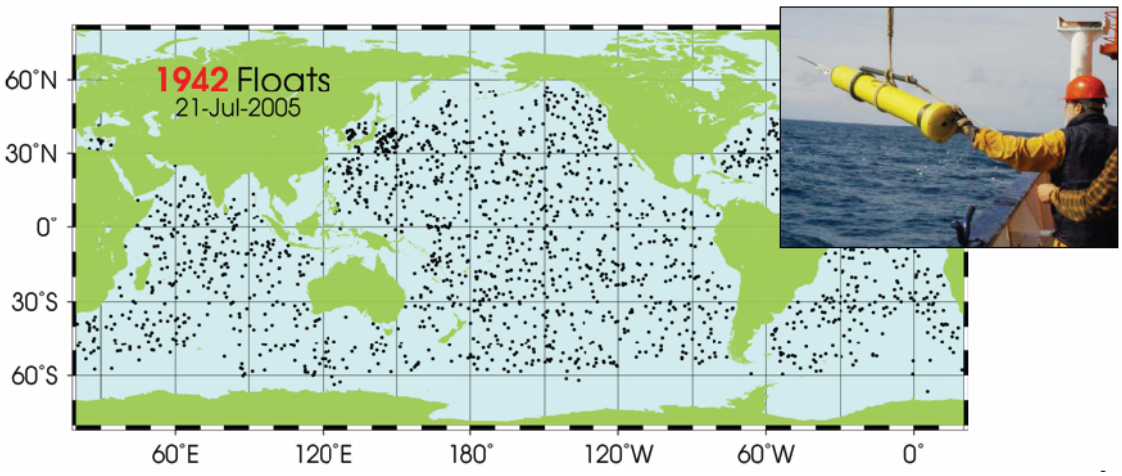


Figure 1-1
The Argo project [1] targets global ocean surveillance. A great number of independent floats have been positioned all over the planet. On the left: an Argo float being deployed from a research ship.

- Testing the interaction of many nodes in a realistic environment is difficult. The reason is that wireless sensor networks are often deployed in inaccessible terrains. The Argo project is a good example.

  Simulations of whole wireless sensor networks are possible. However, making the simulation realistic with respect to sensor node hardware and target environment is not easy. Often, simplifications and assumptions about certain conditions lead to a divergency between simulation and reality.

Embedding network programming and debugging features into the sensor network application itself is widely accepted to be a bad idea. It would increase code size and complexity of the firmware. Additionally, it would affect performance and networking behavior of the sensor nodes during testing.

1.2 Deployment-Support Network

The increasing efforts both from research and industry have spawned many different platforms and proposals for WSNs. Nevertheless, the large number of nodes and the need to test them in a realistic physical environment makes developing of large-scale WSN applications still very hard.

Classic approaches to develop and deploy WSNs use serial cables for reprogramming, testing, monitoring and controlling of the sensor nodes. Although successful in lab setups (confer next section) this method does not scale very well. For larger numbers of nodes, for deployment or even testing in the field, it is infeasible to connect a cable to each node. As an alternative, often computer simulation environments are used. These have the advantage that they are highly scalable. However, simulations can only approximate and have to make assumptions that do not necessarily match the conditions in the real environment.

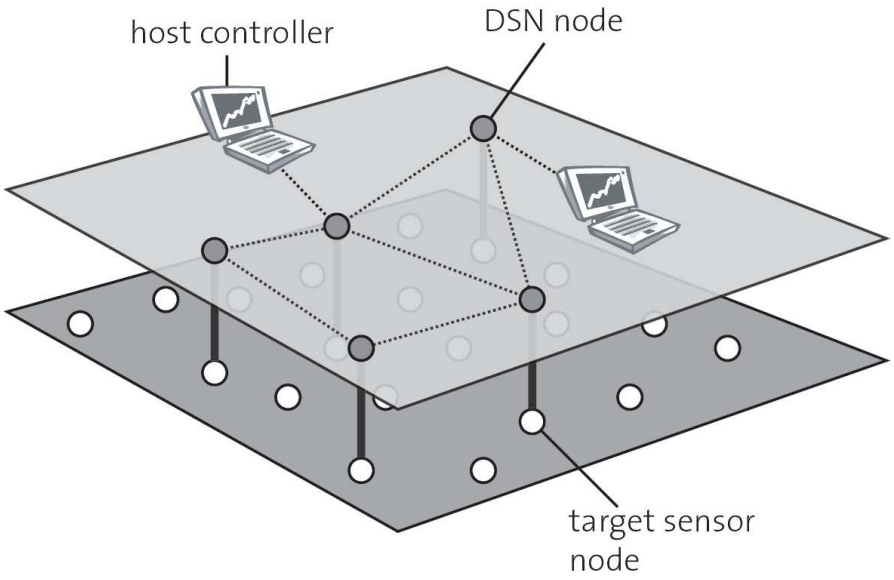


Figure 1-2
Deployment-support network. Some or all nodes of the sensor network are wired to a DSN node. The DSN provides wireless connectivity from one or more host controllers (personal computer) to the target nodes, allowing remote programming, debugging, monitoring and control.

In [6] a *Deployment-Support Network (DSN)* has been proposed as a tool for the deployment of large WSNs. The DSN is a non-permanent, wireless cable replacement that assists in the process of developing WSN applications. The intended features include remote programming, debugging, monitoring and target control. It allows to perform calibration and validation of a WSN application in realistic physical environments.

Figure 1-2 illustrates the concept. It consists of small wireless DSN nodes which are attached to the target sensor nodes. Both, the DSN and the target sensor nodes span up independent wireless networks. The DSN network can be considered as a wireless backbone network. A host, e.g. a personal computer can be used to access and control the DSN network by connecting to one of the DSN nodes.

It is expected that this new approach enhances scalability and flexibility with respect to node location, density, and mobility.

In [7] a first implementation called JAWS of such a DSN is described based on the BTnode platform. It uses Bluetooth to construct a tree-based multi-hop wireless network. As a first service, it is possible to establish virtual connections between arbitrary target nodes and a host PC over the DSN network. More details about JAWS are provided in Section 3.1.

1.3 Related Work

MoteLab [19] is a wireless sensor network testbed located at Harvard University. It consists of 30 MicaZ sensor “motes” from Crossbow. A special interface board (MIB600) is used to attach the motes to the departmental Ethernet. It provides

reprogramming and data logging capabilities. MoteLab provides a web interface where users can upload executables, create and schedule experiments to be run on the moteLab testbed. A persistent data base logs all generated data. The moteLab software is open source and is used by several other universities.

The *sMote* [5] testbed at the UC Berkeley follows the same approach. 78 Mica2DOT sensor motes are powered and connected by Ethernet. The additional Omega testbed consists of 28 Telos motes (rev B) that are connected by USB for power, programming and debugging.

TOSSIM [16] is a simulator for TinyOS [9] sensor network applications. It simulates the TinyOS network stack and provides different abstraction models for the underlying hardware such as RF channels and sensors. It allows to run entire networks of thousands of sensor nodes.

The *EmStar* platform [11, 12] tries to improve the simulation by adding the possibility to interface with real hardware. It provides a wrapper library called EmTOS for TinyOS applications which makes it possible to simulate a great number of different motes with the option to communicate over real hardware. The hardware consists of a wired ceiling array of motes that are connected via a serial multiplexer to the simulating server running TinyOS instances.

Deluge [8, 15] is a network programming protocol to spread program code over a multi-hop, wireless network. Deluge relies on periodic *advertisements* to keep nodes informed of their neighbors' states. Nodes can send *requests* to receive parts of newer program data. An implementation is available for TinyOS.

1.4 Short Task Description

The goal of this thesis is to develop a permanent installation of the JAWS deployment-support network in the TIK building. For this purpose, the system has to be further improved and extended.

- Get familiar with related work. Perform a literature research.
- Evaluate the existing JAWS software. Determine missing features and potential bugs.
- Propose and develop a physical target connection for different targets.
- Implement the necessary DSN services remote command execution, code distribution, target programming, self-programming, etc. according to the determined requirements.
- Present proposals for distributed power, fixation, case. Deploy a permanent JAWS installation.
- Perform tests and benchmarks, interpret the results.
- Document your work accurately in a presentation, a small demonstration and a final report.

For the entire assignment, please refer to Appendix D.

1.5 Thesis Overview

The thesis is organized as follows. In Chapter 2, an introduction to the BTnode and the wireless network technology Bluetooth is given. Chapter 3 begins with a discussion of the existing JAWS framework, followed by our proposals to extend it. In Chapter 4, implementation aspects of the added services are described. In Chapter 5, an new adapter hardware is presented that allows to connect different targets to the DSN nodes. In Chapter 6, the sensor-network demonstrator consisting of permanently installed DSN nodes is described. In Chapter 7, we present the obtained test results and benchmarks. Finally, Chapter 8 concludes the thesis.

2

Platform: BTnode

2.1 Overview

The BTnode [20] is a small, autonomous wireless communication and computing platform developed at ETH Zürich by the Computer Engineering and Networks Laboratory (TIK) and the Research Group for Distributed Systems.

Figure 2-1
BTnode rev3. The printed circuit board is mounted on a 2 AA cell battery case.

It is used in research on mobile and ad-hoc connected networks (MANETs) as well as distributed wireless sensor networks.

The BTnode is a dual radio device built around an Atmel Atmega128l microcontroller. An overview of the system is given in Figure 2-2. It features both a Zeevo ZV4002 Bluetooth system as well as a Chipcon CC1000 low-power radio. The BTnode can power and use both radios independently. The Bluetooth radio supports 7 slaves per piconet and up to 4 independent piconets for scatternet operation. The low-power Chipcon radio is the same as used on the Berkeley Mica2 Motes. Note

that it remains unused in this thesis. Four LEDs of different colors allow to provide status information and are useful for debugging purposes.

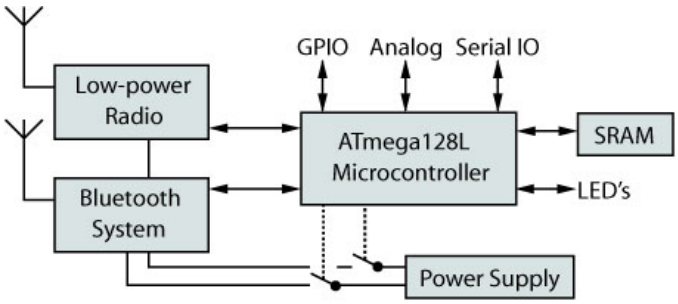


Figure 2-2
BTnode System Overview.

The Atmega128l is an 8-bit RISC microcontroller running at frequencies up to 8 MHz. Most instructions are executed in a single clock cycle. On the BTnode, the system clock is generated from an external 7.3728 MHz crystal oscillator. The Atmega128l provides 128 kB in-system reprogrammable flash, 4 kB EEPROM and 4 kB internal SRAM. Section 2.2 explains how 240 kB additional memory are made available on the BTnode.

The integrated peripherals include general purpose I/O pins, several 10-bit ADC inputs, JTAG, serial peripheral interface (SPI), two-wire serial interface (TWI[1]) and two universal asynchronous receiver transmitters (UART).

On the BTnode, the ATmega128l is connected over the second UART with the Bluetooth module. The first UART is available to applications. It is mainly used as debug and control interface by connecting with standard terminal emulator software.

2.2 Banked Memory

As mentioned before, the Atmega128l comes with 4 kB internal RAM. Additionally, it features an external memory interface which drives the ports A, C and G when enabled. Operating on 16-bit addresses, up to 64 kB external RAM can be attached. However, the first 4 kB are not accessable because they are mapped to the internal memory.

Figure 2-3 shows how 240 kB additional SRAM are made available on the BTnode. Two general purpose I/O pins (PB7 and PB6) are used to access data beyond the first 60 kB, otherwise not addressable via the standard interface. The external memory is divided into four equal sized memory banks. By default, only bank 0 is available providing 60 kb additional *data memory*. In order to access memory bank 1 to 3, from now on referred to as *storage memory*, special memcopy routines exist that control the two additional address lines:

```
u_char cpy_to_xbank(void *buffer, u_long addr, u_short len)
u_char cpy_from_xbank(void *buffer, u_long addr, u_short len)
```

[1]Also known as I^2C.

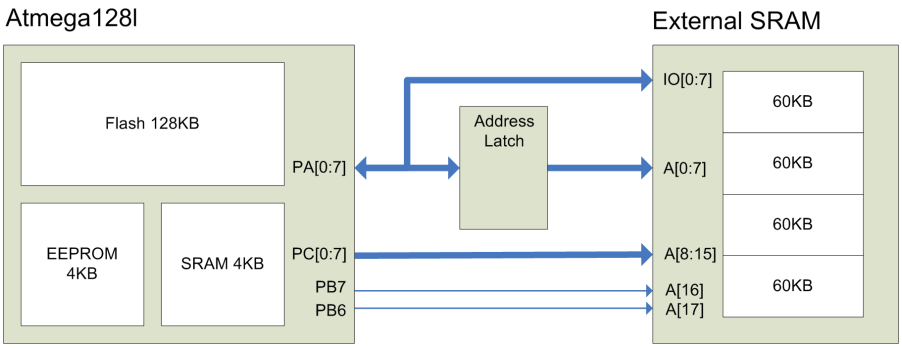


Figure 2-3
External SRAM connected to the Atmega128l.

With these functions data can be read from or written to storage memory. The addresses `addr` are 32-bit wide starting at zero and allow to address the 180 kB storage memory transparently.

One thing to note is that changing the memory bank in these functions requires that interrupts are disabled. This is due to the fact that the Nut/OS interrupt handlers may access data in external memory which is stored in bank 0. Therefore, it has to be made sure that no other bank is mapped in when an interrupt is executed.

More information about BTnode memories can be found in [10].

2.3 Bluetooth

Bluetooth is a short-range radio technology designed for wireless connectivity between all kinds of devices. The design goals were small size, low price and minimal power consumption.

Bluetooth operates in the license-free 2.4 GHz ISM[2] band. It uses a frequency-hopping scheme to improve robustness against interferences from other sources.

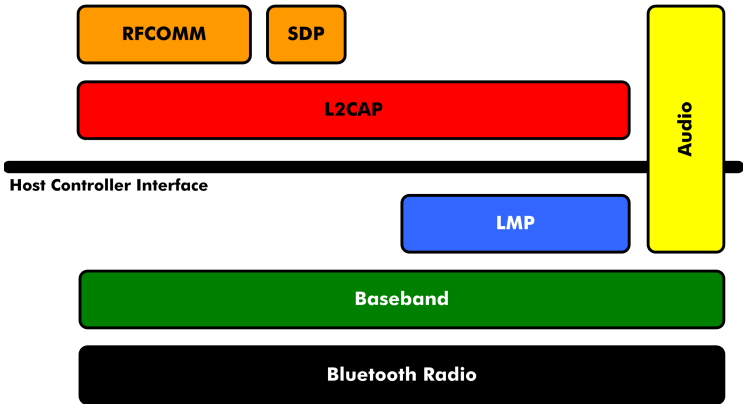


Figure 2-4
Bluetooth Protocol Stack.

In the following, we describe the basic Bluetooth layers depicted in Figure 2-4:

[2]Industrial Scientific and Medical

Baseband The *Baseband* layer controls the *physical radio*. It performs *inquiry* and *paging* to find and access nearby devices.

The Baseband layer provides different types of transport links between devices. The *Asynchronous Connection-Oriented (ACL*[3]*)* link type supports symmetric and asymmetric channels. The symmetric channel can transfer data up to 433.9 kbit/s in both directions. The asymmetric channels supports data rates up to 723.2 kbit/s in one direction and 57.6 kbit/s in the reverse direction.

ACL supports forward error correction and simple packet retransmission to provide basic reliability.

LMP The *Link Manager Protocol* is used for connection setup and control. Responsibilities of the LMP also include authentication and security services.

HCI The *Host Controller Interface* provides a uniform interface to the LMP and Baseband layers. It is the dividing line between the Bluetooth controller and the host controller.

L2CAP The *Logical Link Control and Adaption Layer Protocol* lies on top of the Host Controller Interface. L2CAP uses an abstraction called *channels* to allow multiple data flows over a shared ACL transport link between two Bluetooth devices. The channels are used to provide protocol multiplexing for higher protocol layers. Additional features include segmentation and reassembly of upper layer data and reliable packet transmission.

SDP The *Service Discovery Protocol* is used to search for services on a remote Bluetooth device.

RFCOMM The *Radio Frequency Communications Protocol* emulates a RS-232 serial connection between two Bluetooth devices over a L2CAP connection.

2.3.1 Scatternet Operation

Bluetooth devices have to synchronize to the same frequency hopping sequence in order to transmit data. This is done during connection establishment. When connected, the devices form a *piconet*. In every piconet there is one master to which up to seven slaves can be connected. The hopping sequence and its phase are determined by the Bluetooth address and the clock of the master device and are therefore unique for every piconet.

It is possible that a Bluetooth device takes part in more than one piconet by applying time multiplexing. Note that every device can only be master of one piconet, however, it may act as a slave in other piconets at the same time. This form of overlapping, when several piconet are interconnected, is called *scatternet* (see Figure 2-5).

2.4 Nut/OS

Nut/OS [2] is a simple general purpose operating system for embedded devices developed by an active open source community. Its features include:

[3] It is clear that the most obvious abbreviation would be ACO. However, according to the Bluetooth 1.2 specification this acronym had already an alternative meaning from a previous version.

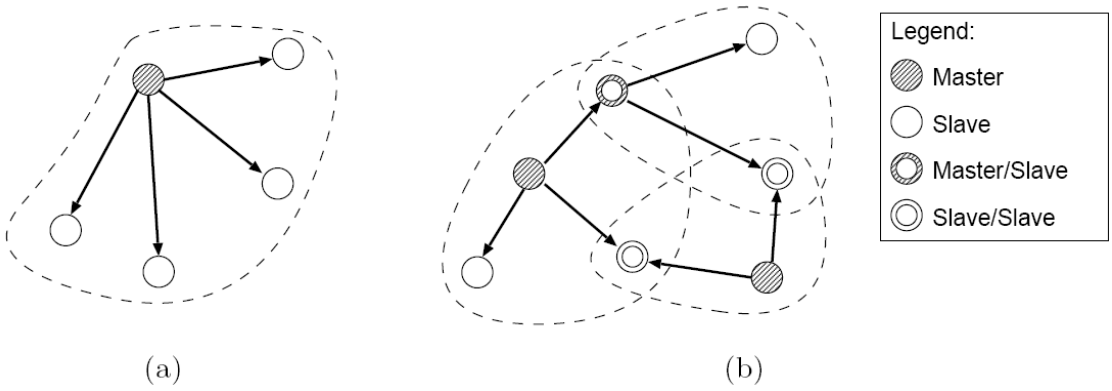


Figure 2-5
Bluetooth networks: Piconet (a) and multiple piconets connected to a scatternet (b).

- Non preemptive, cooperative multi-threading
- Thread priorities
- Events
- Timers
- Dynamic heap memory allocation
- Interrupt driven streaming I/O

The BTnut system software extends the Nut/OS by providing BTnode specific drivers and libraries. Most notably, a Bluetooth stack and several communication protocols have been implemented.

3

Concepts

In the following, we present our proposals to extend the existing JAWS software.

3.1 JAWS

3.1.1 Overview

JAWS [20] is a project that aims at implementing the deployment-support network concepts described in Section 1.2 for the BTnode platform. The JAWS application uses Bluetooth ad-hoc networking to autonomously interconnect the DSN nodes. By default, it establishes a tree-based topology as illustrated in Figure 3-1.

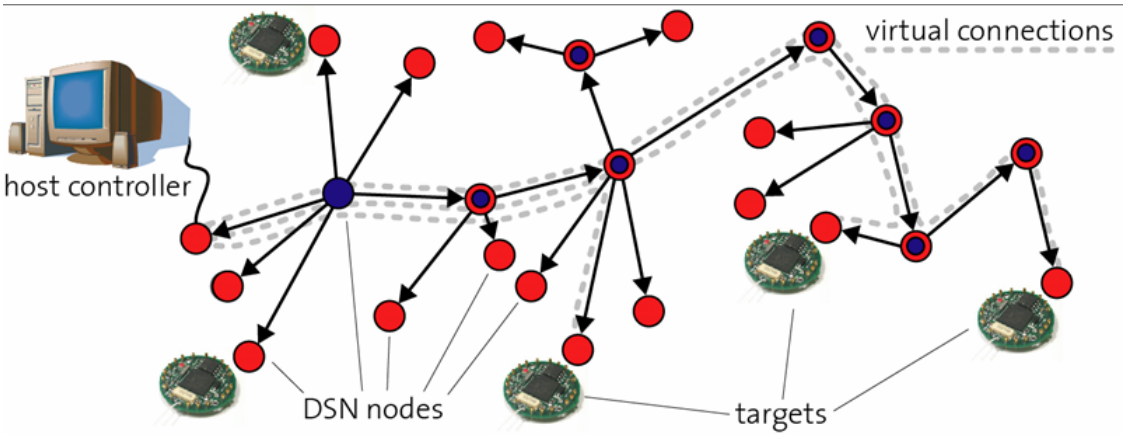


Figure 3-1
JAWS deployment-support network. The DSN nodes autonomously construct a multi-hop Bluetooth scatternet. The DSN nodes allow to control the attached targets from the host controller on the left. Here, Berkeley Motes are shown as exemplary targets.

The blue and red circles represent BTnode devices that take part in the DSN Bluetooth scatternet. The blue ones are master of their piconet while the red ones are

slaves. As explained in Section 2.3 a device can be master and slave in different piconets. Wireless sensor nodes, in the image referred to as target nodes, are attached over a wired connection to the DSN nodes.

One or more *host controllers* can hook into DSN nodes to control the target devices over the DSN network. The host controller is nothing else than a user device, usually a personal computer connected over USB with a DSN device (see Figure 5-2 on page 46). DSN nodes that are attached to a host controller are in the following denoted as *host nodes*.

A simple graphical user interface allowed to visualize the network topology and to control the DSN nodes. A first service that JAWS already provided, was the possibility to establish transparent virtual connections over several hops between one or more DSN nodes with attached targets and a host node.

A principle idea of JAWS is that the DSN nodes autonomously construct a network topology, but do not initiate further multi-hop network traffic. This is done exclusively by the host nodes that pass user requests into the network.

Because the Bluetooth standard does not specify the formation and control of network topologies or how data transport over multiple hops can occur, these functionalities have to be provided by the JAWS application. This is described in the following two sections.

3.1.2 Connection Manager

The *connection manager* constructs and maintains a multi-hop network of DSN nodes. The basic principle is simple: Every DSN node periodically searches for other nodes in its neighborhood and connects to the discovered devices based on certain conditions.

The default connection manager forms tree topologies. The tree structure is maintained by two parallel threads. The inquiry thread (Algorithm 1) periodically performs inquiries and randomly connects to one of the discovered DSN devices.

Algorithm 1 Connection manager: Inquiry thread

loop
  $neighbors := inquiry()$
  randomly select neighbor n from $neighbors$
  connect to n
end loop

The packet-handler thread (Algorithm 2) processes topology control packets that are exchanged between connected devices. Two types of packets are distinguished:

- $tree\_ID\_packet$: This packet type is used to prevent and detect cycles in the network topology.
- $negotiation\_packet$: This packet type is used to compare tree-IDs between neighboring nodes.

All nodes in the same network share the same tree-ID. Initially, every node is in its own network with the tree-ID set to its unique Bluetooth address.

When two nodes connect, they exchange negotiation packets and compare their IDs. If the two nodes were not in the same network before, their IDs will differ. The node with the smaller ID adopts the higher ID of the newly connected neighbor as its tree-ID and broadcasts this new ID to all nodes in its subtree with the smaller tree-ID.

Whenever a node receives a tree-ID packet broadcast containing an ID different from its own, it adopts this new ID. If the received ID is its own ID, there is a cycle in the network and the link over which the broadcast arrived is dropped.

If a node loses a link over which the current tree-ID was received, the node broadcasts its device address as new tree-ID for its subtree.

Algorithm 2 Connection manager: Packet-handler thread

```
loop
  packet := wait_for_packet()
  if packet.type = tree_ID_packet then
    if local_tree_ID = remote_tree_ID then
      drop connection
    else
      local_tree_ID := remote_tree_ID
      broadcast local_tree_ID to my subtree
    end if
  end if
  if packet.type = negotiation_packet then
    if local_tree_ID = remote_tree_ID then
      drop connection
    else
      if local_tree_ID < remote_tree_ID then
        local_tree_ID := remote_tree_ID
        broadcast local_tree_ID to my subtree
      end if
    end if
  end if
end loop
```

This local algorithm provides self-healing tree topologies and is expected to scale well to a large number of nodes.

3.1.3 Transport Manager

The *transport manager* takes care of multi-hop packet forwarding. It receives information about available connections from the connection manager via the `tp_reliable_con_change_cb` callback function (see Figure 4-1 on page 24).

For the transport manager the concept of *host nodes* are important. As explained before, a host node is a user controlled DSN node. Hence, the host node is a source

for commands and a sink for data from the DSN. Multiple host nodes in the same network are possible.

When communication is initiated by a host node, the network is flooded with a route request message. Each DSN node stores the connection handle over which such messages arrived on in a host table; this is the route back to the host to be used on the return path.

The transport manager provides a connectionless transport type based on this principle. The data is always broadcasted to all nodes. It can be specified whether all or only a certain device address actually receives the packet (i.e. the packet is passed to the upper layer). The DSN nodes can send back reply messages to the host which is known by all intermediate nodes at this point.

A second option is to route packets based on ATM virtual-circuit switching. When a node receives packets of this type, it automatically forwards traffic to the appropriate connection based on a virtual-circuit identifier. The connectionless transport mechanism explained above is used to establish these virtual connections between any two nodes in the network. Virtual connections are efficient for large data transfers.

In case of link failures, the host and all endpoints of broken virtual connections are notified, and all virtual-circuit identifiers removed from the local tables. In addition, when a node loses a link to a known host node, it informs the nodes in its subtree about the loss such that they can delete this host from their host tables.

Retransmission of lost packets in case of link failures is not performed and has to be taken care of by higher application layers.

3.2 Remote Command Execution

The transport layer of the existing JAWS provided already a simple interface (`tp cmd`) to execute registered terminal commands on remote DSN nodes. We wanted to extend this functionality to address the following issues. First, an important missing feature was command output. The existing mechanism merely executed the command but did not collect and send back the output to the source node. Secondly, time-consuming commands were a problem. `tp cmd` is direct part of the transport layer and therefore execution of a command blocks other transport activities such as packet forwarding.

We defined a new remote command execution service which is able to do the following (without affecting the transport layer):

- Send a command string to remote nodes. Unicast as well as broadcast shall be supported.
- Execute the command on the specified remote node(s) and collect the output (if any).
- Send the result back to the host node.

3.3 Target Programming

Loading new program code into the target's flash memory is one of the key features of a DSN. Basically, three steps have to be performed:

1. Load the program from the PC to the wired host node.
2. Distribute it to every other DSN node over Bluetooth.
3. Let one (or all) DSN nodes flash their targets with the received program.

3.3.1 Upload to the host node

As explained in Chapter 3.1, the host node is connected to a personal computer by a serial connection over USB. The target program is stored in an Intel HEX file. Our idea was that the user should be able to transfer this file to the host node directly over the serial interface for simplicity. Terminal emulator software typically supports sending of a plain text file (which a Intel HEX file is). Thus, it is possible for the user to upload her programs without additional tools. The host node has to know the Intel HEX format and converts the incoming program into the bootloader format (see Section 4.3.1). We defined a new terminal command called `loadhex` for this task which is explained in Section 4.3.2.

3.3.2 Flooding the network

The next step is to send the program from the host node to all other nodes. The desired features of this process are:

- *Robustness*. Changes in network topology or packet loss should not interfere with code distribution.
- *Short distribution time*. All nodes should receive the uploaded program as fast as possible.

A first idea was that the host node uses broadcast packets to send the program to all other nodes simultaneously. This approach would have been fast but not very reliable. Especially because one major problem we had to face was packet loss due to missing flow control in the Bluetooth chip (see Section 4.9.2). Sending the packets too quickly leads to congestion at nodes that have to forward the data to more than one link. Another problem is when the topology changes due to link loss. The nodes in the subtree that have been disconnected will receive the program only partially – if at all.

To cope with these problems we followed a different approach (Algorithm 3). Each node independently advertises its own program to its immediate neighbors. If a node has a program in memory it will send information about this program from time to time to its neighbors. The neighbors decide based on the given program information, whether they want to download it from the advertising node or not. In our case, the decision is based on the program version. After some time, all nodes will have the newest one. If transmission of a program between two neighbors failed[1], the node will try to redownload automatically when the program gets advertised again.

[1]Because the receiving node doesn't have to forward these packets, transmission errors due to packet loss (because of congestion) are not very likely, provided that there is not much other traffic.

The main characteristics of this code distribution scheme are:

- Simple and robust.
- Completion time depends on the topology.

Algorithm 3 Code Distribution (Sender)

```
loop
  wait some time
  for for all neighbors n do
    send program information to n
    wait for reply with timeout
    if reply = ack then
      send program to n
    end if
  end for
end loop
```

3.3.3 Target Flashing

In the final step, the DSN nodes flash their targets with the new software. The target has to be attached to the DSN node over a wired connection suitable for programming. Based on our task description, we concentrated on targets based on the Atmega128l microcontroller. This includes, besides the BTnode, also the Mica2 and Mica2Dot.

There are four possibilities to program this type of microcontroller:

- *Serial interface (UART).* The target is flashing itself with the program data it receives over a serial interface. Programming this way works only if the target has special software installed that performs the flashing.
- *JTAG.* This interface is often used on microcontrollers to provide debugging capabilities and provides a fast programming mode. It occupies four pins when enabled. On the Atmega128l, all of these pins lie on the ADC port. This may restrict the target's possibilities to use sensor or sensor boards.
- *Parallel Programming.* This is a high-voltage parallel programming mode using many pins. Not intended for our purposes.
- *Serial Downloading.* Memory programming over the SPI programming interface.

We have chosen Serial Downloading for the following reasons:

- It does not depend on the firmware running on the target.
- The pin mapping does not collide with any target functionality of the BTnode.
- It uses an instruction set that is simple to implement.
- In a former semester thesis [14] some experiences where already made with Serial Downloading on the BTnode.

3.4 Self-Programming

Being in active development, it is a desired feature that the DSN is able to upgrade itself. A way to distribute program code was already presented in the previous section. To reprogram themselves the nodes can use the boot loader feature of the Atmega128l.

The program memory on this microcontroller is divided into application and boot flash section (Figure 3-2). The size of the boot flash section can be configured by fuse settings. It can be used to hold a small, resident program that writes to the application section.

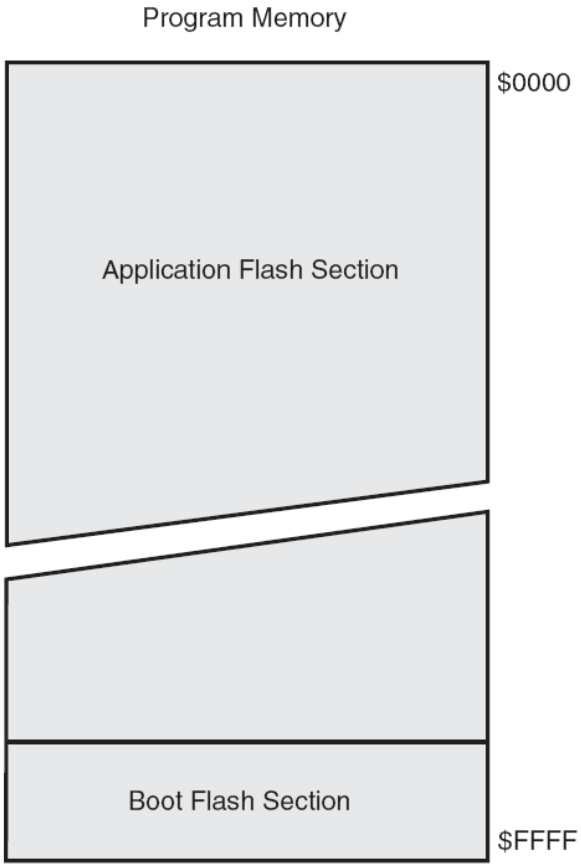


Figure 3-2
Atmega128 program memory layout [4]. The reprogrammable flash memory is organized as 64k x 16. The memory space is divided into an application and a boot section.

The BTnut system provides such a *bootloader* application. On system reboot, it checks the serial interface and the storage memory for new firmware. If there is a new program, it is transferred to program memory and started.

So, once the program is distributed, self-programming can be done by resetting the nodes.

3.5 Target Monitoring and Control

Besides reprogramming, the DSN should also provide means to monitor and control the targets. For the monitoring part we wanted to reuse the planned logging system by defining a special log class that receives all output from the target.

To allow these concepts, further communication between the two is required. Possible options to connect them depend on the type of the target and are discussed in Section 3.7.

3.6 Logging

Log messages consist of a log class, a log level and an arbitrary string, usually a human-readable text message. The log class specifies the origin of the message and provides a way to distinguish different message types. The log level marks the severity of the message, ranging from critical errors which compromise the system up to debug information.

A set of log masks shall be used to determine the behavior of each pair of log class and level. A log mask consists of one bit for each level and specifies whether the associated operation should be performed (1) or not (0).

The log module shall further provide services to capture, store and show log messages generated by both the target node and the DSN node. The services shall also include a mechanism to send all or specific log entries to a remote host. This mechanism should be triggered by hand or automatically. Thus, allowing important messages to be sent to the host immediately.

3.7 DSN Adapter Board

One goal of this thesis is to develop a concept for the integration of DSN nodes with target sensor nodes as depicted in [6]. An experimental connection was used in [14], however no large-scale solution is available up to now.

The first question that arises is which target platform to support. The most obvious and convenient solution is to use BTnodes for the DSN network as well as the target network. Particularly since the BTnode device was developed at the ETH Zürich and is the most used platform among research groups involved in wireless sensor networks at our university. The Berkeley Mica [13] and Mica2 Motes are another, even more widely spread platform (as mentioned in Section 1.3). Fortunately, the BTnode and the Mica Motes are quite similar as both feature an Atmel ATmega128l microcontroller, and may be connected likewise to a DSN node. This is the reason, why we opt for a solution which can be used to connect BTnodes and Mica Motes interchangeably.

Another emerging wireless sensor platform is Telos [18], the latest low-power device developed at the UC Berkeley. As its interfaces are quite different, our target adapter is not required to be compatible to the Telos platform.

There are various methods to connect two ATmega128l featured devices. One can distinguish between the way to program a target node and how to communicate at

runtime. The programming methods are described in Section 3.3.3. As stated there, Serial Downloading is the only feasible method for programming a target node in our environment.

Then, the following ways to communicate with the target node exist:

- Connection of custom pins. This would require the definition and implementation of an own communication protocol. This may be the best method for a simple communication scheme or for some special purpose cases.
- Two-wire Serial Interface (TWI). Simplex, device addressing, requires two bidirectional lines. Compatible to the I^2C bus by Philips.
- Serial Peripheral Interface (SPI). Duplex, no addresses, requires four lines. Similar to TWI, but more data streams oriented.
- UART serial interface. Requires two wires, four if hardware flow control is to be used. Widely used standard interface.
- JTAG Interface. Complicated, a controller would need to be implemented in software. Albeit it would possibly provide the best debugging facility.

The serial communication using an UART port is the simplest connection method that still provides enough flexibility and bandwidth to allow for a sophisticated debug and status interface. As almost every computer has a serial port, it is also easy to debug the target interface itself during the development phase. Additionally, the data transmit and receive pins of the target's first serial port already have to be connected to the DSN node for the programming interface. For these reasons, we have chosen the UART serial connection to connect target nodes and DSN nodes.

A first hardware proposal consisted of a custom PCB (Printed Circuit Board) connecting the DSN BTnode with either another BTnode or a Mica Mote as target sensor node. This board would have had connectors for the nodes as well as for a Mica sensor board featuring a light sensor, temperature sensor, microphone with tone detector, sounder, magnetometer and an accelerometer. However, as the extension connecters mounted on the BTnode and the Mica Mote are not the same, it would have been necessary to develop either a board with both connectors or two board variants.

The final hardware design is based on a wired connection between the nodes. An extension board will be attached to the DSN node as well as the target node. Both boards have a connector that meets the 6-way SPI pin-out specification used by Atmel. A standard 6-pin cable is then used to connect two nodes. As there is already a Mica adapter using the same SPI interface, this solution is compatible to the Mica platform per se. Other compatible devices include the Atmel STK500 developer kit and some ISP programmers. For the BTnode, an adapter board has to be designed. This solution is way more flexible as it not only allows to connect BTnodes and Mica Motes without a change in hardware, but it can also be used as an interface for future devices. We thus implement this connection method.

An important aspect of a permanent installation of DSN nodes is the power supply. The aim is a distributed power system. Although it should be possible to supply each node with current from batteries, changing them every week – or even every day –

is not practicable. Therefore, the DSN adapter board has to provide a connector for an external power supply.

4

Implementation

4.1 System Overview

In Figure 4-1, an overview of the JAWS software is depicted. As can be seen, JAWS uses a modular structure. Transport and connection manager are described in Section 3.1. The `BTstack` thread is part of the BTnut system software. It provides lower level HCI functionality such as receiving and dispatching of HCI events and data packets. The `con_event_buffer` buffers connection events that are later on processed by the connection manager. Transport and connection manager are registered as different services and process their packet types independently.

The DSN service layer and the target adapter are new components that have been implemented in this thesis. The DSN layer uses the multi-hop packet routing functionality of the transport manager to provide different services. The target adapter is a local module that interfaces with the attached target.

In the following of this chapter the new services in the DSN layer and the target adapter are described in detail.

4.2 Remote Command Execution

This section describes implementation aspects of the remote command execution service defined in Chapter 3.2.

4.2.1 Overview

The service registers the following terminal command:

```
dsn cmd <trans-id> [<addr>] [<remote-cmd>]
```

The `remote-cmd` is a normal command string. If this argument is omitted, the `dsn cmd` will ask the user explicitly to enter a command string.

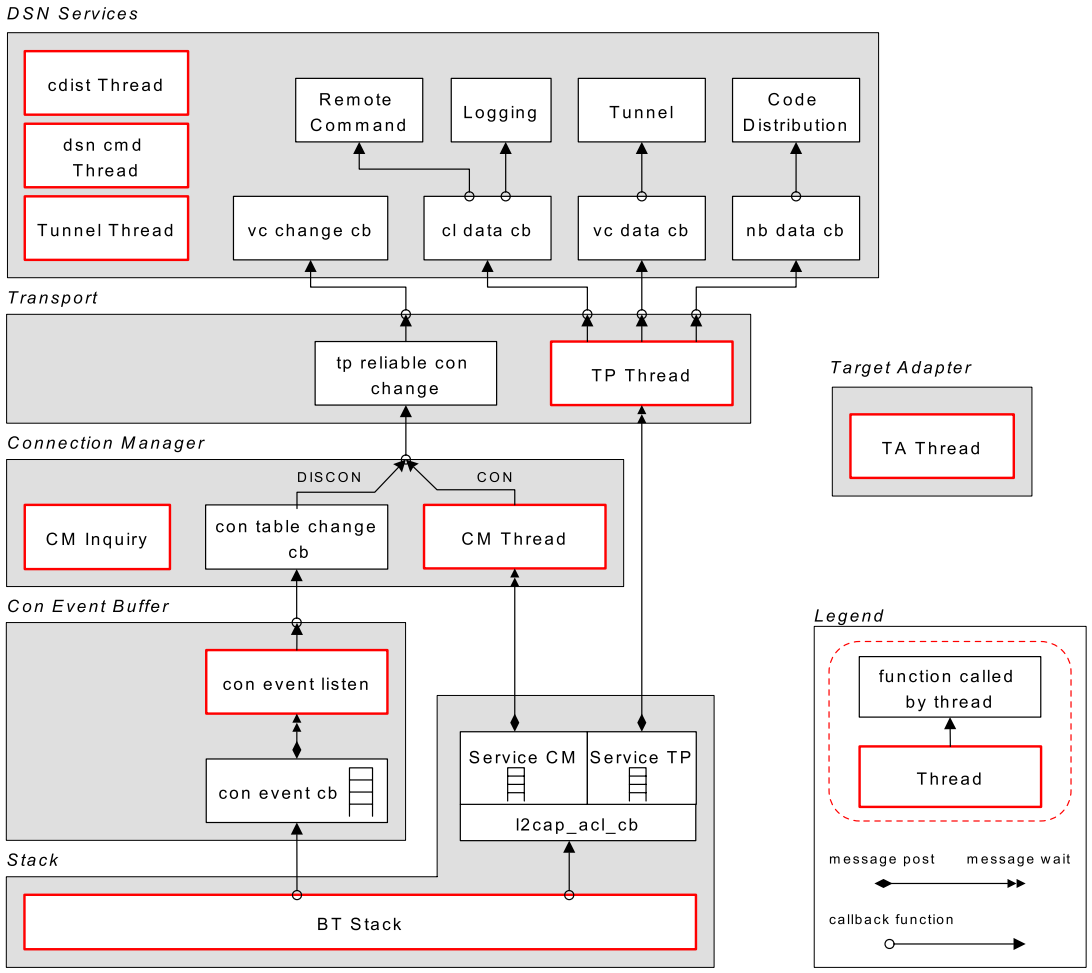


Figure 4-1
JAWS Overview (receiving side).

The `addr` is the remote Bluetooth address where the command shall be executed. If this argument is omitted, the command is sent to all nodes (broadcast).

Now, the `trans-id` field deserves some special attention. Imagine that the user sends a rather time consuming command such as flashing the target of a DSN node which takes approximately 30 seconds. It would be a waste of time to block the terminal just for waiting for the remote command to complete. Therefore, we wanted the `dsn cmd` to behave asynchronously, i.e. the command terminates after the command is sent to the target address. Receiving of the command output is handled in the background by the transport callback `cl_data_cb`. This approach allows the user to issue other terminal commands (possibly `dsn cmd` on other target addresses!) while a `dsn cmd` is in progress on a remote node. In order to distinguish the incoming command outputs we introduced a *transaction ID*. It is a number which is sent together with the command string. When a node receives a command string to execute, it will send back this number together with the command output to the source node. On the source node, the command output can now be mapped to a previously issued `dsn cmd` based on this transaction ID.

4.2.2 Transport Layer Extension

An important question for the implementation was how to realize it on the transport level. As already described in Section 3.1.3 the transport layer provides the concept of virtual connections for multi-hop communication. Despite of its efficient data transfer we didn't consider it as a viable solution because of the following reasons:

- The presented remote command execution mechanism is not intended to transmit large amounts of data. The command strings are short and fit into one packet. The result is not very long either, usually consisting of a few lines of terminal output. Opening and closing a virtual connection (involves broadcasts of transport manager commands) would mean a large overhead compared to the actual data transferred.
- Broadcasting of remote commands would have caused some problems. Either the host node has to sequently open and close temporary virtual connections or it has to open and maintain permanent virtual connections to every DSN node. Both variants were considered to not scale very well.

The transport layer internally uses special broadcast packets to do its job (e.g. to establish a virtual connection). The terminal command `tp cmd` uses this packet type to send its command string in one broadcast to all nodes. We extended the transport manager API by the function

```
tp_broadcast_cl_data( bt_l2cap_pkt_t *pkt,
                      bt_addr_t destination,
                      u_short len );
```

to access this functionality directly. The packet pointed to by `pkt` of length `len` is sent to all nodes. And every node that forwards these packet adds the source node to its host table. The `destination` argument defines, whether all or only a specific node passes this packet to the higher layer by calling the registered `cl_data_cb` callback function.

To send a reply, the nodes can use the already existing `tp_send_cl_data()` function. This function is able to send data back to a host without broadcast by following the host table entries as explained before in Section 3.1.3.

Comparing this connectionless data transfer scheme to the virtual connection, the following points can be made:

- No connection setup overhead.
- Easy broadcast.
- Slightly reduced payload size, because broadcast packets have a larger packet header. As this transport type is not intended for high data traffic, this is not considered as a problem.

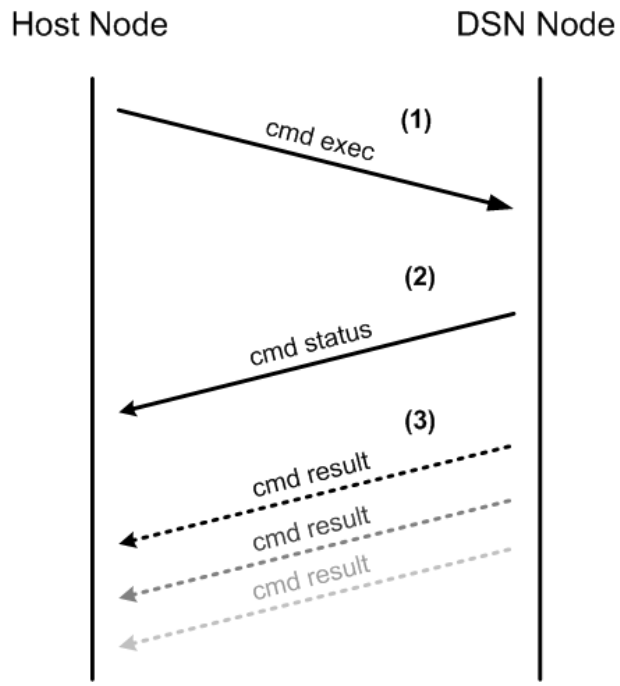


Figure 4-2
Messages when executing a remote command.

4.2.3 Messages

Figure 4-2 shows the message flow. First, a *cmd exec* packet (1) is sent to the target node. It contains the command string and the given transaction ID. The receiving node will always send back a *cmd status* packet (2). This status packet contains again the transaction ID and an error code to indicate, whether the command was successfully executed or not. On success, it will additionally contain the length of the command's output (may be zero). In the current implementation the output is just a string for every command containing what it would print, if the command was executed directly on the terminal. This string is returned in *cmd result* packets (3). Because it does not necessarily fit into a single packet it can be split up into several ones. Each *cmd result* packet contains:

- *Transaction ID*.
- *Sequence number*. For every command output the result packets are sequenced starting at zero. This allows to recognize if a part is missing or if the packets are disordered.
- *Length*. The length of the output part in this packet.
- *Output data*. This is a null terminated substring of the complete output string.

The status and result packets are printed in a parsable format to the terminal when they arrive. Depending on the error code the *cmd status* message is printed differently:

```
:C <source-addr> <trans-id> completed: <total-len>
```

```
:C <source-addr> <trans-id> failed: <reason>
```

The first format is used if the command was successfully executed, the latter if an error occurred. `reason` stands for a short string describing the cause of the error (Table 4-1).

Reason	Explanation
`busy`	The destination node is already executing a command. Try again later.
`cmd not found`	The command string is not a registered terminal command.

Table 4-1: Possible DSN cmd errors reported by cmd status messages.

The *cmd result* messages (if any) are printed as:

```
:CO <source-addr> <trans-id> <seq-nr> <len>
<cmd-output-data>
```

4.2.4 Control Flow

Figure 4-3 illustrates the control flow in our implementation. As can be seen, incoming data is handled by the transport thread callback. A dedicated `DSNCmd` thread executes incoming commands. Only one command can be processed at a given time.

4.2.5 An Example

Now, let's assume the following scenario: The user is connected to the host node A and wants to retrieve the battery voltage of a DSN node B with address `00:04:3f:00:00:87`. She could enter the following command on node A:

```
dsn cmd 3 00:04:3f:00:00:87 get bat
```

If the command was successfully executed the user sees the following output printed on the terminal:

```
:C 00:04:3f:00:00:87 3 completed: 24

:CO 00:04:3f:00:00:87 3 0 24
Battery Voltage: 2.57 V
```

Or, if node B was already executing a command from another host:

```
:C 00:04:3f:00:00:87 3 failed: busy
```

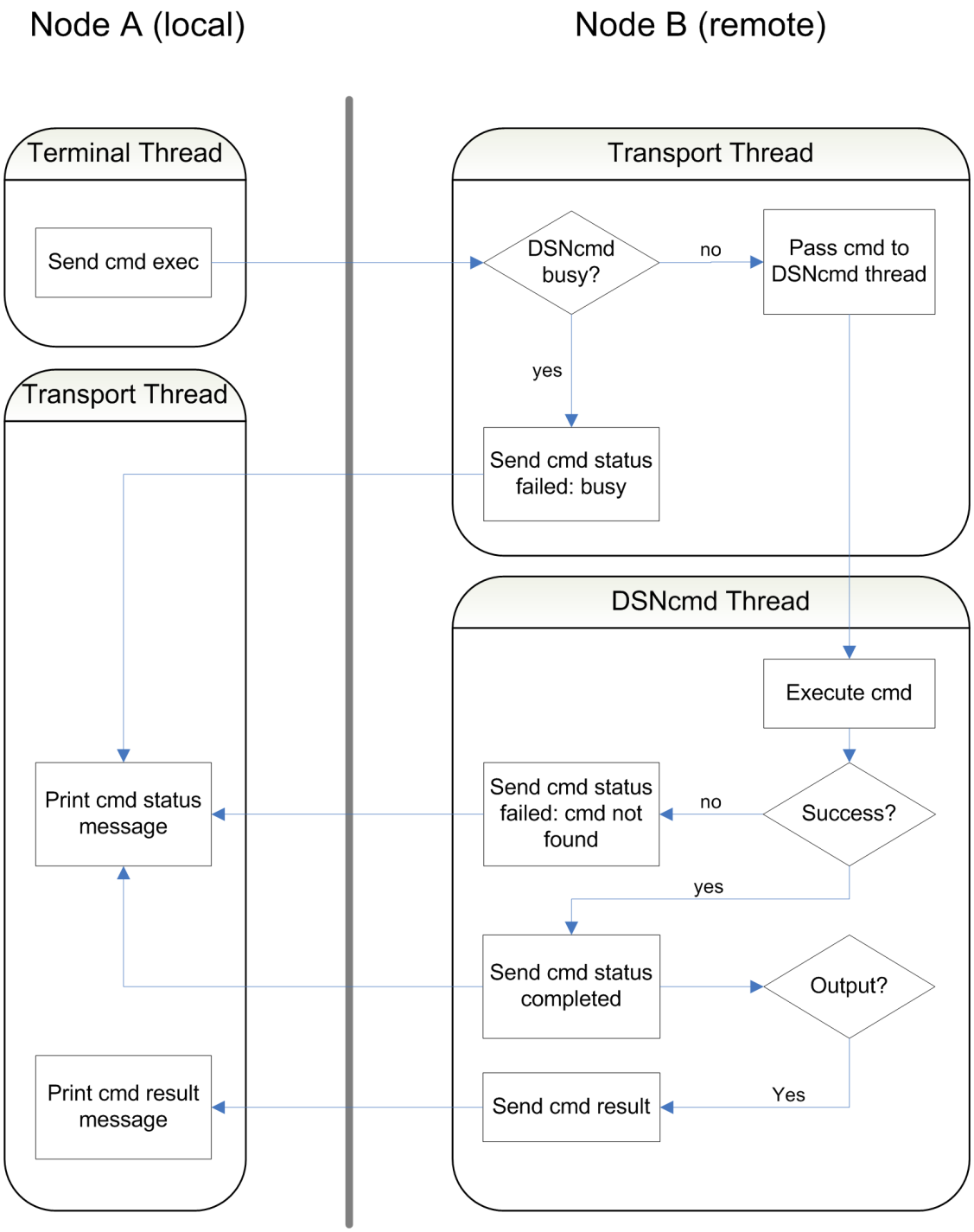


Figure 4-3
DSNcmd control flow and involved threads. The user on node A sends a command for execution to node B.

4.3 Program Management

As described in Chapter 2, the BTnode features 180 kB storage memory. In JAWS we have two applications for this additional memory space: program and log data. The first 129 kB are reserved to store a program in bootloader format (see Section 4.3.1). Due to the record based format it is possible that the space needed for a program exceeds 128 kB. This is why an additional kilobyte is reserved.

Additionally, the following information is stored on every node:

- *Type*. We differentiate whether the program is intended to flash the target or the DSN node itself.
- *Version*. This is a four byte integer number. Note that we don't store the real program version here. The implemented code distribution scheme favors version numbers which are higher than others. Therefore, this number has to be increased when the user uploads an other program, because she expects that this program will be distributed regardless of its real program version. The graphical JAWS user interface fills this number with the current upload time. This ensures that the most recently uploaded program is always distributed.
- *Size*.
- *Name*.
- *Boot address*. Depending on whether the stored program is or includes a boot loader application the fuse values on the ATmega128l have to be changed when flashing the target. The boot address is determined automatically from the uploaded HEX file.

The stored program might be accessed by different threads. To avoid corruption the threads must acquire a program lock before they read or modify the program in the storage memory. Getting the program lock is a non-blocking process.

The file `xbankdata.h` defines an interface for accessing programs in banked memory.

4.3.1 The Bootloader Application

The BTnut system provides an application called *bootloader* that fits into the boot flash section (Fig. 3-2). Its main purpose is to flash the BTnode with new application software that is transferred over the serial interface. As a second option it can load program data from storage memory (SRAM). There, the program has to be in the bootloader format to be recognized. It is defined as:

Field	`type`	`version`	`active`	`proglen`	`crc`	`progdata`	`crc`
Bytes	1	4	1	4	2	`proglen`	2

where

`type` is the program type.

`version` is the program version.

active is a special flag that indicates wether the bootloader shall write this program to the flash section or not.

proglen defines the length of the program data.

crc holds a checksum over the program header fields above.

progdata contains the program code in a record based format. Each record consists of three bytes defining the length (`lll`) of the record and three bytes defining an address offset (`aaa`) for the data followed by the actual program data (`d`):

```
lllaaa[d...]
```

crc contains a checksum over the program data. A program is only valid if both checksums are correct.

We reused the bootloader program format to store programs that are distributed in the DSN. It allows the bootloader to do the required self-programming: When the BTnode is rebooted the bootloader becomes active and writes the firmware to the program memory – provided that `type` and `active` fields are set correctly.

4.3.2 Loadhex

In the AVR world, compiled programs typically are stored in the *Intel HEX File Format*. This format is a way of representing binary object files in ASCII. The file is blocked into records of different types containing all needed data. Each record is based on the following hexadecimal format:

```
:llaaaatt[dd...]cc
```

where

: is the record start mark.

ll specifies the number of data bytes `dd` in the record.

aaaa defines the 16-bit starting address offset for the data bytes `dd` in the record.

tt is the record type. The GNU-AVR utilities make use of the following record types:

- **00** Data record.
- **01** End of file record.
- **02** Extended segment address record. This record type is required to place data at addresses greater than 64k.
- **05** Start linear adddress record. Defines the start address for execution on system power up.

dd represents one byte of data in hexadecimal format. A record may have multiple data bytes. The number of data bytes in the record must match the number specified by the `ll` field.

cc contains a checksum over the record. The checksum is calculated by summing the values of all hexadecimal digit pairs in the record modulo 256 and taking the two's complement.

We implemented a terminal command `loadhex` that reads a HEX file from the serial interface and writes the program code to SRAM in the afore mentioned bootloader format (Section 4.3.1).

To optimize performance, every record is read in two steps:

- Read the first 9 characters and convert length, address and type field to binary.
- Read the rest of the record including the checksum field. The number of bytes to read depends on the length field.

When uploading the HEX file, we measured data rates of about 4.9 kB/s. This would allow to load a binary program in less than 30 seconds. However, the hexadecimal data representation basically doubles the amount of data to be transferred.

4.4 Code Distribution

In Section 3.3.2 a simple code distribution algorithm was introduced. This chapter explains the integration with JAWS.

4.4.1 Overview

Regarding the code distribution, each DSN node is in one of three states (Figure 4-4). Normally, the node resides in the ready state. From there it can change to either download or advertise mode. Advertising is done in a separate `cdist` thread. Downloading is handled by the transport callback function.

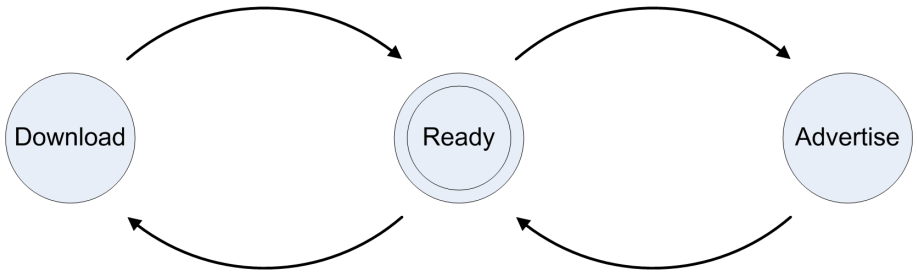


Figure 4-4
Code distribution states. The nodes can not download and advertise at the same time.

The `cdist` thread sleeps most of the time waiting for an event to take action. Possible events that wake up the `cdist` thread are:

- The maximal waiting time of 60 seconds is over. All nodes advertise at (more or less) regular intervals.
- A new program has been downloaded. This ensures that new programs are spread as quickly as possible.

When running, the `cdist` thread tries to switch the state to advertising. This fails, if the node is not in the ready state, i.e. the node is downloading a new program. Once switched to advertising, the code distribution Algorithm 3 on page 18 is performed. If there is no program to advertise the thread does nothing.

4.4.2 Transport Layer Extension

The presented code distribution scheme requires communication between immediate neighbors in the network topology. Data transfer has to be efficient because programs should be transferred as fast as possible. Because the transport layer featured no suitable mechanism to do this, we extended it by a new packet type *neighbor data* with the following characteristics:

- Sent to direct neighbors only.
- No routing overhead.
- Small packet header.
- A certain neighbor is addressed by the corresponding connection handle.

Some new functions were added to the transport interface. To get a list of neighbors one can use `tp_get_rel_cons()`.

```
tp_send_nb_data( bt_l2cap_pkt_t *pkt,
                 bt_con_handle_t con_handle,
                 u_short len )
```

This function works analogously to the sending functions for connectionless and virtual connection data. To send a neighbor data packet:

1. Allocate a packet.
2. Use `tp_get_nb_data_pointer()` and fill in the data.
3. Use `tp_send_nb_data()`.

To receive such packets one can register a callback function the same way as with every packet type.

4.4.3 Messages

This section describes in more detail how programs are transmitted. Have a look at Figure 4.4.3. It shows the message flow when a node advertises a program.

First, it sends a *prog info* packet (1). This packet contains all relevant information the node stores about it (cf. Section 4.3):

- Program type
- Version
- Size
- Name
- Boot address

Based on this program information the neighbor sends back a *reply* packet (ACK or NACK) indicating whether it wants to receive the program or not (2). The *reply* is of type ACK only if the advertised program version number is higher[1] and the

[1]The neighbor accepts every version number if it has no program yet.

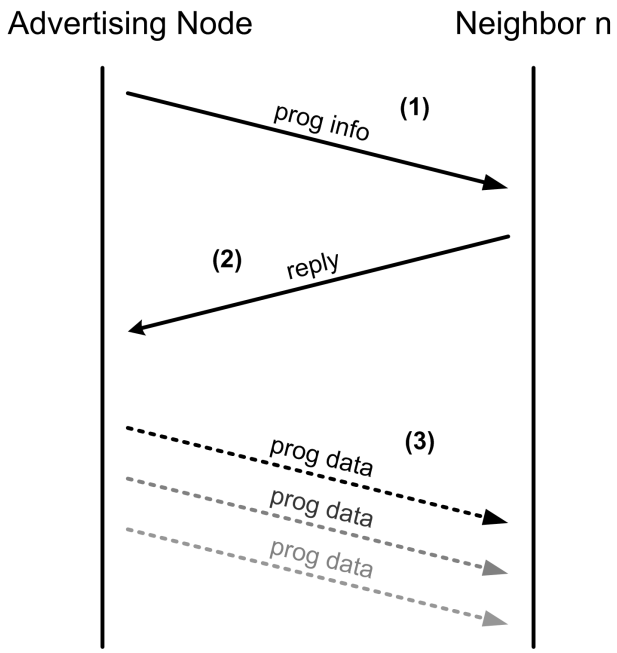


Figure 4-5
Code distribution message flow. The node on the left advertises its program version to one of its neighbors.

neighbor is ready to receive the program at the moment. Reasons why a node might not be ready to receive a program include:

- Code distribution state is not ready. That means that the node is already downloading from an other node or that it is advertising and possibly uploading its program to one of its neighbors.
- The program lock has been set.

In these cases the neighbor returns a NACK reply. Otherwise it prepares to receive a new program. The code distribution state is switched from ready to download and any previously existing program is deleted.

When the advertising node receives an ACK reply, it starts transmitting the program (3). The program is stored contiguously in the storage memory. The start address is always the same. The end address depends on the program size. This memory area is split up into equal sized chunks and sent in *prog data* packets to the receiver where they are reassembled and copied to the same memory locations. The receiver knows the size of the program, therefore it is able to recognize the last packet. When it does so it checks header and data CRCs of the received program and sets the code distribution state to ready.

To further ensure correct transmission the *prog data* packets contain a sequence number. Thus, the receiver notices immediately when a packet is lost. In this case, it terminates the download and returns to ready state.

Timeouts when waiting for reply or *prog data* packets guarantee that the nodes do not get stuck in either advertise or download mode when the remote device does not

respond. For both packet types the timeout is set to 10 seconds by default.

4.5 Target Adapter

The target adapter provides functionality to interface with the attached target device. It provides terminal commands for target control and communication as well as some simple monitoring features. The file `target_btnode.c` contains our implementation for BTnode targets.

4.5.1 Reprogramming

In Section 3.3.3 we chose to reprogram the targets by the Serial Downloading feature of the Atmega128l. It uses the serial SPI bus for data transfer. SPI provides fast, full-duplex data transfer between two endpoints. One is the master and the other is the slave. The data is transferred over the two MISO/MOSI lines simultaneously. The former sends data bits from the slave to the master, the latter from the master to the slave. The transmission is synchronized by the master's clock rate (SCK). The slave select (SS) line is only used in normal SPI operation, not in SPI programming mode. Therefore it remains unconnected. In our scenario the target is the slave and the DSN node is the master.

The SPI programming mode is activated on the target if the Atmega128l is reset, i.e. the RESET pin is set to ground. Incoming data on the SPI bus is now interpreted as serial programming instructions. The available instruction set allows to:

- Read, write or erase program memory.
- Read, write or erase EEPROM.
- Change fuse and lock bits settings.

For a complete list of available instructions have a look at [4]. We implemented a small SPI programming library (files `spiprog.h` and `spiprog.c`) which is used by several target adapter commands.

The target adapter provides the `tg flash` terminal command. It checks, whether a target program is available in storage memory. If this is the case, it uses the above mentioned library to reprogram the target over SPI. In addition, it sets the correct fuse settings according to the boot address (confer Section 4.3). When the command finished programming, it prints `:TF ok` on success or `:TF failed: <reason>` if something went wrong. Again, `reason` is a short error string. In Table 4-2 a list of `tg flash` specific errors is given. Additionally, all commands that use the SPI programming mode report the errors in Table 4-3.

Changing Atmega128l fuse settings needed for bootloader support is handled by `tg flash`. Additionally, the target adapter provides commands to read and set custom fuse values.

4.5.2 Monitoring

The target adapter contains a special thread that listens on the first serial interface (UART0), where the target is connected. Every received line is passed to the logging

Reason	Explanation
`program locked`	Because an other command is already accessing the program, the program lock could not be acquired.
`program check`	There is no valid program in the storage memory.
`program type`	There is a program, but it is not a target program.

Table 4-2: Possible errors reported by `tg flash`.

Reason	Explanation
`spi busy`	The SPI bus is already used by an other command.
`spi error`	Could not enable programming mode on target.
`cpu signature`	The target is not a Atmega128l with 128 kB flash

Table 4-3: Possible errors reported by all commands that use SPI programming mode.

system. This allows to capture debug or other output from the target and can be sent to a host node later on.

Section 3.5 describes the concept of a monitor in the target application that provides status information. Figure 4-6 illustrates the situation. We implemented such a monitor in the file `monitor.c` for BTnode targets. It consists of a set of terminal commands summarized in Table 4-4.

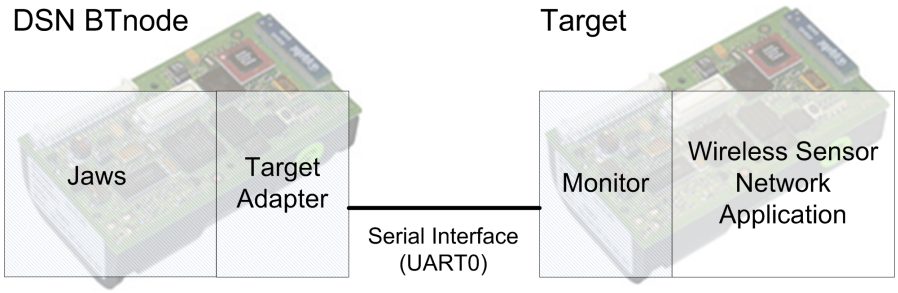


Figure 4-6
A special monitor hooks into the target application and provides status information to the target adapter on the DSN node. Communication is line-based via the first serial interface (UART0).

The `mon reg` command is able to print the memory content at a given address. Besides data memory, it is also possible to print the memory mapped contents of ports and registers. A decent application is to print status registers or to check whether an I/O pin is currently high or low on the target.

The `mon cmd` command executes a given command string as the terminal would do. The difference is that it inserts start and end markers for the command output. Additionally, it prohibits that other threads write to the UART. This ensures that the command output is not intermixed with output from other commands. The start and end markers clarify which lines belong to the same command output when it is several lines long.

Command	Explanation
`mon bat`	Print battery voltage.
`mon cmd <cmd>`	Execute `cmd` and mark its output (see text).
`mon heap`	Print free heap memory.
`mon irq [<nr>]`	Print interrupt counters.
`mon net`	Print packet statistics
`mon reg <addr>`	Print memory location at address `addr`.
`mon threads`	Print thread list.
`mon timers`	Print timer list.
`mon ver`	Print version string.

Table 4-4: List of available monitor commands.

The `mon ver` command is used in conjunction with the `tg get version` command. `tg get version` issues the `mon ver` command on the target and waits at most 2 seconds for the corresponding output. The output of `mon ver` is filtered out by the target adapter thread and passed to the waiting `tg get version` which finally prints the version string.

4.5.3 Target Control

The DSN Adapter Board allows to switch power on and off for the target, if it is not running on its own batteries. The target adapter provides commands to control target power and for reset.

The `tg cmd <cmd>` command allows to send an arbitrary text string (`cmd`) to the target. If the target application uses the BTnut terminal library, this string will be interpreted as terminal command. This makes it possible to control the targets with commands they support.

4.6 Logging

Log classes are primarily bound to the different modules sketched in Section 4.1. Table 4-5 describes the different classes. There are two special classes, first, LOG_TERMINAL is used for any terminal output. That is, all output generated as a response to a terminal command as well as everything needed by the GUI (see command specification in Appendix B). Second, the class LOG_TG captures any output from the target node.

Log Class	Description
LOG_TERMINAL	Terminal and GUI messages.
LOG_CM	Connection manager module.
LOG_TP	Transport manager module.
LOG_DSN	DSN service layer.
LOG_TG	Output from target, if connected.
LOG_JAWS	General JAWS class.
LOG_TA	Target Adapter.

Table 4-5: Log classes in JAWS

Table 4-6 shows the different log levels in detail.

Log Level	Description
LOG_ERROR	Severe errors that impact the system.
LOG_WARNING	Warnings. Indicate a non-critical failure.
LOG_INFO	Informational messages.
LOG_DEBUG	Debug output intended for developers.

Table 4-6: Log levels in JAWS

As log messages make up a good deal of the overall program size, we introduced the concept of static and dynamic log masks. Static log masks are used to define which log classes and levels are included in the final program code. Only these pairs of class and level can be stored and/or displayed later on. This allows to exclude partial or all log information in the final program code and thus saving precious program memory.

Then, there are three dynamic masks which control the behavior during runtime. Namely, these are:

- *logmask*. Defines whether a message is written to the log storage or not.
- *verbosity*. Messages will be printed to the terminal if this mask is set.
- *sendmask*. If a valid host address has been provided, messages to levels with this mask set will immediately be sent to this address.

Several functions are provided to handle log messages:

```
int log_line( u_char class, u_char, level, PGM_P fmt, ...);
```

Used to add a new log message to the log storage. The parameters `class` and `level` specify the log class and level of the message. The format string `fmt` and additional arguments are handled the same way as in `printf()` (Standard C library).

```
void log_show( u_char class, u_char mask);
```

This function prints all log messages which match the given class and level. The arguments may be zero, indicating that all messages of that class or level should be printed.

```
void log_send( bt_addr_t host_addr, u_char class, u_char mask);
```

Sends log messages to the given host address. Again, `class` and `level` are used to specify which messages to send.

```
void log_process_log_data(bt_addr_t source, u_char* data,
                          u_short len);
```

On the receiving side, this function is used to print received log messages to a terminal according to the format description in Section B.3.

For convenience, the following macros are defined:

```
#define ERROR( class, text, ...)
#define WARNING( class, text, ...)
#define INFO( class, text, ...)
#define DEBUG( class, text, ...)
```

They are merely an abbreviation for a call to the function `log_line()` with the argument `level` set to corresponding log level.

The log storage is implemented as a ring buffer containing variable sized records. This buffer is located in storage memory and uses 10 kilobytes by default.

The record format is as follows

Field	`length`	`class`	`level`	`time`	`string`
Bytes	2	1	1	4	`length`-8

where

`length` is the length of the whole record.

`class` is the log class assigned to this record.

`level` is the log level.

`time` is the time measured in milliseconds.

`string` is a arbitrary text message of length `length`-8.

Three offset pointers are used to manage the ring buffer. These are

- *xbank_start*. Points to the first message in the buffer.
- *xbank_end*. Points one byte beyond the most recent message. Can be used to find the starting position of a new message.

- *xbank_tail*. One byte beyond the last message, that is the message with the highest memory address.

Initially, when the buffer is empty, all tree offsets point to zero (see Figure 4-7). As messages are added to the buffer, both *xbank_end* and *xbank_tail* are increased up to the point where the next message does not fit in the remaining free space (Figure 4-8). Then, *xbank_end* is wrapped around to zero, and *xbank_start* gets increased while messages are freed until the new log messages fits in. Thenceforward, messages at *xbank_start* get freed when new messages are added at *xbank_end* (Figure 4-9).

Figure 4-7
Empty log buffer.

Figure 4-8
Full log buffer, without wrapping.

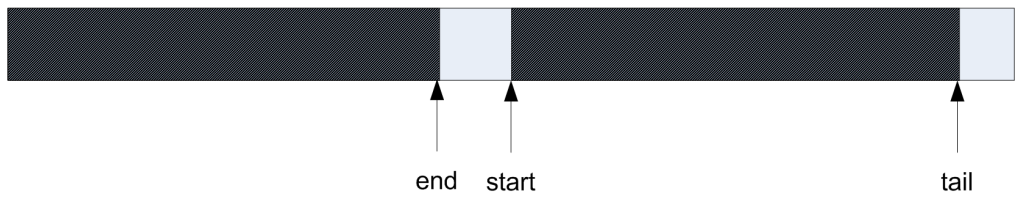


Figure 4-9
Full log buffer, wrapped around.

An additional feature of the log module are per-thread buffers. A thread can register itself using the function `log_register_thread()` and activate the thread buffer using `log_start_capture()`. From then on, all log messages to the log class LOG_TERMINAL are not only stored in the log storage or printed out (depending on the log masks), but also stored in a special buffer belonging to the running thread. Using `log_stop_capture()`, the thread can disable capturing messages to his buffer and receive a data pointer to the saved messages.

This mechanism is used by the `DSNcmd` thread to capture the output of terminal commands.

4.7 L2CAP Support

The BTnut system provides a modular Bluetooth stack to handle the Host Controller Interface (HCI) to the Bluetooth Controller. In addition, it contains two layers that build up on the HCI layer, the *acl_com* and the *l2cap* layers. Both of them can be used to send and receive Asynchronous Connection-Oriented (ACL) data packets. *Acl_com* is a simplistic (and small!) implementation using service multiplexing only, whereas the *l2cap* layer supports creating and handling channels, signalling packets as well as segmentation & reassembly of data packets and is thus compatible to the official Link Layer Control and Adaptation Layer Protocol (L2CAP) specification. However, it does not support quality of service or piconet group addresses (yet). Due to its larger functional range, it also needs considerably more program memory.

JAWS, as described in Chapter 3.1, makes use of the *acl_com* layer. One of our goals was to augment JAWS in such a way as to enable it to communicate with other Bluetooth capable devices. This makes a lot of new applications possible, such as other devices participating in the JAWS network, drop the need for a host node or running a small GUI on a PDA equipped with a Bluetooth module (as done in a current student thesis).

Since the interfaces to both the *acl_com* and the *l2cap* layers are quite similar, we chose to implement a wrapper layer. This allows to switch the ACL layer at compile time without any code changes. All functions and data types are defined as macros in the file `l2cap_layer.h`, for example `BT_CONNECT(..)` or `BT_SEND(..)`. By using the define USE_ACL_COM, the developer can choose whether these macros should be resolved to the function names of the *acl_com* layer (when USE_ACL_COM is defined) or *l2cap* layer (otherwise).

4.8 GUI

While it is possible to use JAWS as a Deployment Support Network with a terminal program only, using a Graphical User Interface (GUI) is strongly recommended, especially with increasing number of active nodes. For this purpose, JAWS contains a GUI written in Java (see picture 4-10). It is implemented as a Java servlet running on the host computer, that is connected to the host node, and a Java applet running on the user's workstation.

Originally it was written by Matthias Dyer and later on modified by Sven Zimmermann during his student thesis. Additionally, we made some small changes to adapt the GUI to the latest JAWS features.

In its current state, the GUI is able to:

- Display the connected nodes and their topology with variable zoom.
- Display various additional information, such as battery status, information about targets and program version.
- Open, close and display virtual connections.
- Send arbitrary commands to one or all nodes.

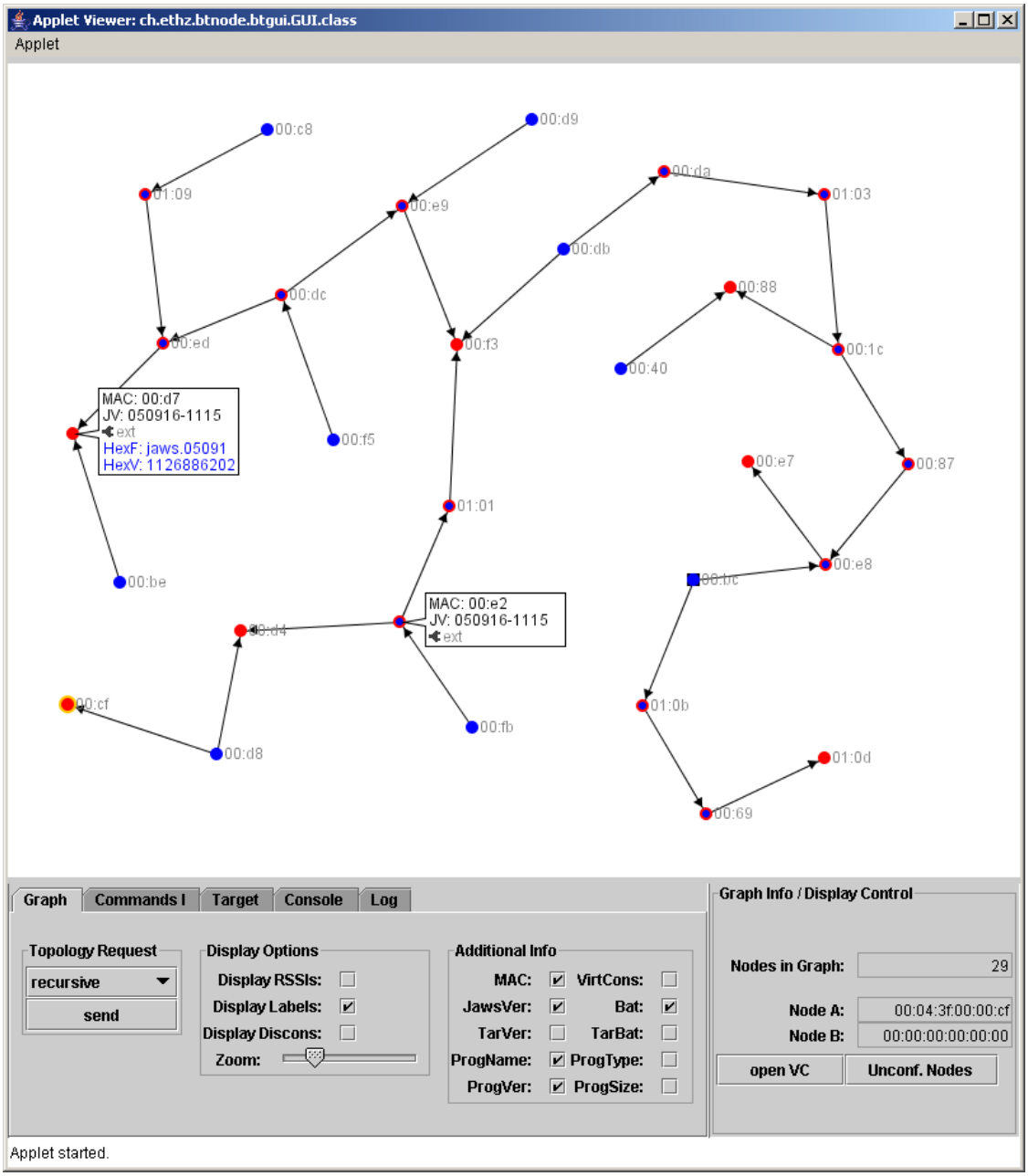


Figure 4-10
The JAWS GUI.

- Request and display log messages from remote nodes.
- Load and distribute program code.
- Issue miscellaneous commands regarding the targets, such as power on/off or reprogramming.

4.9 Problems and Difficulties

4.9.1 Nut/OS Interrupt Latency

A lot of effort during our thesis went into bug fixing and optimizations of the BTnut system software and the underlying Nut/OS. One major problem were UART overruns resulting in lost bytes. This is especially problematic on the Bluetooth UART. UART overruns occur, when the corresponding interrupts are not processed fast enough, i.e. the interrupt latency is too long. Mainly, two reasons were identified:

- UART interrupt execution time was rather long. Therefore, UART0 could starve out UART1.
- Long critical sections.

The biggest improvement was the optimization of the receiving UART interrupt handler. It was possible to almost half the execution time from 45 μs to 26 μs. The main difference was the removal of indirections over function calls which seem to be rather costly on AVR microcontrollers.

To identify long critical sections, the tracing functionality provided by Nut/OS was used. We located some frequently called critical sections that took considerably longer than 10 μs. The measurement results were forwarded to the Nut/OS developers which were able to decrease the execution times.

Finally, by working together with the Nut/OS developers it was possible to reduce the interrupt latency such that both UARTs can now be used simultaneously running at 57.6 kbit/s without problems.

4.9.2 Missing HCI Flow Control

As already mentioned in Section 2.1 the Bluetooth module (Zeevo ZV4002) is connected over the second serial interface (UART1) to the ATmega128l microcontroller. The serial interface is used to exchange HCI commands and data. Both sides manage data buffers for received packets and for packets to send. The Bluetooth standard defines a flow control mechanism that ensures that both sides do not run out of their buffers.

The microcontroller knows how many free sending buffers the Bluetooth module has by looking at the packet completed events and delays consequent packets, if necessary. In the opposite direction, the microcontroller can inform the Bluetooth controller about how many free packet buffers it has for receiving data.

Unfortunately, this HCI flow control does not work correctly on the BTnode rev3 due to a firmware bug in the Zeevo Bluetooth chip. Therefore, the microcontroller cannot

control the number of packets it receives at a time. When too many packets arrive, it has to discard them.

This caused some problems because:

- The current JAWS connection manager and transport layer implementation are not packet loss aware.
- Code distribution was affected. Big program transfers were likely to fail because of discarded packets.

In order to reduce the impact of packet loss we decided to reduce the baud rate between Atmega128 and Bluetooth module such that the microcontroller is always fast enough to process incoming packets.

As explained in the previous section, the maximal baud rate is 57.6 kbit/s considering the interrupt latency issues that arise with higher values. Several experiments were conducted to determine the maximal packet rate the Atmega128l can handle. Two devices had to transfer a maximum sized program without error under several conditions. The maximal incoming data rate that allows stable operation without packet loss was found to be around 3.2 kB/s. The next smaller baud rate of 19.2 kbit/s was then chosen as default Bluetooth UART speed. This solved the code distribution problem completely, however at the cost of reduced communication speed.

Although this measure decreases the probability of packet loss between neighboring nodes greatly, it does not completely solve the problem. Especially in a tree topology it is still possible that congestion occurs and packets have to be dropped when intermediate nodes are not able to forward the data fast enough.

4.9.3 Speed

Overall performance of the system was always an issue during our thesis. Considering that an 8 MHz microcontroller is used, one could assume that it would allow for higher throughput than the 3.2 kB/s measured above.

However, it has to be considered that the use of a fully featured operating system such as the Nut/OS involves overhead at many occasions. Due to the aimed portability, Nut/OS contains a lot of indirections and nested function calls. This is actually very nice for programming but can affect performance quite a bit as the example of the UART interrupt latency problem has shown. Then, JAWS evolved to a complex application featuring ten different, cooperatively running threads to do its tasks.

5

DSN Adapter Board

This chapter describes the DSN adapter board. An overview is given first, followed by details regarding the power options, debugging and sensor connectors. Then, the ISP connectors are further explained.

5.1 Overview

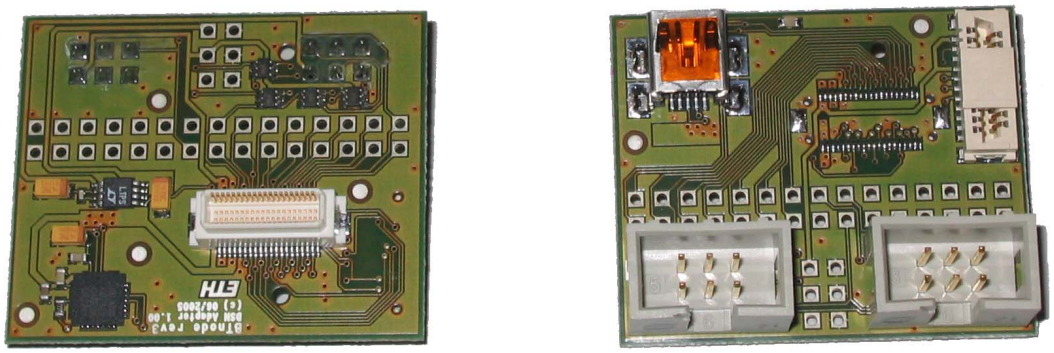


Figure 5-1
BTnode DSN adapter board, bottom view (left) and top view (right).

The DSN adapter board (Figure 5-1) is based on the BTnode USB programming adapter [20] (see Figure 5-2). The main differences are a smaller USB jack and a second (also smaller) 6-way ISP connector which is used as programming and communication interface to a target node. This plug has a switchable power line, which means that the target can either power itself or may be powered by the DSN adapter board. Additionally, the Teco sensor board connector is rearranged to the upper right corner.

A BTnode with a mounted DSN adapter board may be used as DSN node or as target sensor node as well. When used as DSN node, the 6-way cable has to be connected to the connector on the right side in Figure 5-1. When used as a target node, the left connector has to be used.

Figure 5-2
BTnode USB programming adapter.

5.2 Power

A BTnode attached to a DSN adapter board can be powered in three ways.

- Using its own batteries.
- By the power line of the mini USB jack, 3.8V to 20V, usually 5V.
- By the power line of the ISP In connector (on the left side in the top view in Figure 5-1), 2.7V to 5.5V, usually 3.3V.

Because the DSN adapter board features a power regulator and a switchable power output on the 6-way ISP Out connector (on the right side in the top view in Figure 5-1), the following scenario is possible. The DSN adapter board uses the 5V power line of the mini USB jack as current supply and feeds it to the attached BTnode. In addition, it generates a voltage of 3.3V and acts as a power supply for the connected target node. So both the DSN node as well as the target node are powered by the DSN adapter board. This setup is well suited for a long-term deployment of a larger number of nodes during development phase. Especially since USB power supplies are present on most computers today, the deployment support network may easily be placed in office buildings. During the deployment process, the BTnode and the target node may run self-contained, using batteries.

Another property of the DSN adapter board is the ability to sense the target's supply voltage. That is to say, the power line of the ISP Out connector is routed to one of

the analog inputs of the DSN BTnode. If the target node provides its own power, the DSN node is able to sense whether a target is connected. Depending on the target node, this feature may even be used to measure the target's battery voltage.

5.3 Debug (SUART)

As mentioned in Section 3.7, the first serial port (UART0) of the DSN node is used for communicating with the target node. As the second serial port (UART1) is hard-wired with the Bluetooth module, the USB controller cannot be connected to a serial port anymore on the DSN adapter board. Instead, the USB controller is connected to two general purpose pins (PE2 and PE4) to allow for an emulated serial port in software (Soft UART or SUART). The transmission speed of this interface is usually very low, however it simplifies the development of the JAWS software considerably. The standard configuration in JAWS DSN mode (see Appendix B.8) enables the SUART interface set to 2400 Baud. And the output to that serial port is kept to a minimum using the log mask described in Section 4.6.

5.4 Sensor Board

The DSN adapter board features a sensor board connector that is compatible to the sensor boards by Particle Computer [3]. Available in three variants, the fully featured board contains:

- High precision daylight and IR light sensor.
- High precision temperature sensor.
- High precision and high linear capacitive microphone.
- 2 Acceleration sensors: 3 axis, 10g max, +/- 40 mg resolution, high responsive ($<$1ms).
- Prepared for additional analog sensors e.g. pressure sensor (only sensor is missing, amplifier implemented).
- 1 LED (can be replaced by e.g. vibration motor).
- Provides Physical I/O for Core Board.
- Interfaces to Core Board via analog input, digital Input/Output lines.
- Power supply through main board.
- Size: 17x22 mm.

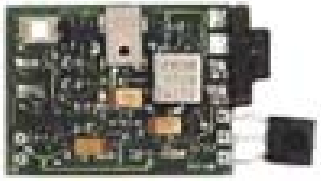


Figure 5-3
The Ssmall *Teco sensor board.*

5.5 ISP Pin Mapping

For programming and communication, two 6-way molex connectors are mounted. The first one – called ISP In connector – is used when the board is attached to a target BTnode. A cable interconnects the ISP In connector of the target board with the ISP Out connector of the adapter board attached to the DSN BTnode. This standard interface allows the SPI In-System Programming of the target node's microcontroller and is used by various ISP programming devices. The ISP pin-out descriptions are listed in the following tables.

Table 5-1 shows the pin mapping of the 6-way ISP In connector shown in Figure 5-1 (on the left side of the top view).

Pin	Name	Description
1	UART0_TxD	This is the Master In Slave Out (MISO) pin on the target Atmega128.
2	TARGET_VCC	This pin can be used to power the target node.
3	SCK	SPI Serial Clock Output. This is the SPI clock output signal.
4	UART0_RxD	This is the Master Out Slave In (MOSI) pin on the target Atmega128.
5	RESET	Active-Low target RESET control pin.
6	GND	Common ground connection between programming and target BTnode.

Table 5-1: Pin Mapping ISP In

Table 5-2 shows the pin mapping of the 6-way ISP Out connector shown in Figure 5-1 (on the right side of the top view).

Pin	Name	Description
1	PROG_MISO	Master In Slave Out. This is the SPI data input pin to the programmer
2	PROG_VCC	Output pin of the switchable power supply.
3	PROG_SCK	SPI Serial Clock Output. This is the SPI clock output signal.
4	PROG_MOSI	Master Out Slave In. This is the SPI data output pin from the programmer.
5	PROG_RESET	Active-Low target RESET control pin.
6	PROG_GND	Common ground connection between programming and target BTnode.

Table 5-2: Pin Mapping ISP Out

The full pin-out and the schematic of the DSN adapter board can be found in Appendix C.

6

Demonstrator

6.1 Installation

A casing was worked out to hold a coupled pair of nodes of both the DSN and the target network. The case is transparent so that the nodes and their respective LEDs are visible and that a possibly attached light sensor remains functional. Both nodes or only the DSN BTnode may be powered by an external electricity supply. The enclosure is shown in Figure 6-1.

Figure 6-1
The demonstrator enclosure featuring two BTnodes connected by wired adapter boards. The DSN node is situated on the left side, the target node with a mounted sensor board on the right side.

For demonstration purposes, a permanent setup (Figure 6-2) of 19 BTnodes was deployed on the floor of the Computer Engineering and Networks Laboratory at the Swiss Federal Institute of Technology in Zürich. All nodes are powered over USB to

ensure the availability of this long-term installation. Furthermore, the network is expandable with battery-operated pairs of nodes on demand.

The demonstrator not only exemplifies the concept of a deployment-support network and serves as a base for tests and measurements, it also provides an excellent platform for further research on wireless sensor networks and distributed algorithms.

The user interfaces consists of a stationary host computer and a Java applet available on `http://www.btnode.ethz.ch`.

Figure 6-2
Placement of 19 units in an indoor environment. The nodes are powered by USB either directly attached to personal computers or clustered around USB hubs.

6.2 Senso – A Sensor Network Application

For demonstrating purposes, we developed a small sensor application. It makes use of the possibility to attach a Teco sensor board to the DSN adapter board (described in Section 5.4). It has the following features:

- Sampling of analog microphone values. On the same time, the four LEDs are adjusted according to the measured sound intensity.
- Performing light measurements. The light sensor is connected to the TWI bus. The sensor values are converted to the actual illuminance measured in lux. Every two seconds a light measurement is printed to UART0.
- Integration of the BTnode target monitor (Section 4.5.2).
- Program size: 30 kB.

This application allows to demonstrate all features of the JAWS deployment support network such as target monitoring, logging and control.

7

Test Results and Benchmarks

7.1 Stability

A long-term stability test has been performed using the demonstrator described in Chapter 6. A special node – the host node – was used to issue commands to all other nodes in regular intervals of two minutes. All nodes were supposed to respond with a status packet. The reception of this status information indicated that the sending node is reachable and was recorded accordingly. Finally, the number of reachable nodes was counted and drawn in a graph.

Depending on the topology, a link failure may thus lead to different number of unreachable nodes. Since we used a tree topology originating at the host node, the breakdown of one node causes the whole subtree connected to that node to be unreachable.

A first experiment is shown in Figure 7-1. This test was conducted with 22 nodes and one host node and lasted almost three days. Apparently, there are some stability issues as nodes drop out of the network at random times and cannot reestablish their connection. Notable is the point in time of the first node failure, the network was stable for almost a full workday so that these random failure did not stand out during earlier development.

The analysis of the problem resulted in another issue with the Bluetooth controller. Very rarely, the BT module did not respond correctly to commands on the HCI interface. For example, the *Inquiry Command* was answered with *Inquiry Result* packets, but the final *Inquiry Complete Event* was missing. This behavior led to a deadlock of the BTnut system. Consequently, no further packets could be sent or received.

Since no insight into the Bluetooth module is possible, a software watchdog has been implemented. The watchdog has a timeout of ten minutes and is restarted on incoming data packets on the transport layer. Because the code distribution algorithm (Section 4.4) exchanges data packets in shorter intervals, an unintended system reset is avoided.

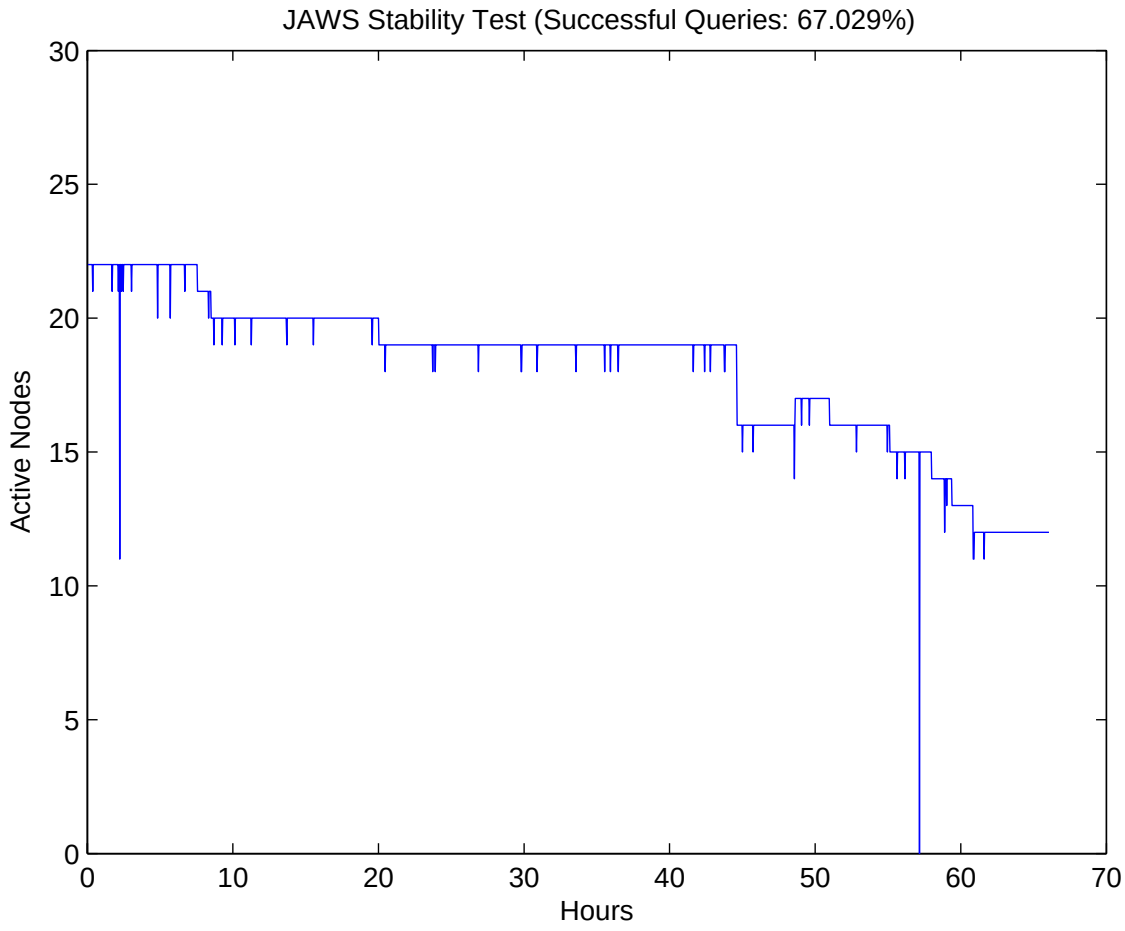


Figure 7-1
Failed stability test.

Another, clearer solution would be the redesign of the BTnut system to account for unexpected behavior on the HCI interface. Also, in order to deal with the deficient Bluetooth controller, the BTnut system should be able to restart the BT module separately which is feasible with the current BTnode hardware.

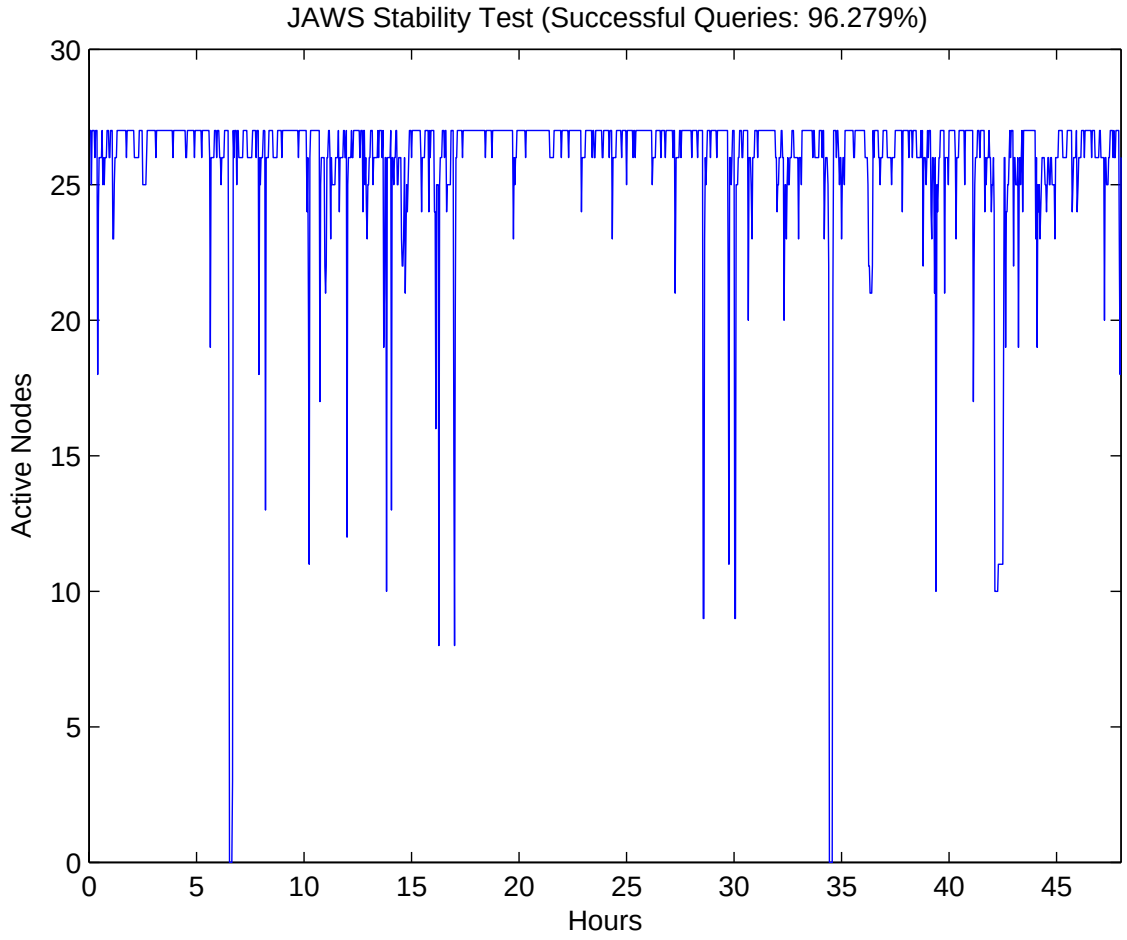


Figure 7-2
Stability test, first half.

Figure 7-2 and 7-3 show the result of a later stability test. Here, 27 nodes were used beside the host node and the test was run over four days. The above mentioned watchdog yielded a much stabler network where 96.279 % of all queries were answered. However, compared to Figure 7-1 this test contains a lot more spikes, that is more nodes were unreachable for only a short time. The reason is found in the test setup where the nodes were placed further apart. Due to the simple topology control – links were established independent of link quality – more weak links arose than in the first experiment.

Also, such link failures caused by weak connections are not distinguished from node failures which eventually lead to a system restart by the watchdog. In order to quantify the restarts, a counter was introduced (see command specification, Table B-3).

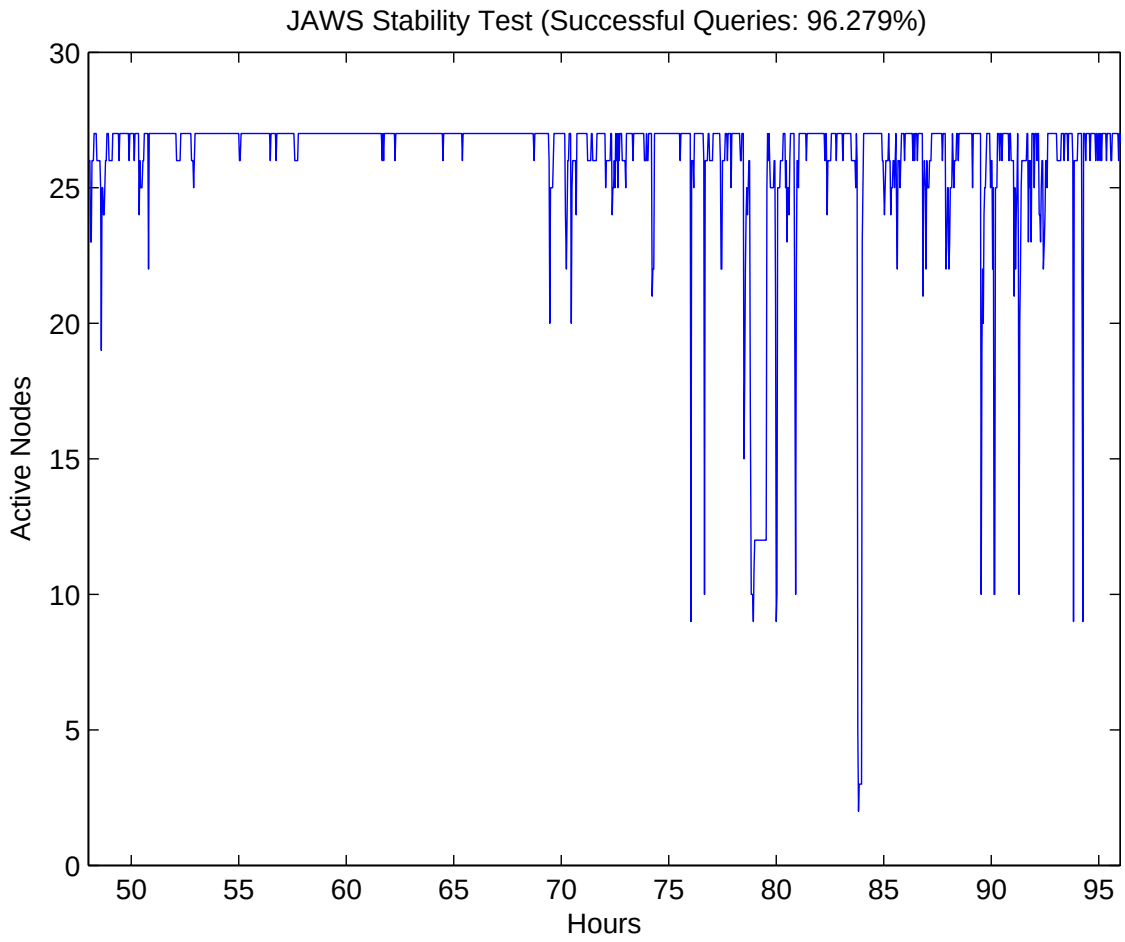


Figure 7-3
Stability test, second half.

7.2 Code Distribution

To perform tests and benchmarks of the code distribution the new logging facility (described in Section 4.6) has been used. A special log class has been defined to store code distribution log events.

In Figure 7-4, the result of an example test run is shown. The experiment has been conducted on the nodes of the demonstrator and on some more nodes placed in the same room around the host node. The network topology is depicted in Figure 7-5. A program of the size 120 kB has been loaded on the host node marked as square on the left. The average transmission time from node to node was 73 seconds with a minimum of 72 and a maximum of 94 seconds. After 14 minutes and 51 seconds, the program has been sent to the last node.

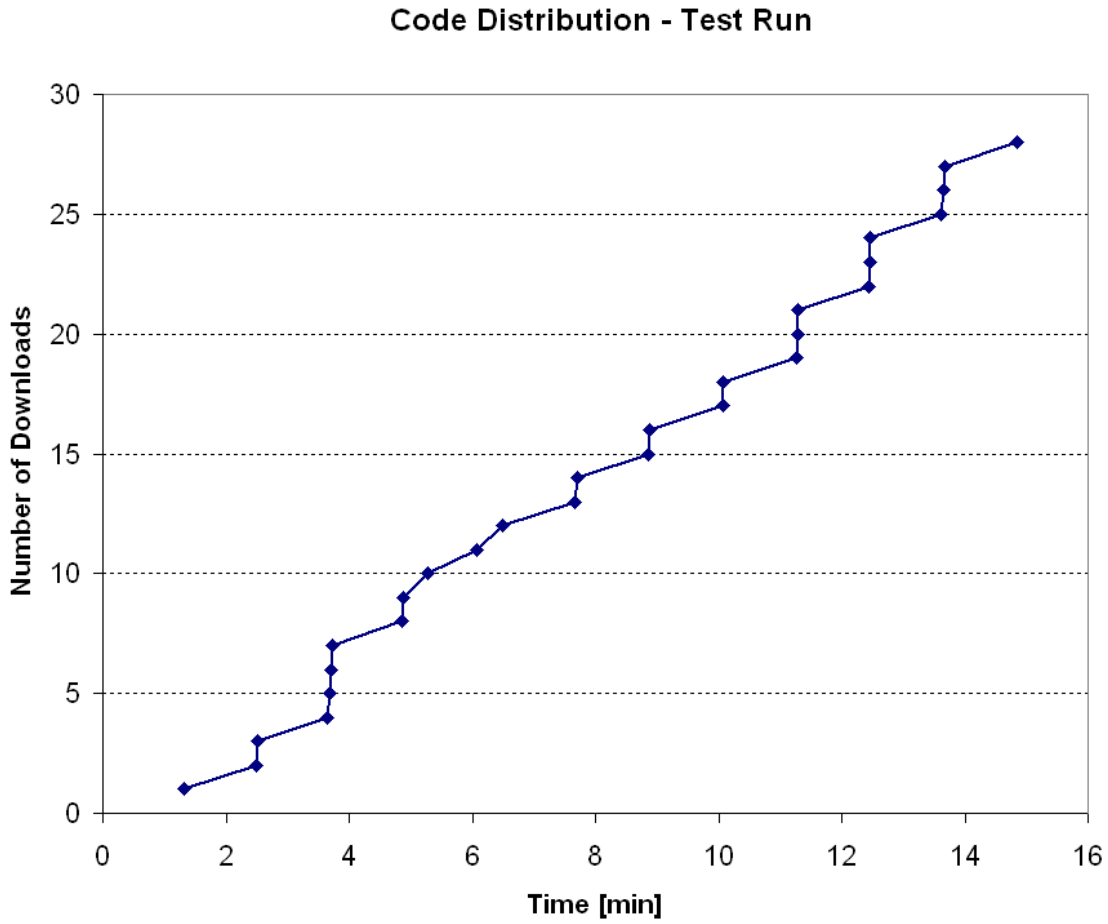


Figure 7-4
Code distribution test run with 29 nodes.

Clearly, the completion time depends on several factors:

- Transfer rate between two neighboring nodes.
- Network topology. The more possibilities exist to transfer programs simultaneously, the shorter the completion time will be.

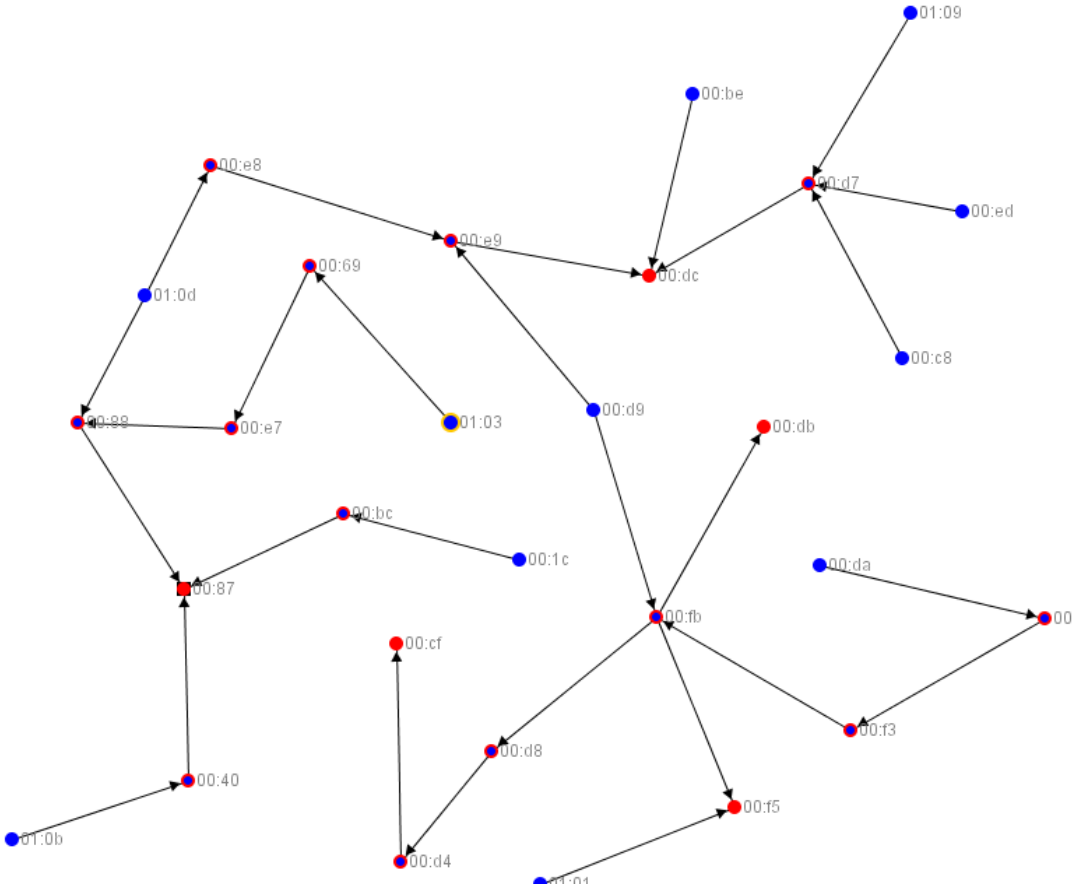


Figure 7-5
Network topology during the code distribution test run in Figure 7-4. The node marked as red square on the left is the host node from which the program is being distributed.

- Order in which the neighbors of an advertising node are processed. In the current implementation the order is random. Optimal with respect to the tree building connection manager would be that the neighbor with the largest subtree is served first.
- The probability of transmission failures.

7.2.1 Transfer Rate

Averaged over several experiments, the transfer rate from node to node has been determined as about 1.6 kB/s. Compared to the possible Bluetooth data rates this is slow. The reason for this is that the Bluetooth UART baud rate has been reduced to 19.2 kbit/s. Due to a bug in the firmware of the Bluetooth module the microcontroller cannot control the incoming packet rate. When the system is not fast enough to process the incoming data, packets have to be discarded resulting in packet loss. At higher baud rates this would lead to many failed program transfers. By reducing the speed between microcontroller and Bluetooth module to a level where the microcontroller is always fast enough to process the incoming data, the problem has been

mitigated. See Section 4.9 for the complete discussion.

Once this problem is sorted out, the code distribution performance will benefit considerably by increasing the Bluetooth UART baud rate.

7.2.2 Transmission Failures

Two different mechanisms assure correct program transfers:

- Lost packets are detected by comparing sequence numbers.
- In addition, the 16-bit program CRC is checked after all packets arrived.

During our tests[1] we observed transmission failures only in rare cases. The reason for an error was always a lost packet. An analysis of these cases has shown that the flow-control problem of the Bluetooth module is not the cause. We therefore assume that these missing packets are caused by transmission errors on the ACL link due to bad link quality and/or interferences.

It is expected that the use of topology control which incorporates link quality such as XTC [17] makes the probability of such errors even smaller.

7.3 Battery

A battery measurement was carried out to determine the runtime of battery powered BTnodes running the JAWS software. Three BTnodes on battery supply were connected with direct links to the host node. The BTnodes were powered by Ansmann AA Ni-MH 2400 mAh rechargeable batteries and measured the battery voltage using the built-in analog-digital converter.

The logging facility was used to transmit the self-measured battery voltage from each node to the host node every 60 seconds. This resulted in a steady load of the network due to the regular transmission of data packets.

Figure 7-6 illustrates the decreasing voltage measurements of each node. While the voltage level fairly differ, the runtime of all nodes was around 27 hours. A good indicator for the necessity to replace the batteries soon is 2.4 V, an urgent replacement is needed at 2.3 V.

[1] Bluetooth UART baud rate was already reduced to 19.2 kbit/s as mentioned in Section 4.9.2.

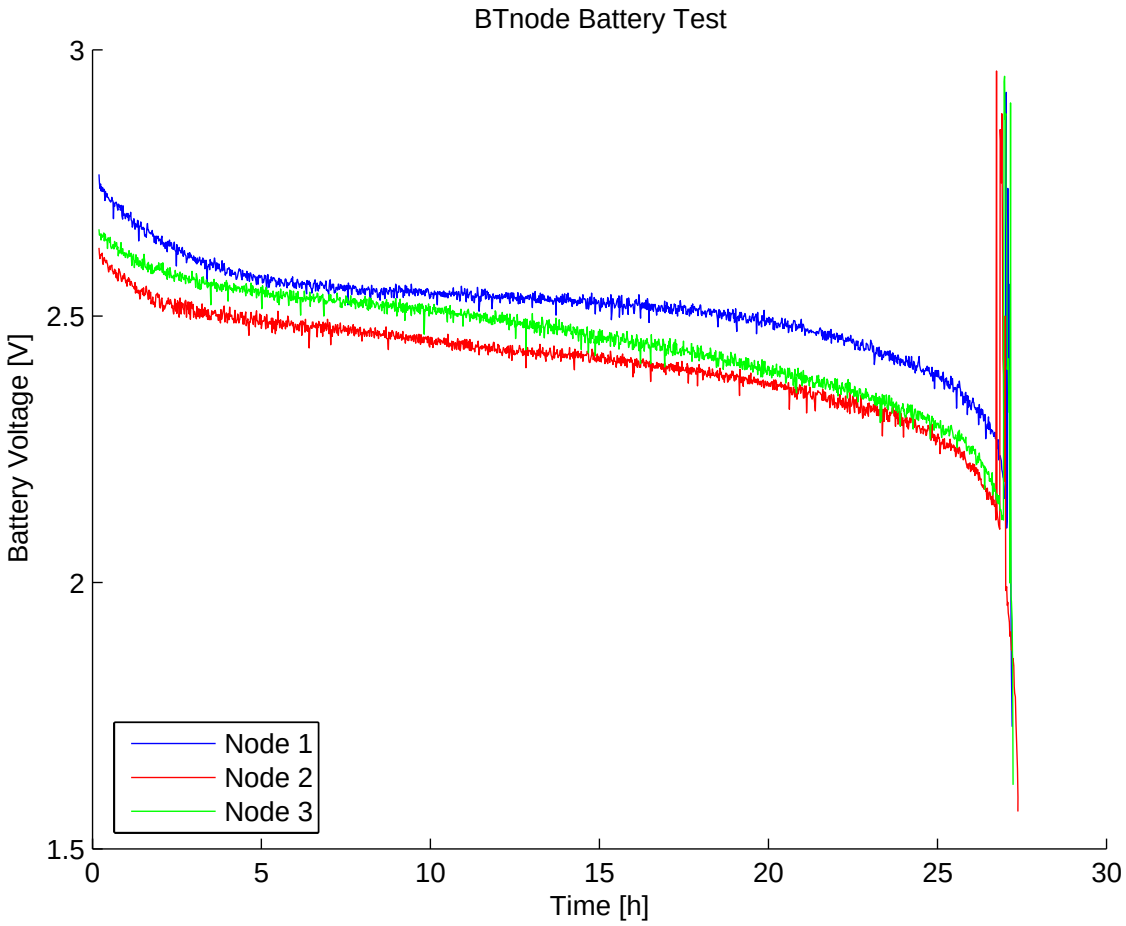


Figure 7-6
Battery Test with 3 Nodes.

8

Conclusion

The last chapter summarizes the achievements of this thesis and proposes some topics for future workings.

8.1 Summary

The concept of a deployment-support network (DSN) is a novel tool for wireless development and real-world deployment of large-scale sensor networks. JAWS, an exemplary implementation on the BTnode platform, forms a self-organizing network based on Bluetooth scatternets and provides multi-hop forwarding.

In this thesis, several services to extend the JAWS software were evaluated, specified and implemented. They include mechanisms to program, control and monitor the attached sensor nodes. Several modifications to the graphical user interface were made in order to support the newly developed services. A hardware solution, the DSN adapter board, was developed to connect target sensor nodes to their DSN counterparts. It has been designed as flexible as possible with the option to not only attach it to the DSN node, but also to attach the same board to targets which use the BTnode as platform. An enclosure holding a DSN node and the corresponding target was worked out and used in a permanent installation of 19 DSN nodes for demonstration purposes.

The demonstrator served as a base for various stability and performance measurements. A lot of effort went into the overall stability of the wireless network. With the demonstrator, it was possible to show that JAWS is mature for durable deployment. Running unattended, without further interaction needed, it is thus ready for long-term applications.

Furthermore, it was shown that the concept of deployment-support networks is feasible and may ease the development of wireless sensor applications considerably.

8.2 Future Work

In order to make the JAWS software usable for less experienced users, the graphical user interface needs to be reworked. Currently, the user still has to know much about the actual terminal commands. Altogether, the handling of large numbers of nodes should be possible in a more user-friendly way.

While the most important development services have been implemented in this thesis, it would be possible to further enhance the debugging capabilities of the DSN. For instance, it could be feasible in the future to include a JTAG controller in the target adapter software. This would allow to provide debugging services that approach the comfort of graphical debuggers common in software development. Also, the monitor fragment running on the target node may need some adaptations depending on the respective sensor network application.

Furthermore, additional target platforms may be supported in the future by developing an appropriate adapter board or cable. While support for Atmel AVR based platforms is simply a matter of correct wiring, other microcontroller technologies may require more complex modifications.

Using the demonstrator that is introduced in this thesis, a thorough comparison of algorithms is feasible. For example, the recently implemented XTC topology control algorithm may be compared to the currently used treenet algorithm. Interesting criteria are the quality of Bluetooth links as well as the availability and stability of the formed network.

Other topics for future theses lie in research fields that investigate time synchronization and location services. Here, the JAWS system may be used for measuring the accuracy and reliability of algorithms.

A

Protocol

A.1 Connection Manager Packets

Field	Bytes	Explanation
ptype	1	Set to TYPE_TREE_ID
id	6	Tree-ID (Bluetooth device address)

Table A-1: Tree-ID packet.

Field	Bytes	Explanation
ptype	1	Set to TYPE_NEGOT
id	6	Tree-ID
role	1	Master or slave in Bluetooth piconet.

Table A-2: Negotiation packet.

Field	Bytes	Explanation
ptype	1	Set to TYPE_NEGOT_RESP
id	6	Tree-ID

Table A-3: Negotiation packet response.

A.2 Transport Manager Packets

Field	Bytes	Explanation
ptype	1	Set to CMD_REQUEST_PKT or CL_DATA_PKT
seq	1	Broadcast sequence number
source	6	Source address
dest	6	Destination address
cmd	1	Command code
verbose/ttl	1	Bit 0-6: time to live, bit 7: verbose flag
hop_cnt	1	Hop count
data	*	Payload

Table A-4: Connectionless data packet. This packet type is used to broadcast transport manager commands and connectionless data to DSN nodes. Additionally, the same format is used to send connectionless data back to a host node.

Field	Bytes	Explanation
ptype	1	Set to VC_DATA_PKT
vci	2	Virtual-circuit identifier
data	*	Payload

Table A-5: Virtual connection data packet.

Field	Bytes	Explanation
ptype	1	Set to NB_DATA_PKT
data	*	Payload

Table A-6: Neighbor data packet. Used to communicate with direct neighbors in network topology. Therefore, these packets are not routed.

A.3 DSN Packets

A.3.1 NB Data

Field	Bytes	Explanation
ptype	1	Set to CDIST_INFO
prog type	1	Program type: 0=dsn, 1=target
size	4	Size all program data records in bootloader format.
boot addr	4	Boot address. Needed to define, whether bootloader is used or not.
version	4	Encoded upload time.
name	*	Null terminated string.

Table A-7: cdist prog info packet.

Field	Bytes	Explanation
ptype	1	Set to CDIST_DATA
seq nr	2	Sequence number starting at zero.
data	*	Program data.

Table A-8: cdist prog data packet.

Field	Bytes	Explanation
ptype	1	Set to CDIST_ACK
seq nr	2	

Table A-9: cdist ack reply packet.

Field	Bytes	Explanation
ptype	1	Set to CDIST_NACK

Table A-10: cdist nack reply packet.

A.3.2 CL Data

Field	Bytes	Explanation
ptype	1	set to CMDEXEC
trans id	1	Transaction ID
cmd	*	Null termianted command string.

Table A-11: dsn cmd exec packet.

Field	Bytes	Explanation
ptype	1	Set to CMDSTATUS
trans_id	1	Transaction ID.
error_code	1	0 = ok, 1 = unknown, 2 = busy, 3 = cmd not found
output_len	2	Length of the command output.

Table A-12: dsn cmd status packet.

Field	Bytes	Explanation
ptype	1	Set to CMDRESULT
trans_id	1	Transaction ID.
seq_nr	1	Sequence number starting at zero for this command result packet.
data	*	Null terminated substring of the command output.

Table A-13: dsn cmd result packet.

Field	Bytes	Explanation
len	2	Total length of the log record.
class	1	Log class
level	1	Log level
time	4	Time stamp in milliseconds
data	len-8	Text message

Table A-14: log record

Field	Bytes	Explanation
ptype	1	Set to LOG_DATA
log records	*	One or more log records.

Table A-15: log data packet.

B

Command Specification

This chapter provides an overview on the available terminal commands. Important, changed or new commands are described in separate tables. Command output beginning with a colon is intended for being parsed by the JAWS graphical user interface.

B.1 Connection Manager Commands

```
cm id
cm relcons
cm inq
cm autinq [ 0|1 [0|1] ]
cm aiperiod [ SECONDS [0|1] ]
cm aitime [ SECONDS [0|1] ]
cm maxsl [1-7]
cm blink
cm disconall
```

B.2 Transport Manager Commands

```
tp openvc xx:xx:xx:xx:xx:xx
tp send <vc-handle> <len>
tp psend <vc-handle> <len> <#pkts> <pause 100ms>
tp clsnd xx:xx:xx:xx:xx:xx <len>
tp close <vc-handle>
tp vt (vc table)
tp ht (host table)
tp rc (reliable con)
tp bc
tp hd (host delete)
tp trace xx:xx:xx:xx:xx:xx
tp con xx:xx:xx:xx:xx:xx
```

```
tp rname xx:xx:xx:xx:xx:xx
tp cmd <ttl> xx:xx:xx:xx:xx:xx
tp verbose [1|0]
tp wd [reset]
```

Command	`tp con <addr>`

Output (async)	`:T <source-addr> <num-entries>`
	`:TE <1st neighbor addr> <con-state> <NRSSI>`
	`:TE <2nd neighbor addr> <con-state> <NRSSI>`
	`...`

Table B-1: Topology Request

con-state 0 = unconnected, 1 = neighbor is my master, 2 = neighbor is my slave, 3 = denied connection

NRSSI Negative RSSI in dB. Range: [0..100] (0=perfect, 100=bad)

Command	`tp trace <addr>`

Output (async)	`:TR 1 <1st hop-addr>`
	`:TR 2 <2nd hop-addr>`
	`...`
	`:TR N <addr>`

Table B-2: Trace: The target node sends a reply packet back to the host. On the way to the host each intermediate node appends its address to the packet.

Command	`tp wd [reset]`

Table B-3: Transport watchdog: After 10 minutes without incoming packet the node is reset. This command prints or resets the watchdog statistics, i.e. how many times the watchdog has been triggered.

B.3 DSN Commands

```
dsn testpkt <vc-handle> <nr>
dsn time <vc-handle> <num> <len> <sleep>
dsn cmd <trans-id> [<addr>] [<cmd>]
dsn cdist on|off|run
dsn sendlog <addr> [<class> [<mask>]]
dsn logfilter <class> [<mask>]
```

Command	`dsn cdist run`

Table B-4: Start code distribution immediately on local node.

Command	`dsn sendlog <host-addr> <log-class> <log-level>`
Output (on specified host node)	`:DL <source-addr> <log-class> <log-level> <length> <log-data>`

Table B-5: Send stored log entries to a specified host address.

Command	`dsn logfilter <host-addr> <log-class> <log-level>`
Output (on specified host node)	`:DL <source-addr> <log-class> <log-level> <length> <log-data>`

Table B-6: Permanently send new log entries to a host address according to log class and log levels specified. Deactivated by choosing log level zero. Omitting log class/level returns current setup.

Command	`dsn cmd <trans-id> [<host-addr>] [<remote-cmd>]`
Input	`<remote-cmd>`
Output (async)	`:C <source-addr> <trans-id> failed: <reason>` `:C <source-addr> <trans-id> completed: <total-len>`
Output (async)	`:CO <source-addr> <trans-id> <seq-nr> <len> <remote-cmd-output>`

Table B-7: Remote command execution.

trans-id Unique id used by GUI/user to map cmd result pkts to issued cmds.

seq-nr Sequence number always starting at zero.

total-len Length of the complete `<remote-cmd-output>` in bytes.

len Length of the following `<remote-cmd-output>` part in bytes.

B.4 Target Adapter Commands

```
tg cmd <cmd>
tg flash
tg get bat
tg get fuses
tg get status
tg get version
tg reset
tg set fuses
tg set power on|off
```

Command	`tg flash`
Output 1	`:TF ok`
Output 2	`:TF failed: <reason>`

Table B-8: Reprogram the target.

Command	`tg get fuses`
Output 1	`:TGF ok: 0xEEHHLL`
Output 2	`:TGF failed: <reason>`

Table B-9: Reads extended, high and low fuse bytes.

Command	`tg set fuses`
Output 1	`:TSF ok: 0xEEHHLL`
Output 2	`:TSF failed: <reason>`

Table B-10: Sets the standard fuses: `0xFF408E`.

Command	`tg get bat`
Output	`:TB <voltage> V`

Table B-11: Print target battery voltage.

Command	`tg get version`
Output 1	`:TV YYYYMMDD-hhmm`
Output 2	`:TV unknown`

Table B-12: Print target version. If no target monitor support is compiled into the target application, the string `unknown` is printed..

Command	`tg cmd <cmd>`

Table B-13: Execute arbitrary terminal commands on the target.

Command	`tg reset`

Table B-14: Reset the target.

Command	`tg set power on | off`

Table B-15: Control target power. If target runs on batteries, this command should not be used.

B.5 Monitor Commands

```
mon bat
mon cmd <cmd>
mon heap
mon irq [<nr>]
mon net
mon reg <reg/port/mem-addr>
mon timers
mon threads
```

B.6 Logging Commands

```
log show [<class> [<mask]]
log clear
log logmask <class> [<mask>]
log verbosity <class> [<mask>]
```

Command	`log show <log-class> <log-level>`
Output	`<log buffer entries>`

Table B-16: Output log buffer to terminal.

B.7 Other Commands

```
loadhex <ver> <type: 0=dsn, 1=tg> [<name>]
blink
get bat
reset
xbank get proginfo
xbank get status
xbank set progname <name>
xbank set progtype dsn|tg
xbank set progver <ver>
```

Command	`loadhex <version> [<type> <name>]`
Output	`ready to receive hex data, press enter for quit`
Input	program hex data
Output 1	`:LH completed: <nl> lines read`
Output 2	`:LH failed: <reason>`

Table B-17: Load program code stored in Intel HEX file to local storage memory.

version Time encoded as 32-bit integer.

type Program type. 0 = dsn, 1 = target

name Program name.

Command	`get bat`
Output 1	`:B <voltage> V`
Output 2	`:B external`

Table B-18: Print battery voltage.

voltage Format example: `2.56 V`

Command	`xbank get proginfo`
Output 1	`:XNP`

Output 2	`:XPT <progtype>`
	`:XPN <progname>`
	`:XPV <version>`
	`:XPS <progsize>`
	`:XPB <boot-addr>`

Table B-19: Print information about program in storage memory.

B.8 JAWS Commands

```
jaws get version
jaws get name
jaws set mode dsn|gui
jaws set name <name>
```

Command	`jaws get version`
Output	`:JV YYMMDD-hhmm`

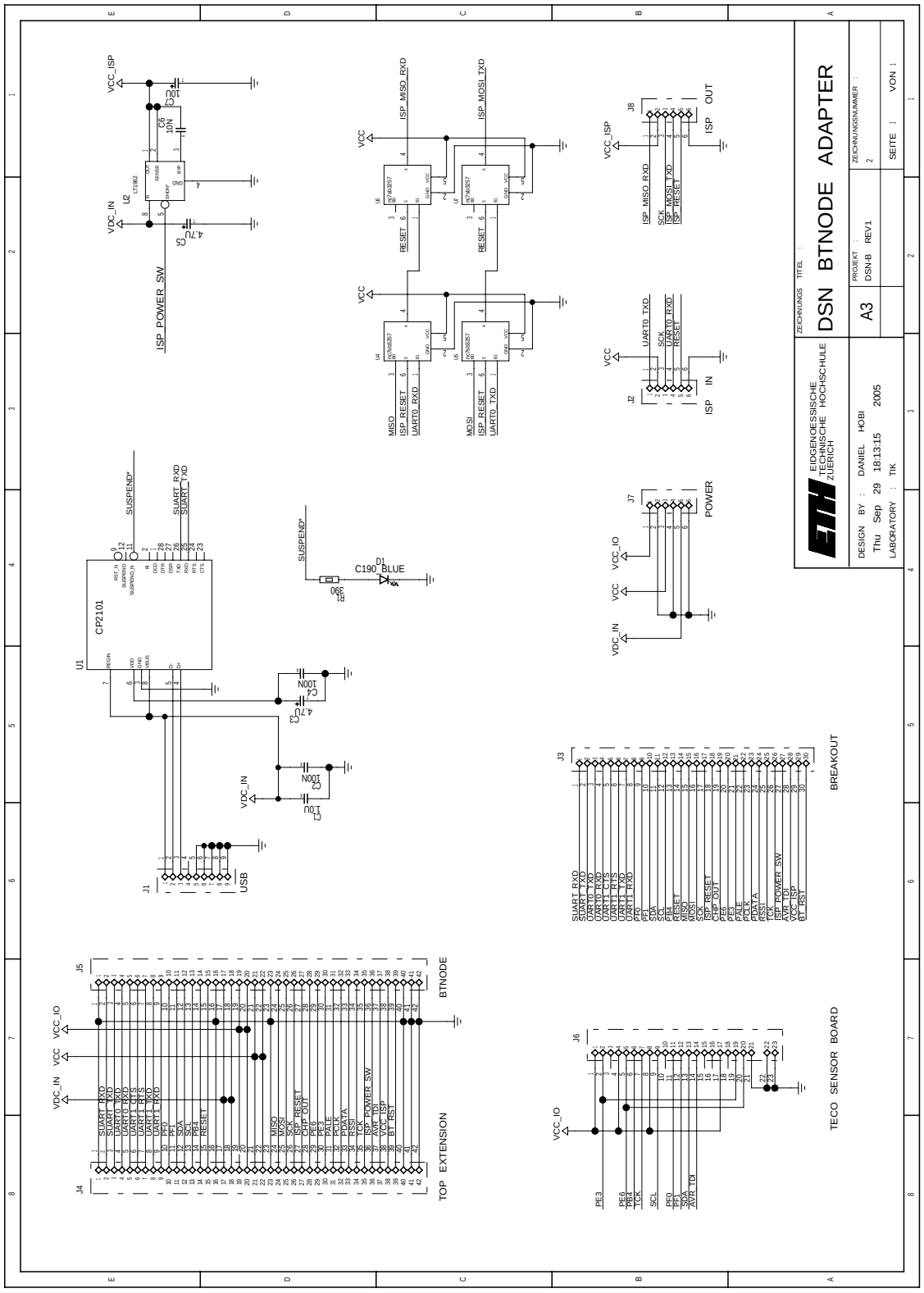
Table B-20: Print JAWS program version.

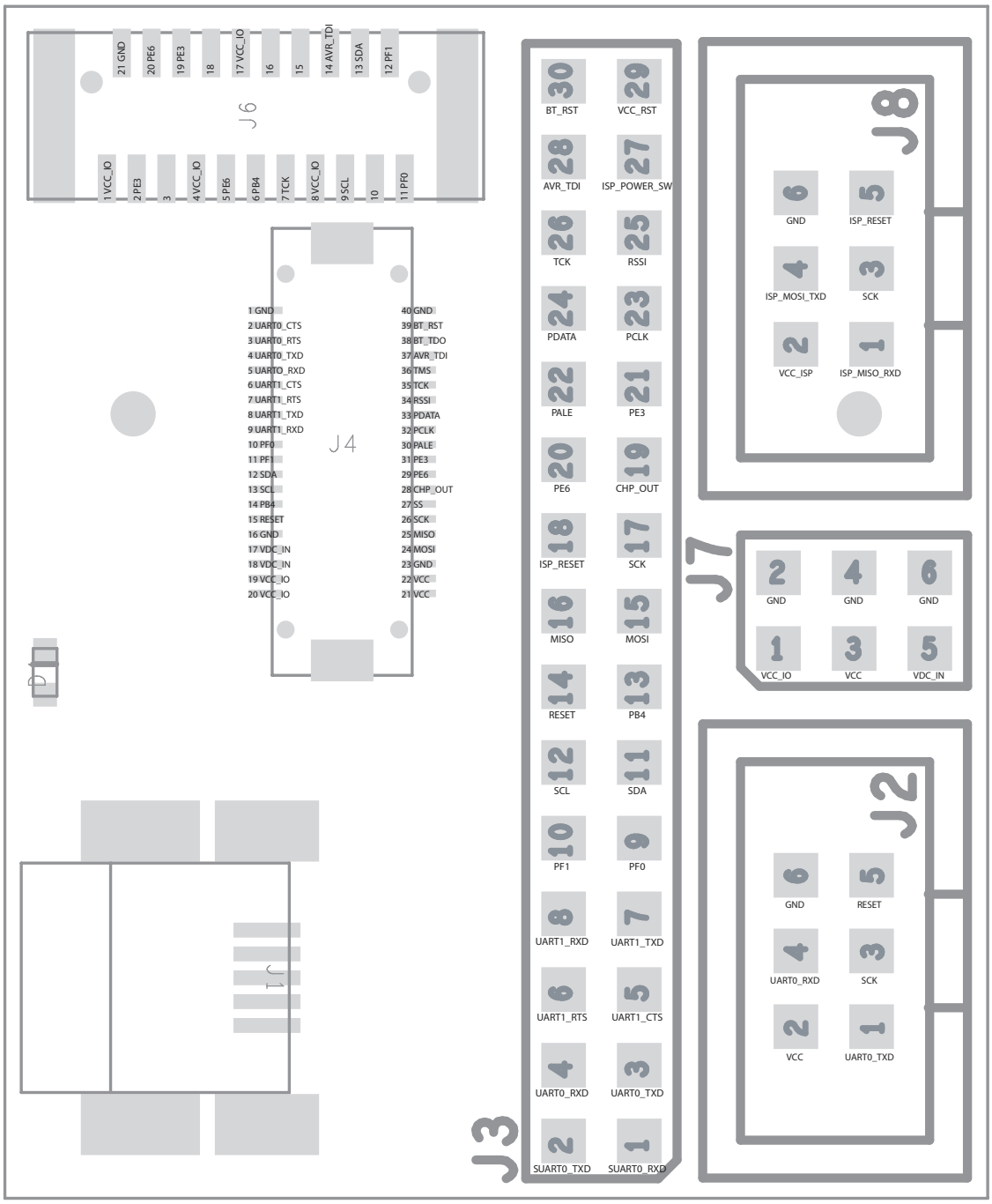
Command	`jaws set mode dsn|gui`

Table B-21: Set JAWS mode. Use this command to configure the local node as normal DSN node or as GUI node. Defines, whether the DSN adapter board or the USBprog adapter board can be attached.

C

DSN Adapter Schematic & Pin-Out


DSN BTNODE ADAPTER
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE ZUERICH
DESIGN BY : DANIEL HÖBI
Thu Sep 29 18:13:15 2005
LABORATORY : TIK
A3
PROJEKT : DSN-B REV1
SEITE 1 VON 1
TECO SENSOR BOARD
BREAKOUT
TOP EXTENSION
BTNODE
USB
POWER
ISP OUT
ISP IN
VCC_ISP
VDC_IN
VCC_IO
VCC
ISP_POWER_SW
SUSPEND*
SUART_RXD
SUART_TXD
ISP_MISO_RXD
ISP_MOSI_TXD
UART0_TXD
UART0_RXD
ISP_RESET
SCK
MISO
MOSI
CP2101
C190_BLUE


J6
21 GND
20 PE6
19 PE3
18
17 VCC_IO
16
15
14 AVR_TDI
13 SDA
12 PF1
1 VCC_IO
2 PE3
3
4 VCC_IO
5 PE6
6 PB4
7 TCK
8 VCC_IO
9 SCL
10
11 PPD
J4
1 GND
2 UART0_CTS
3 UART0_RTS
4 UART0_TXD
5 UART0_RXD
6 UART1_CTS
7 UART1_RTS
8 UART1_TXD
9 UART1_RXD
10 PF0
11 PF1
12 SDA
13 SCL
14 PB4
15 RESET
16 GND
17 VDC_IN
18 VDC_IN
19 VCC_IO
20 VCC_IO
40 GND
39 BT_RST
38 BT_TDO
37 AVR_TDI
36 TMS
35 TCK
34 RSSI
33 PDATA
32 PCLK
30 PALE
31 PE3
29 PE6
28 CHP_OUT
27 SS
26 SCK
25 MISO
24 MOSI
23 GND
22 VCC
21 VCC
D1
J1
J3
30 BT_RST
29 VCC_RST
28 AVR_TDI
27 ISP_POWER_SW
26 TCK
25 RSSI
24 PDATA
23 PCLK
22 PALE
21 PE3
20 PE6
19 CHP_OUT
18 ISP_RESET
17 SCK
16 MISO
15 MOSI
14 RESET
13 PB4
12 SCL
11 SDA
10 PF1
9 PF0
8 UART1_RXD
7 UART1_TXD
6 UART1_RTS
5 UART1_CTS
4 UART0_RXD
3 UART0_TXD
2 SUART0_TXD
1 SUART0_RXD
J8
6 GND
5 ISP_RESET
4 ISP_MOSI_TXD
3 SCK
2 VCC_ISP
1 ISP_MISO_RXD
J7
2 GND
4 GND
6 GND
1 VCC_IO
3 VCC
5 VDC_IN
J2
6 GND
5 RESET
4 UART0_RXD
3 SCK
2 VCC
1 UART0_TXD

D

Assignment

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik (TIK)

Sommersemester 2005

MASTERRARBEIT

für
Daniel Hobi und Lukas Winterhalter

Betreuer: Jan Beutel
Stellvertreter: Matthias Dyer

Ausgabe: 1. April 2005
Abgabe: 1. Oktober 2005

Large-scale Bluetooth Sensor-Network Demonstrator

Einleitung

Eine bekannte Vision für ad hoc Netzwerke [16] geht davon aus, das unendlich viele, kleinste "Sensorknoten" kollaborativ ein Netzwerk und somit eine Applikation bilden. In anderen Visionen [13, 6] wird davon ausgegangen, das solche System weite Anwendungsbereiche abdecken können und das die einzelnen Komponenten unterschiedliche Ressourcen aufweisen.

Die BTnodes [5] bestehen aus einem Atmel AVR Mikrokontrollerm einem Bluetooth Modul und ein Low-Power Radio. Zusammen mit der im NCCR-MICS [30] entwickelten BTnut System Software bilden sie eine sehr kompakte programmierbare Platform für die Entwicklung mobiler ad hoc und Sensornetze. An diese Knoten können diverse Peripheriegeräte (z.B. Sensoren) angehängt werden. Mit der geeigneten Software bauen viele Sensorknoten selbstständig ein Sensornetzwerk auf, worüber die Sensordaten transportiert werden können.

Die Mica Motes und ihr Betriebssystem TinyOS sind ein ähnliches System das an der UC Berkeley entwickelt [12, 19, 18] und von Crossbow [29] kommerzialisiert wurde, das aber mit einem proprietärem Funkprotokoll auf Basis eines Chipcon CC1000 [8] Radios arbeitet. TinyOS ist heute der de-facto Industriestandard für Sensorplatformen.

Heute werden Applikationen für Sensornetze meist explorativ entwickelt. Hierzu ist relativ viel Aufwand von Personal, Know-How und entsprechenden iterativen Designzyklen notwendig. Erfahrungsberichte von solchen Experimenten gibt es von Szewcyk [23, 24, 22], Cerpa [7], Hemingway [11], Mainwaring [20] und anderen. Erste Ansätze die koordinierte Methoden und Verfahren eines ganzheitlichen Entwicklungsprozesses zum Ziel haben gibt es bereits. Insbesondere sind in den Teilbereichen der Simulation [18, 21, 17], Emulation [10], Entwicklung [9, 5], Inbetriebnahme [15], Test [26], Validierung und Verifikation [4] Lösungen vorhanden. Einer besonderer Ansatz stellt hier das sogennante Deployment-Support Netzwerk (DSN) [3, 2, 4] dar, welches als temporäres Werkzeug während des Entwicklungs- und Inbetriebnahmenprozesses sowie zur Überwachung angewendet werden kann.

Das Ziel dieser Arbeit ist, einen dauerhaften DSN Demonstrator zu entwickeln der im Bereich des ETZ Gebäudes installiert wird. Dazu werden Anpassungen der vorhandenen BTnodes und der Stromversorgungen notwendig sein sowie die Entwicklung von Funktionen und Tools zur Steuerung und Programmierung des Netzwerkes.

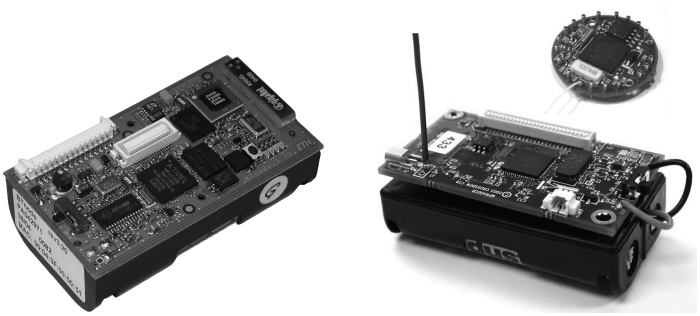


Abbildung 1: Der BTnode rev3 und die Mica2 Mote Familie.

Konkret soll eine fixe und zuverlässige Installation des JAWS deployment-support networks am TIK realisiert werden (siehe Abb. 2). Diese Anwendung wurde am TIK u.a. mit Unterstützung von Semester-/Diplomarbeiten entwickelt. Um aber diese Anwendung in einen permanenten stabilen Zustand für die Demonstration zu bringen, muss diese Software noch verbessert werden.

Aufgabenstellung

1. Erstellen Sie einen Projektplan und legen Sie Meilensteine sowohl zeitlich wie auch thematisch fest [27]. Erarbeiten Sie in Absprache mit dem Betreuer ein Pflichtenheft.
2. Machen Sie sich mit den relevanten Arbeiten im Bereich Sensornetze, Platformen, Systeme, Software und Fast-prototyping vertraut. Führen Sie eine Literaturrecherche durch. Suchen Sie auch nach relevanten neueren Publikationen. Vergleichen Sie bestehende Demonstratoren anderer Universitäten (Motelab [25], Smote [1], Kansai, Mirage, EmStar [10]). Prüfen Sie welche Ideen/Konzepte Sie aus diesen Lösungen verwenden können.
3. Arbeiten Sie sich in die Softwareentwicklungsumgebung der BTnodes [28] ein. Machen Sie sich mit den erforderlichen Tools vertraut und benutzen Sie die entsprechenden Hilfsmittel (Versionskontrolle, Bugtracker, online Dokumentation, Mailinglisten, Application Notes, Beispielapplikationen).
4. Nehmen Sie das JAWS Deployment-Support Network auf einigen Knoten in Betrieb und testen Sie dieses auf Zuverlässigkeit und Leistung. Erstellen Sie eine Liste der noch fehlenden Eigenschaften die für einen zuverlässigen Dauerbetrieb notwendig sind sowie der noch vorhandenen Fehlfunktionen. Benutzen Sie hierzu soweit Möglich computergestütze tools (Bugtracker).
5. Das DSN besitzt noch keine Target Anbindung, diese wurde jedoch in vorhergegangenen Arbeiten [14] schon separat entwickelt. Machen Sie sich mit diesen Funktionen vertraut und entwickeln Sie ein Konzept zur Anbindung verschiedener Targets (BTnode, Mica Motes, etc.) Schlagen sie eine physikalische Verbindung vor (Kabel oder PCB) und implementieren Sie diese. Überprüfen Sie ob es möglich ist auch einen standard Sensor in diese Hardware zu integrieren um Referenzmessungen innerhalb des DSNs zu ermöglichen.
6. Implementieren Sie die erforderlichen DSN services wie *remote command execution*, *code distribution*, *target programming*, *self-programming*, etc. entsprechend der von Ihnen festgestellten Anforderungen.
7. Erarbeiten Sie einen Vorschlag für die Befestigung sowie eine verteilte Stromversorgung der DSN Knoten. Zur Fertigung von mechanischen Baugrupen kann z.B. die Elektrotechnikwerkstatt hinzugezogen werden. Installieren Sie mit diesem Konzept ein 40 Knoten Netzwerk am TIK (ETZ G Geschoss).
8. Führen sie Tests und Benchmarks durch. Interpretieren Sie die Resultate.
9. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

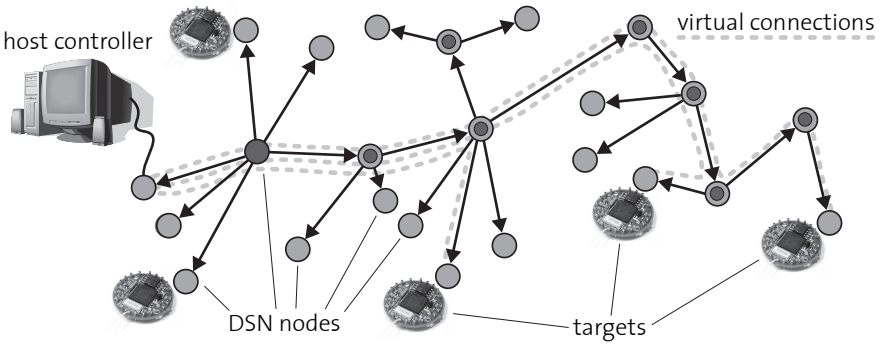


Abbildung 2: JAWS: Deployment-Support Network. Die BTnodes (DSN nodes) bilden und unterhalten selbstständig ein Backbone-Netzwerk über welches die Targetnodes (z.B. die Berekley Motes) programmiert bzw. überwacht werden können.

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 *1. Oktober 2005* dem betreuenden Assistenten oder seinen Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.
- Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Source- und Objectfiles, Konfigurationsfiles, benötigten Directorystrukturen usw. bestehen bleiben. Der Programmcode sowie die Filestruktur soll ausreichen dokumentiert sein. Eine spätere Anschlussarbeit soll auf dem hinterlassenen Stand aufbauen können.

Literatur

[1] UC Berkeley. Smote: Berkeley network sensor testbed. http://smote.cs.berkeley.edu/.

[2] J. Beutel. *Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems*, chapter Location Management in Wireless Sensor Networks. CRC-Press, Boca Raton, FL, 2004.

[3] J. Beutel, M. Dyer, M. Hinz, L. Meier, and M. Ringwald. Next-generation prototyping of sensor networks. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 291–292. ACM Press, New York, November 2004.

[4] J. Beutel, M. Dyer, L. Meier, and L. Thiele. Scalable topology control for deployment-sensor networks. In *Proc. 4th Int'l Conf. Information Processing in Sensor Networks (IPSN '05)*, page to appear, April 2005.

[5] J. Beutel, O. Kasten, F. Mattern, K. Römer, F. Siegemund, and L. Thiele. Prototyping wireless sensor network applications with BTnodes. In *Proc. 1st European Workshop on Sensor Networks (EWSN 2004)*, volume 2920 of *Lecture Notes in Computer Science*, pages 323–338. Springer, Berlin, January 2004.

[6] L. Blazevic, L. Buttyan, Capkun S., S. Giordano, J.P. Hubaux, and J.Y. Le Boudec. Self organization in mobile ad hoc networks: the approach of Terminodes. *IEEE Communications Magazine*, 39(6):166–174, June 2001.

[7] A. Cerpa, J.E. Elson, M. Hamilton, J. Zhao, D. Estrin, and L. Girod. Habitat monitoring: application driver for wireless communications technology. *ACM SIGCOMM Computer Communication Review*, 31(2):20–41, April 2001.

[8] Chipcon. *CC1000, Single Chip Very Low Power RF Transceiver*, April 2002.

[9] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. ACM SIGPLAN 2003 Conf. Programming Language Design and Implementation (PLDI 2003)*, pages 1–11. ACM Press, New York, June 2003.

[10] L. Girod, J. Elson, A. Cerpa, T. Stathapopoulos, N. Ramananthan, and D. Estrin. EmStar: A software environment for developing and deploying wireless sensor networks. In *Proc. USENIX 2004 Annual Tech. Conf.*, pages 283–296, June 2004.

[11] B. Hemingway, W. Brunette, T. Anderl, and G. Borriello. The Flock: Mote sensors sing in undergraduate curriculum. *IEEE Computer*, 37(8):72–78, August 2004.

[12] J.L. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. 9th Int'l Conf. Architectural Support Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104. ACM Press, New York, November 2000.

[13] J.P. Hubaux, T. Gross, J.Y. Le Boudec, and M. Vetterli. Toward self-organized mobile ad hoc networks: The Terminodes Project. *IEEE Communications Magazine*, 39(1):118–124, January 2001.

[14] T. Hug and F. Süss. Mote/TinyOS meets BTnode, February 2004.

[15] J.W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 81–94. ACM Press, New York, November 2004.

[16] J.M. Kahn, R.H. Katz, and K.S.J. Pister. Next Century Challenges: Mobile Networking for Smart Dust. In *Proc. 5th ACM/IEEE Ann. Int'l Conf. Mobile Computing and Networking (MobiCom '99)*, pages 271–278. ACM Press, New York, August 1999.

[17] O. Landsiedel, K. Wehrle, and S. Götz. Accurate prediction of power consumption in sensor networks. In *Proc. 2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, page to appear. IEEE, Piscataway, NJ, May 2005.

[18] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003)*, pages 126–137. ACM Press, New York, November 2003.

[19] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, Brewer E., and D. Culler. The emergence of networking abstractions and techniques in TinyOS. In *Proc. First Symp. Networked Systems Design and Implementation (NSDI '04)*, pages 1–14. ACM Press, New York, March 2004.

[20] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proc. 1st ACM Int'l Workshop Wireless Sensor Networks and Applications (WSNA 2002)*, pages 88–97. ACM Press, New York, September 2002.

[21] V. Shnayder, M. Hempstead, B. Chen, G. Werner-Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 188–200. ACM Press, New York, November 2004.

[22] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 214–226. ACM Press, New York, November 2004.

[23] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin. Habitat monitoring with sensor networks. *Communications of the ACM*, 47(6):34–40, June 2004.

[24] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proc. 1st European Workshop on Sensor Networks (EWSN 2004)*, volume 2920 of *Lecture Notes in Computer Science*, pages 307–322. Springer, Berlin, January 2004.

[25] Harvard University. MoteLab: Harvard network sensor testbed. http://motelab.eecs.harvard.edu/.

[26] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A wireless sensor network testbed. In *Proc. 4th Int'l Conf. Information Processing in Sensor Networks (IPSN '05)*, page to appear, April 2005.

[27] E. Zitzler. Studien- und Diplomarbeiten, Merkblatt für Studenten und Betreuer. Computer Engineering and Networks Lab, ETH Zürich, Switzerland, March 1998.

[28] BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. http://www.btnode.ethz.ch.

[29] Crossbow Technology Inc. http://www.xbow.com.

[30] NCCR-MICS: Swiss National Competence Center on Mobile Information and Communication Systems. http://www.mics.org.

Date	Section	Changes
Jan. 06, 2005		Initial version
Sep. 28, 2005		Minor revisions

Tabelle 1: Revision History

Bibliography

[1] Argo: Global ocean surveillance. http://www.argo.ucsd.edu.

[2] Ethernut: An open source hardware and software project for building tiny embedded ethernet devices. http://www.ethernut.de.

[3] Particle computer – sensor networks and solutions. http://www.particle-computer.net.

[4] Atmel. *Atmel ATmega128L - 8-Bit AVR Microcontroller with 128k in-System programmable Flash*, November 2004.

[5] UC Berkeley. Smote: Berkeley network sensor testbed. http://smote.cs.berkeley.edu/.

[6] J. Beutel, M. Dyer, L. Meier, M. Ringwald, and L. Thiele. Next-generation deployment support for sensor networks. Technical Report 207, Computer Engineering and Networks Lab, ETH Zürich, Switzerland, November 2004.

[7] J. Beutel, M. Dyer, L. Meier, and L. Thiele. Scalable topology control for deployment-sensor networks. Technical Report 208, Computer Engineering and Networks Lab, ETH Zürich, Switzerland, November 2004.

[8] Adam Chlipala, Jonathan Hui, and Gilman Tolle. Deluge: Data dissimination for network programming at scale. http://www.cs.berkeley.edu/~jwhui/research/deluge/cs262/cs262a-report.pdf, 2003.

[9] D. Culler et al. TinyOS: An operating system for Networked Sensors. http://webs.cs.berkeley.edu/tos.

[10] U. Frey. Topology and position estimation in Bluetooth ad hoc networks. Master's thesis, Computer Engineering and Networks Lab, ETH Zürich, Switzerland, March 2003.

[11] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. In *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, 2004.

[12] Lewis Girod, Thanos Stathopoulos, Nithya Ramanathan, Jeremy Elson, Deborah Estrin, Eric Osterweil, and Tom Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems*, Baltimore, MD, 2004. To appear.

[13] J.L. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November 2002.

[14] T. Hug and F. Süss. Mote/TinyOS meets BTnode, February 2004.

[15] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003)*, pages 126–137. ACM Press, New York, November 2003.

[17] K. Martin. Adaptive xtc on btnodes. Master's thesis, Computer Engineering and Networks Lab, ETH Zürich, Switzerland, May 2005.

[18] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. 4th Int'l Conf. Information Processing in Sensor Networks (IPSN '05)*, pages 364–369. IEEE, Piscataway, NJ, April 2005.

[19] Harvard University. MoteLab: Harvard network sensor testbed. http://motelab.eecs.harvard.edu/.

[20] BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. http://www.btnode.ethz.ch.