

Semester Thesis

TinyOS meets BTnode

6. Februar 2003

Thomas Hug
Florian Süß

Assistant: Jan Beutel, Martin Hinz
Professor: Lothar Thiele

Abstract

BTnodes are embedded devices with a Bluetooth module and an ATmega128 processor on it. This thesis describes the reprogramming of other ATmega128 devices, each connected as *target* to a BTnode which build a backbone network for code distribution. A development PC is used to enter new code into the network. This code is distributed via multihop to all BTnodes in the backbone network. Berkeley Motes were used as target devices because of their low power capabilities.

Contents

1	Introduction	5
2	Network Environment and Devices	6
2.1	Backbone Network	6
2.1.1	BTnode	6
2.1.2	ATmega128	7
2.2	Target Network	8
2.2.1	Structure	9
2.2.2	Target Devices	9
2.3	Network Node: Combination Host - Target	10
2.3.1	Alternatives	10
3	Remote Firmware Update	12
3.1	Approach	12
3.2	Programming Procedure	12
3.3	Realization in Host System	13
3.3.1	Init	13
3.3.2	Target Detected	14
3.3.3	Check Version	14
3.3.4	Programming	15
3.3.5	Failed	15
3.4	Target Programming	15
3.4.1	SPI Interface	15
3.4.2	Serial Downloading	15
3.4.3	Instruction Set	16
3.4.4	Software Implementation	17
3.5	Definitions	18
3.6	Performance	19
4	Usage	21
4.1	Installation BTnode System Software	21
4.2	Building BTnode Firmware	21
4.3	Runing Target Programming	22

4.4	Debugging on BTnodes	23
4.5	Multihopping	24
5	Conclusion	25
A	Aufgabenstellung	27

Chapter 1

Introduction

A problem in wide ad-hoc networks is to manage a large amount of devices. Software upgrades would not be possible if there are no mechanisms supporting the distribution of new program code to many embedded devices.

The BTnodes [14] are embedded devices with a Bluetooth radio interface. In a former diploma thesis [1], the distribution of program code in an ad hoc network with BTnodes was already implemented.

This thesis uses the code distribution capabilities of BTnodes to program other devices which must not be able to provide communication capabilities. It is possible to program any kind of devices with a certain compatibility in processor architecture to the BTnodes.

Several possibilities of code distribution and reprogramming were evaluated and the most suitable procedure was implemented on the BTnodes.

The basics of BTnodes and Bluetooth are described in the diploma thesis of Urs Frey [1].

Chapter 2

Network Environment and Devices

The BTnodes [14] are able to connect each other to form a scatter network. This functionality is provided by the *treenet* application.

The goal of this thesis is to find a solution to reprogram different nodes in a heterogenous network using a BTnode network as a backbone. Even devices with no Bluetooth device should be programmable.

Different approaches were discussed in section 2.5 to accomplish the task.

2.1 Backbone Network

The *treenet* application is responsible to interconnect the BTnodes to a tree (Figure 2.1). Using the Bluetooth inquiry function, the BTnodes search other BTnodes. The BTnodes then try to connect to each of these neighbor nodes. Every BTnode running *treenet* can be added or removed from the tree. The tree nodes are independent of each other and every node can be used to send data to the backbone network. The tree is used as a backbone network to transport data.

2.1.1 BTnode

The BTnode, developed at ETH Zurich [14], is a versatile, autonomous wireless communication and computing platform based on a Ericsson Bluetooth radio and an ATmega128 microcontroller [2]. Figure 2.2 shows the architecture of BTnodes. It serves as a demonstration and prototyping platform for research in mobile, ad hoc and distributed wireless sensor networks (WSN). Because the Bluetooth interface can transmit data very fast, reliable and has a wide range compared to other radio techniques, BTnodes are well suited to be used as backbone network.

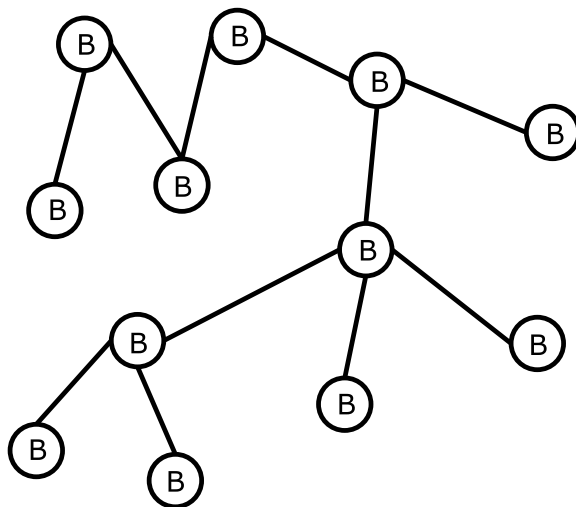


Figure 2.1: Treenet with BTnodes

2.1.2 ATmega128

BTnodes are equipped with an ATMEL ATmega128L microprocessor [2] with additional SRAM memory. Table 2.1 gives an overview of the different memory sections and their behavior.

Section	Size	on Reset	Persistence
Registers	32Bit	overwritten	volatile
SRAM intern (Stack)	4kB	initialized	volatile
SRAM extern (Data section)	60kB	initialized	volatile
SRAM extern (additional banks)	180kB	retain data	volatile
Flash	128kB	retain data	non-volatile
EEPROM	4kB	retain data	non-volatile

Table 2.1: Memory sections

It contains 128KB reprogrammable flash memory, which is divided into 64K words of 2 bytes each. These words are addressed by giving the 16 bit word address and one additional address bit to specify the low or high byte of the word. Usually, flash is used to store executable program code. The flash memory is divided into a boot flash section and an application flash section. If the processor is actually running in the boot memory section, the application memory section can be reprogrammed.

The 4K EEPROM memory section is used to store system information and program version numbers.

Additionally to the internal 4KB SRAM, ATmega128 provides an I/O

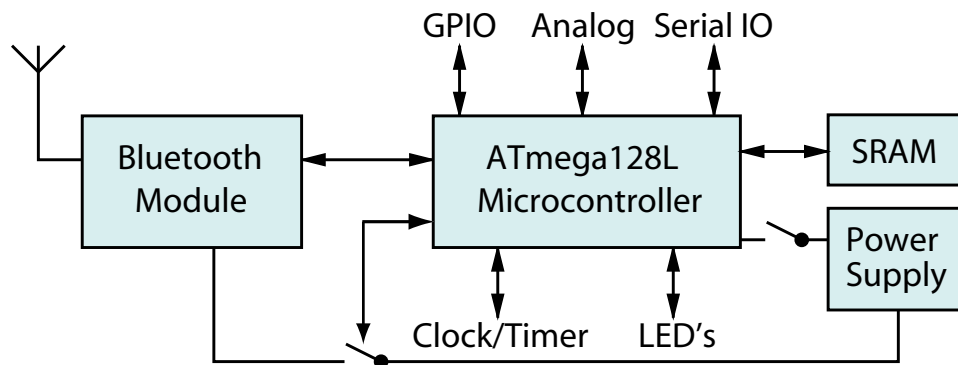


Figure 2.2: BTnode Architecture

memory interface for addressing 60KB external SRAM memory. Because this is insufficient for many use cases (for example to store new program code of 128KB before reprogramming the flash memory), there is an additional 180KB SRAM added, divided into 3 banks of each 60KB. In order to address these additional memory banks, two general purpose I/O pins of the processor are configured to act as additional address lines. All memories BTnode is equipped with are shown in figure 2.3. The details about implementation of the external SRAM banks are described in [1].

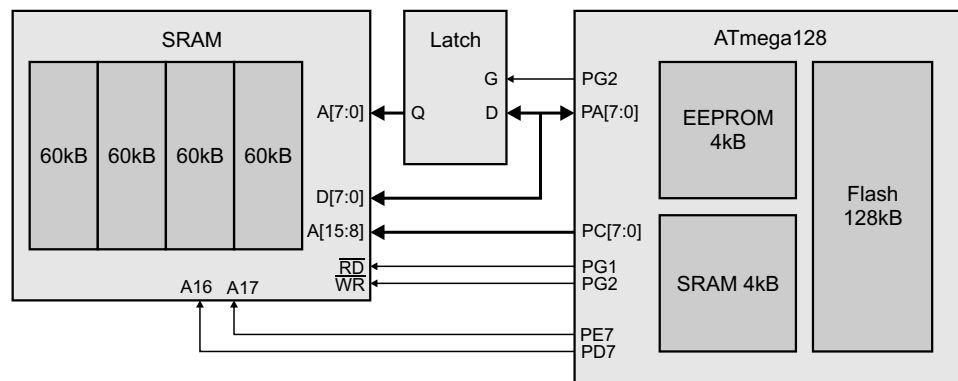


Figure 2.3: Memory Sections

2.2 Target Network

A set of several devices have to be updated. These devices build an overlay network (dashed lines in Figure 2.4) to the backbone network. A group of devices have the same purpose, for example a sensor network. The over-

lay network has only an unreliable character, the targets have no direct connection to each other and there must not be interaction between them.

2.2.1 Structure

The network is assumed to be heterogenous, as shown in figure 2.4. The BTnode hosts could be with or without a target. Additionally, multiple overlay networks with different target types are allowed. The overlay networks and therefore the targets are completely independent of the backbone network.

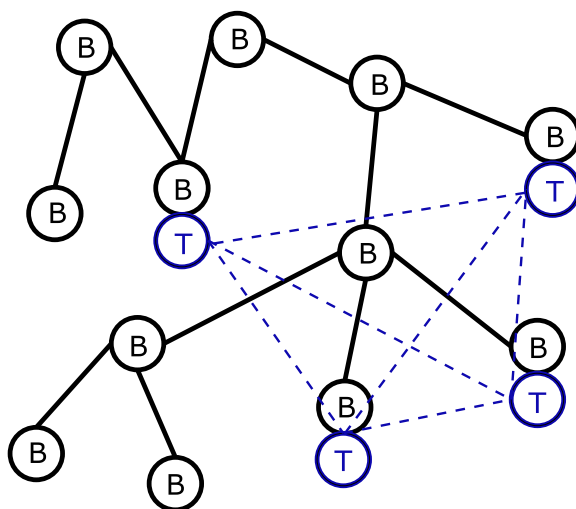


Figure 2.4: Treenet consisting of hosts (B) and targets (T)

2.2.2 Target Devices

The Berkeley Motes [7] were chosen based on their low power radio interface from ChipCon [10] and their ATmega128 processor. The ChipCon radio is more than ten times slower than the Bluetooth radio of the BTnode and also have a much shorter radio range. Therefore, they are not optimal to be used for code distribution. Additionally, Motes don't have any external SRAM memory. Instead they are equipped with 512KB serial flash, which is very slow and uses a lot of power.

In combination with code distribution and reprogramming, a transmission failure would cause a Mote to crash, because not all program code was transmitted and then a part of this code is missing. A restore would only be possible via PC with an appropriate programming toolkit (for example an AVR In-System Programmer) and is unacceptable in an environment with more than 100 nodes.

2.3 Network Node: Combination Host - Target

The fast radio of BTnodes and the SRAM memory provides a perfect solution to reprogram other devices. Code is distributed through the backbone network of BTnodes and the received code is stored in the SRAM. As soon as code was received and the distribution ended, the programming of the target is started.

2.3.1 Alternatives

Figure 2.5 reflects the combination of a host and a target device in a node of the *treenet*.

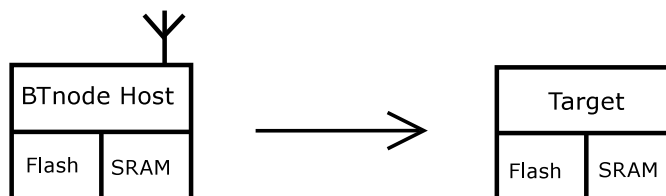


Figure 2.5: Combination of Host and Target

In order to program the target, the host device has to send new program code, that was distributed in the backbone network, to the memory of the target device. This could be done with a cable between the two devices or using a wireless technology.

It depends on the target device and especially on the microprocessor on it what techniques are reasonable to program target devices.

In this work, only devices equipped with an ATmega128 processor were used, and there are several programming mechanisms provided. At a node of the *treenet*, host and target are close to each other, so the present solutions are all based on a wired connection. There are several possibilities:

- *Serial Interface (UART)*. It would be possible to use a common serial interface to program the target. But then, the target device should also be able to communicate with the host device during programming to put retrieved data into flash. So, there is additional firmware to be programmed and loaded to the target device and the target has to participate active in the connection.
- *Serial Downloading*. With serial downloading, the target can be programmed without applying any functions on it. There is an instruction set provided that can be used by the host and is understood by the target. This way, it is possible to write new code into flash of the target without knowing anything about the firmware running on it.

- *Parallel Programming.* This works basically equally to serial downloading. It is much faster than serial downloading, but unfortunately uses much more connection pins. Additionally, parallel programming needs a higher voltage (5 Volts instead of 3.3 Volts for serial programming).
- *JTAG.* Similar to a serial interface, programming with JTAG control only needs few pins. It is quite similar to serial downloading, but is faster and better suited for debugging. But the implementation of a JTAG control connection is more complicated than the solution with serial downloading.

It would be nice to use a protocol that could read the status information about a target without rebooting it. This would have the advantage to provide full “plug and play” capability. The solutions *Serial Downloading*, *Parallel Programming* and *JTAG* all need to reset the target after each action concerning it. Therefore, a host cannot determine the exchange of a target without rebooting it.

Finally, there should be the possibility to switch off the backbone network and work only with the target networks in order to reduce power consumption.

Chapter 3

Remote Firmware Update

3.1 Approach

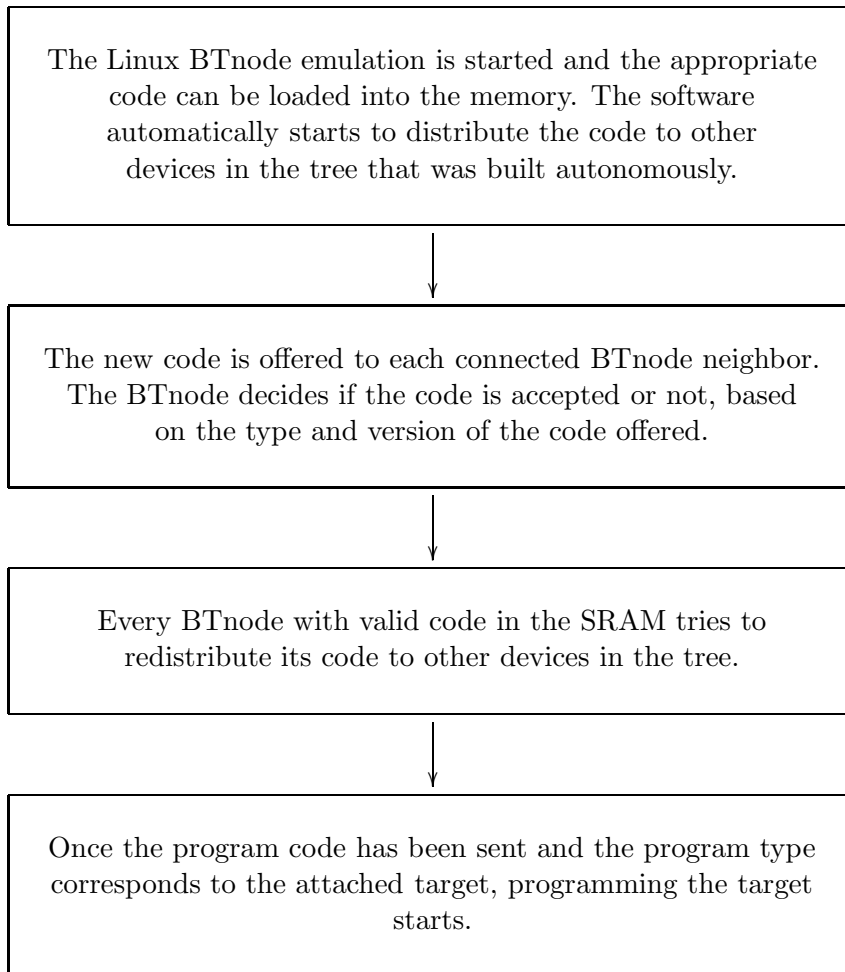
Given two connected devices, the simplest way of interaction in between is carried out if one of the devices is a master (host) and the other a slave (target). The master has full control over its slave. The slave is therefore independent of the master.

The ideal solution to achieve this control behaviour is *Serial Downloading* (see section 2.5). This solution uses the same procedure as an AVR In-System Programmer if the devices are directly programmed from a PC. A further advantage is the independence from the operating system running on the target. Even targets with no operating system running could be programmed. Only the programming interface has to be compatible.

A major disadvantage is the speed. The serial connection is a bottleneck concerning bandwidth and access time. A direct connection to the memory over the 8Bit bus is more than ten times faster and has a lower latency.

3.2 Programming Procedure

The following diagram gives an idea of the intended procedure:



Because target programming has to be flexible, a "plug and play" solution is needed. Independent of the target state, the host must be able to interrupt the target, reprogram it and reboot it at any given time.

3.3 Realization in Host System

Figure 3.1 shows the procedure that runs on the host BTnode. An arriving program packet triggers the init procedure if the program type matches the value of the attached target device.

3.3.1 Init

Initialization of host and target. The host checks if there is a target connected via the SPI interface. If the connection attempt returns a success, the host saves this flag in its EEPROM and changes to the state *target detected*.

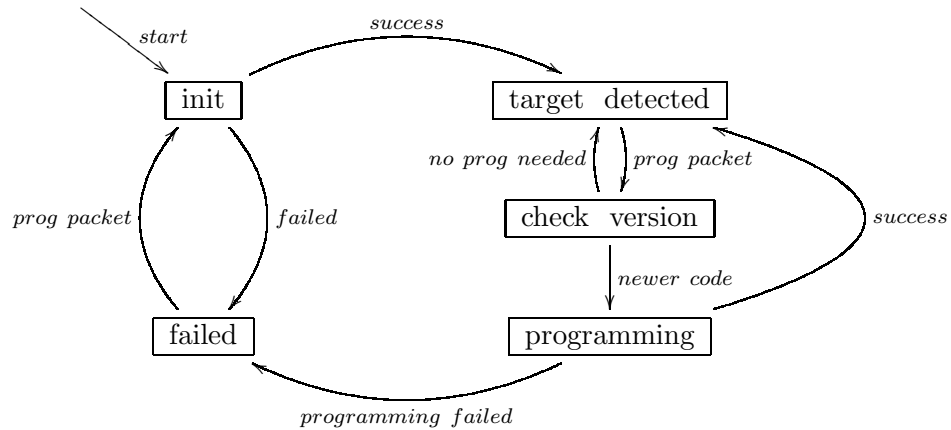


Figure 3.1: State machine in the host BTnode

If the connection attempt failed, nothing is done and the system changes to state *failed*.

The initialization procedure is executed after booting the host device and whenever the system is in state *failed* and a new programming request packet arrives.

3.3.2 Target Detected

In the state *target detected*, the EEPROM value `target detected` of the host system is true. It is mandatory that the hostsystem really has a target connected. Because of the Serial Downloading solution, it is not possible to check the connection without resetting the target device. The hostsystem therefore isn't informed when the target is disconnected. Only its last state is known.

3.3.3 Check Version

As soon as new program code of the target type is available on the host and the hostsystem is in the state *target detected*, the version of the new program is compared to the running version in the target. The version of the target is only saved in the hosts EEPROM. To gain more flexibility and transparency with different programs on targets and different targets, a routine to check the targets version directly was not implemented. The target operating system should be capable of independently using its EEPROM and memory.

If the offered program has a greater version number than the one in the hosts EEPROM, the program is accepted and the host system tries to

program the target in the *programming* state.

In case of an older program or an equivalent to the already loaded one, the request is denied and the system changes back to *target detected* with no effect.

3.3.4 Programming

The program data in the SRAM of the host system is sent to the target. If the programming attempt failed, the EEPROM value `target detected` is set to false and the system changes to *failed*.

On success, the new program version is saved in the EEPROM and the system changes to *target detected*.

3.3.5 Failed

The last initialization or programming attempt has failed. The EEPROM value `target detected` is false.

As soon as a new programming packet arrives, this state is left and a new initialization attempt is processed.

3.4 Target Programming

3.4.1 SPI Interface

ATmega128 processor provides a high-speed synchronous interface, called Serial Peripheral Interface (SPI). This interface is provided on BTnodes at port B (J6) using pins PB0 (used as reset), PB1 (clock), PB2 (master output / slave input) and PB3 (master input / slave output) of the ATmega128 processor. A device using SPI can be configured as master or slave, corresponding to the value of the master / slave select bit in the SPI Control Register. To utilize the SPI interface, the SPI enable bit in the SPI Control Register must be set.

A connection between two BTnode devices, both using the SPI interface, behaves like a common serial connection. In order to communicate to each other via SPI, the BTnodes have to use an instruction set given by ATMEL, described in section 3.4.3.

3.4.2 Serial Downloading

It is possible to program ATmega128 memory (flash and EEPROM) with the serial SPI interface, called *Serial Downloading*. This is basically the same that is done while uploading new program code to an embedded device with a programmer, for example with AVR In-System Programmer (ISP). Serial Downloading is not a synchronous communication anymore, the host sends instructions and the target has to execute them without any interaction. A

cable is needed to connect the SPI-port (J6) of the host device with the ISP-port (J1) of the target device (figure 3.2). To enter *Serial Downloading* mode, the reset pin of the target device must be set to 0. This functionality is provided by a general purpose input / output pin of the host device. Now, the target is in a 'dead' mode, stopped working immediately (all data in SRAM of the target are lost!) and is ready to receive new program code. When programming is finished, the reset pin must be set back to 1 to return to operation mode, then the target device reboots and starts executing the new code.

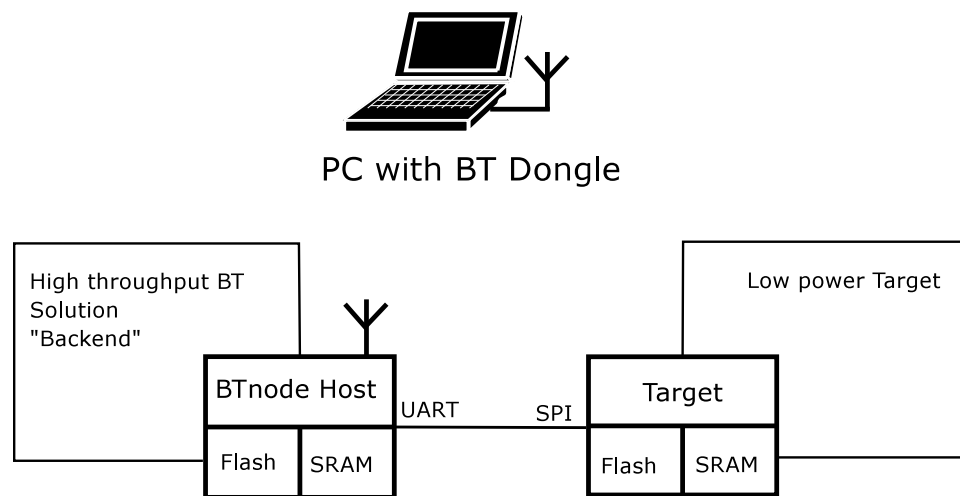


Figure 3.2: Combination of Host and Target

3.4.3 Instruction Set

The SPI Serial Programming instruction set uses instructions that are 4 bytes long. Each byte is echoed by the interface, this can be used to test if the connection is working properly. After pulling down the reset signal, the Programming Enable instruction ($0xac, 0x53, 0xFF, 0xFF$) must be sent from the host to the target ($0xFF$ stands for "don't care"), followed by the instructions needed for programming.

The most important instructions are:

- *Chip Erase*, ($0xac, 0x80, 0xFF, 0xFF$). Erases flash memory of the target device, all values in flash are set to $0xFF$. It is recommended to perform chip erase before starting downloading new code.

- *Read from EEPROM memory.* With instruction $(0xa0, 0xXa, 0xbb, 0x00)$, the value $0x00$ at address $0xab$ in EEPROM is returned.
- *Write to EEPROM memory.* To write data $0xii$ to EEPROM address $0xab$, instruction $(0xc0, 0xXa, 0xbb, 0xii)$ is used.
- *Read Memory Page.* Instruction $(0x20, 0xaa, 0xbb, 0x00)$ returns value $0x00$ of the low byte at flash address $0xab$ and $0x28, 0xaa, 0xbb, 0x00$ returns the value of the high byte at this flash address.
- *Load Memory Page.* With ATmega128, a whole flash memory page (128 words / 256 bytes) must be filled with data before writing it. $(0x40, 0xXX, 0xbb, 0xii)$ for low byte respectively $(0x48, 0xXX, 0xbb, 0xii)$ for high byte loads data $0xii$ to low or high flash address $0xbb$. Remember to use only values from 0 to 127 as flash addresses $0xbb$ due to the page size of 128 words.
- *Write Memory Page.* After loading, the page at flash address $0xab$ is written at once with instruction $(0x4c, 0xaa, 0xbb, 0xXX)$. For address $0xbb$, only the values 0 or 128 are allowed, because this is just the address of the memory page.
- *Device Code Test.* It is possible to test what equipment the connected target uses:
 Manufacturer test $(0x30, 0xXX, 0x00, 0xyy)$: returns $0x1e$ if processor was manufactured by ATMEL.
 Part family and memory size test $(0x30, 0xXX, 0x01, 0xyy)$: return value $0x9n$ indicates AVR with 2^n KB flash memory.

3.4.4 Software Implementation

- *Init Routine*

After the SPI Interface has been initialized (by setting the ATmega128 SPI pins and pull down reset pin of target device), the *init routine* checks if there is a target attached and if there is one, what processor it uses. Also there is a test of the amount of flash memory. If there is no ATMEL processor found, the action aborts and target programming fails.

These tests are always carried out when powering-up or rebooting the host device. If a correct target has been detected, the host reads the running program version and stores it in its EEPROM.

Finally, all SPI settings have to be reset, the pins used in the ATmega128 reset to original state and reset pin of target set to 1. The host now continues with normal work.

- *Download new code*

When a host decides to download new code to the target, it first checks if there is a target attached (executes again *init routine*), because if the host is in state *target detected* and the target was removed, it has no information whether the target is still attached or not. Also, the version number of the newly arrived code is read and compared to the running version on target. Should one of these checks fail, the newly arrived code is rejected and host system status changes to *failed*.

If the correct target was detected, the host system starts the programming routine by erasing the flash on the target.

With ATmega128 processor, flash memory is divided into pages of 128 words (256 bytes). To write to the flash, a whole page must be loaded and written afterwards. With a mechanism called polling it is possible to detect if page writing finished. One byte in this page is read and if the value change from *0xff* (after flash erase, all values are set to 0xff) to the correct value, the whole page was written.

- *Sending and receiving data*

Sending and receiving data is basically the same mechanism: the data byte is loaded into the *SPI Data Register* and is autonomously shifted out to the other device. As soon as sending / receiving data has finished, a flag in the *SPI Status Register* is set.

- *Event handling*

While sending or receiving data, the BTnode must wait until the data byte is shifted out of the data register. This is implemented in a loop. But in an event driven system, it is not allowed blocking the system in a loop without taking care about event handling, so while waiting for writing / receiving finished flag, an event handler is called in order to process all pending events.

- *Verifying*

In order to verify if data was written correctly, flash memory of the target must be read again and compared to the SRAM on host. This takes again the same amount of time like programming flash, so checking flash doubles the time used for programming the target (see section 3.6).

3.5 Definitions

Figure 3.3 reflects the implemented headerformat of the packets.

prog_type	prog_ver	active_flag	prog_len	CRC	...data...	CRC
1	4	1	4	2	prog_len	2

Figure 3.3: Packetformat

prog_type and *prog_ver* are used to identify the program code. The type describes the target device which is destined for this code. This could be the BTnode host itself or different types of targets. The definitions of program types are stored in *boot_defs.h*. *prog_ver* is additionally used to determine if a certain program is newer than another. The *active_flag* is set to 1 as soon as there is active program data in the memory. This byte is not transmitted and only used locally. The *CRC* (Cyclic Redundancy Check) value is a hash value over the header (first) and over the whole data (second). It is used to check the integrity of the header and payload data.

Additionally, the EEPROM of the BTnode host is used to store the following values:

- connection status of the target
- type of the target
- version running on the target

These values are identified by four letters and read and set with the *FOURC* macro function. The BTnode host uses the value in the EEPROM to decide if a certain program is required or not.

To store this target information, EEPROM memory is used because it is non-volatile. Contrary to using SRAM memory, target information is still available after rebooting the host device.

3.6 Performance

With the *Serial Downloading* protocol, there are always four bytes instructions sent to the other device in order to write or read one byte of target flash memory. So, remote programming takes a considerable amount of time:

- It takes about 60 seconds to programm the whole flash memory (128KB) of the target. This time may vary a lot because during programming, all pending events of the BTnode system are processed. Waiting a few seconds for building up a connection to another BTnode device while programming a target is not uncommon (although these connections have nothing to do with target programming).

- If flash verifying is performed after programming, this takes approximately the same time. Both programming and verifying take about 115 seconds of time.
- To compare, writing from SRAM to flash of the same BTnode takes about 2 seconds, and Bluetooth transmission of 128KB from one BTnode to another needs approximately 20 seconds.

Chapter 4

Usage

This chapter describes the installation process and usage of the target programming. A running Linux installation is assumed.

4.1 Installation BTnode System Software

Download the BTnode system software directly from CVS and compile it with the following commands:

```
CVSROOT=:pserver:anonymous@cvs.sourceforge.net:/cvsroot/btnode
export CVSROOT
cvs login
cvs -z3 co btnode_system

cd btnode_system
./bootstrap
./configure
make
make install
```

The libraries of the BTnode system software are installed to `/usr/local/btnode`.

4.2 Building BTnode Firmware

Proceed now to the target programmer:

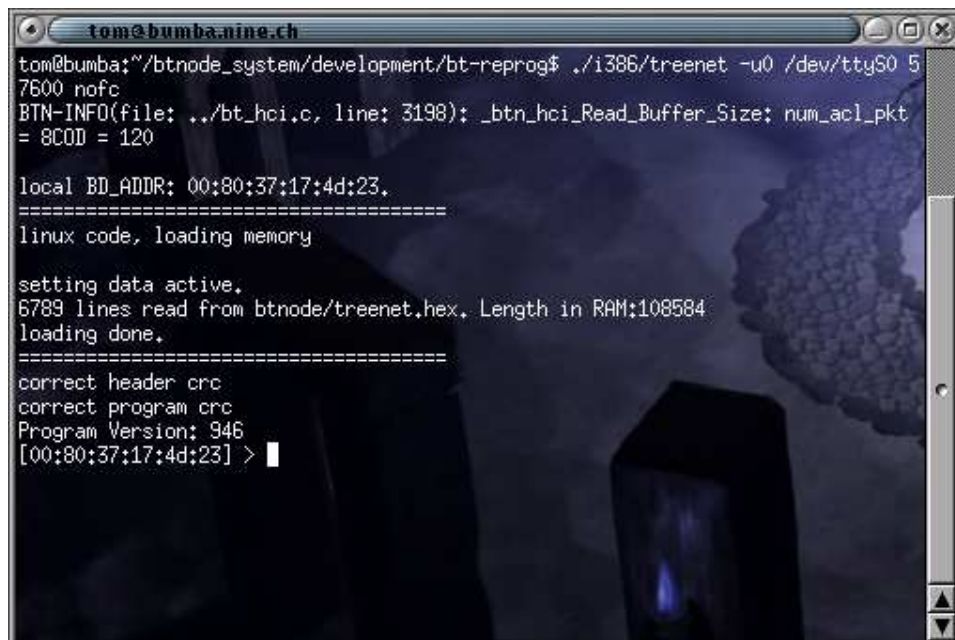
```
cd development/bt-reprog
make btnode
make i386
```

The first make is used to generate the firmware for the BTnodes while the second one builds the linux binary.

4.3 Runing Target Programming

The *treenet* binary in the *i386* directory is the Linux application. It is started (assumed a serial Bluetooth device is connected to */dev/ttyS0*) with:

```
./i386/treenet -u0 /dev/ttyS0 57600 nofc
```



```
tom@bumba:~/btnode_system/development/bt-reprog$ ./i386/treenet -u0 /dev/ttyS0 57600 nofc
BTN-INFO(file: ../bt_hci.c, line: 3198): _btn_hci_Read_Buffer_Size; num_acl_pkt
= 8COD = 120

local BD_ADDR: 00:80:37:17:4d:23.
=====
linux code, loading memory

setting data active.
6789 lines read from btnode/treenet.hex. Length in RAM:108584
loading done.
=====
correct header crc
correct program crc
Program Version: 946
[00:80:37:17:4d:23] >
```

Figure 4.1: Treenet application after start

If a binary for a target (Intel hex binary) is to be distributed, the process is initiated by the command *client* (Figure 4.2). The program looks for a binary called *client.hex* in the working directory. Figure 4.3 gives an overview to an application programmer developing software for targets.

The content of the binary is now in the memory. As shown in Figure 4.4, a set of parameters can be configured. A program version and a program type is set and is used by the distribution process to differentiate between old and new code. The program type was added to operate a network with different targets.


```

tom@bumba.nine.ch
=====
Menu:
inq          - execute inquiry, print results.
scan        - execute treenet scan.
ba          - let tree blink
br          - let the root node blink
bl          - let all leaves blink
bw          - display a complete walk through the graph.
b           - let levels blink, sequentially from root to leaves
print       - print discovered devices and state infos. [short 'p']
connect     - set auto-connect on/off (0 = off).
cc idx#     - open baseband connection to device w/ idx#.
dc hdl      - disconnect baseband connection w/ handle
con hdl psm - open l2cap channel to device w/ handle
disc l_cid  - disconnect l2cap channel.
send l_cid str - send on l2cap channel a chunk of legh.
destroy     - destroys completed tree
delegate    - tells a leaf to become my root node
ping index# - ping previously discovered device.
rname idx#  - request remote name of device w/ idx#.
role hdl    - discover role for connection handle.
sr hdl role - switch role for con_hdl to role {MASTER (0), SLAVE (1)}.
wlp hdl policy - write link policy (0: disable, 1: RSWITCH, 2: HOLD, 4: SNIFF,
8: PARK)
rlp hdl     - read link policy
ver         - print local version info.
name name   - change local name to name.
rsmode      - read scan mode : 0 none; 1 inquiry
wsmode mode - write scan mode: 2 page; 3 inquiry+page.
debug state# - turn 'on=1'/'off=0' tree_debug infos.
boot        - boots the btnode.
sram        - prints sram code (w/o program code).
now         - print current time.
program     - send program code to other btnode
client      - program a client
quit        - quit program.
=====
[00:80:37:17:4d:23] >

```

Figure 4.2: Treenet application menu

4.4 Debugging on BTnodes

To check the reprogramming on the BTnodes, it is possible to connect a PC's serial interface to the UART port of the BTnode. On the PC a common terminal emulation like *minicom* for Linux is used.

The *sram* function allows to show information about the actual content in the SRAM.

```
[00:80:37:17:4d:08] > sram
```

```

-----
| prog_type | prog_ver | active_flag | prog_len | crc | data crc |
|    1     |    45   |    1       | 105112  | 1   |    1     |
-----

```



Figure 4.3: Compiling

[00:80:37:17:4d:08] >

4.5 Multihopping

Figure 4.4 shows the distribution of code via multiple hops. The reprogramming is started not until every device in the connection list of a BTnode received the code. The entire process of programming a network is faster if the reprogramming is done after the distribution.

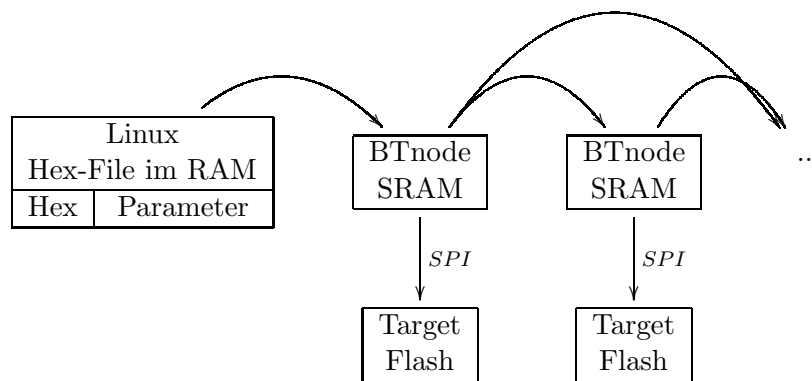


Figure 4.4: Code distribution

Chapter 5

Conclusion

In the current implementation, verifying flash memory of the target device is always executed after writing to it. Host and target devices are connected with a cable, the error rate of this direct connection is therefore very low. To speed up the reprogramming process, the verifying could be skipped without major disadvantages.

The flooding algorithm that is used to distribute uses no memory function. The default behaviour of a BTnode is to send its SRAM contents to each neighbor. In the current implementation the connection attempts are not systematically. An extension to improve stability and controllability would be a table with BTnode addresses and their current versions. To avoid inconsistencies over the program versions in the tree, a BTnode should be able to detect if the connected target or the BTnode itself has not an actual version. This could be achieved with a request for version function. A BTnode is then, for example after a reboot, able to ask the neighbor BTnodes for sending their actual version.

The developed software is only capable of communication in one way. A missing feature of a multifunctional network is to inquire the status of the devices and request logfiles. A tree network with thin asts has to provide a reliable bandwidth to process the whole data. It is possible that a radio of a Mote sens r is not as fast to transport enough data in a given timeframe. This management task has to be provided by the BTnode backbone network.

If this two way communication is implemented, a regularly notify message to the host BTnode could be used to check targets. The host BTnode is able to detect a failure in its target after a certain timeout of missing notify messages. This could be used to automatically reboot targets or perform a hard reset combined with new programming.

Reprogramming a connected target via cable allows building scalable multifunctional networks which are easily upgraded via wireless connections. The operating system running on the target device is completely independent of the host system. As targets, any devices with an ATMEL processor could be used and don't have to provide any wireless capabilities.

Appendix A

Aufgabenstellung

Einleitung

Eine bekannte Vision für ad hoc Netzwerke [3] geht davon aus, das unendlich viele, sehr kleine “Sensorknoten” kollaborativ ein Netzwerk und eine Applikation bilden. In anderen [4, 5] wird davon ausgegangen, das solche System weite Bereiche abdecken können und das die einzelnen Komponenten unterschiedliche Ressourcen aufweisen.

Die BTnodes bestehen aus einem Atmel AVR Mikrokontroller und einem Bluetooth Modul. Zusammen mit der im NCCR-MICS [6] entwickelten BTnode System Software bilden sie eine sehr kompakte programmierbare Netzwerk Plattform für mobile ad hoc Netzwerke. Heute kann ein BTnode zwar mit vielen verschiedenen anderen Bluetooth Geräten wie Handys und Laptops kommunizieren, ist aber dabei nicht sehr flexibel was die Netzwerk-topologie und Reaktivite angeht. Die Motes und ihr Betriebssystem TinyOS sind ein ähnliches System das an der UC Berkeley entwickelt [7, 8] und von Crossbow [9] kommerzialisiert wurde, das aber mit einem proprietärem Funkprotokoll auf Basis eines Chipcon CC1000 [10] arbeitet.

Diese Plattformen eignen sich hervorragend um sowohl Applikations Prototypen und neue Algorithmen aus dem Bereich der ad hoc Netzwerke zu implementieren. Innerhalb weniger Stunden kann ein “Neuling” die ersten Schritte erlernen und beginnen eigene Applikationen zu entwickeln. Ein grosses Problem besteht heute jedoch darin, eine auf einem oder wenigen Knoten entwickelte Applikation auf grossen Anzahlen dieser neuen Sensorknoten zu entwickeln, zu verteilen und zu betreiben.

Das Ziel dieser Arbeit ist es die verschiedenen Technologien der Mote- und BTnode Netzwerke zusammenzubringen und Mechanismen zu implementieren, die das verteilte Entwickeln und den Betrieb von heterogenen Netzwerken ermöglichen. Dazu sollen in dieser Arbeit mehrere Kommunikations Interfaces an einen AVR Mikrokontroller angeschlossen werden und simultan betrieben werden um den Zugang zu unterschiedlichen Net-

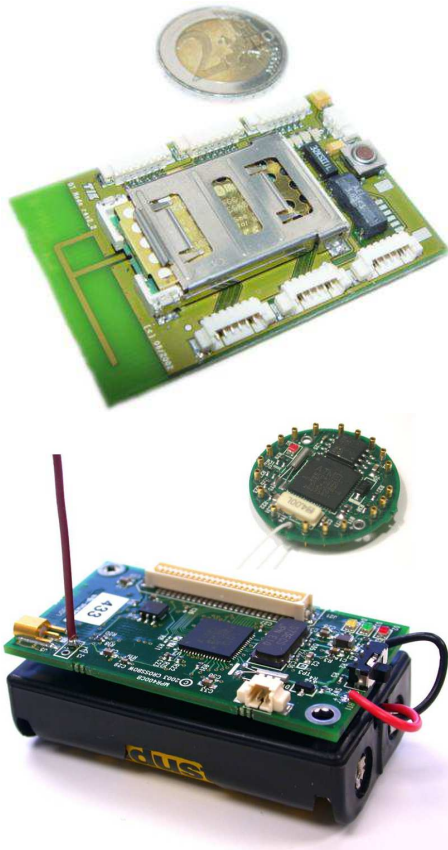


Figure A.1: A BTnode and some Mote devices.

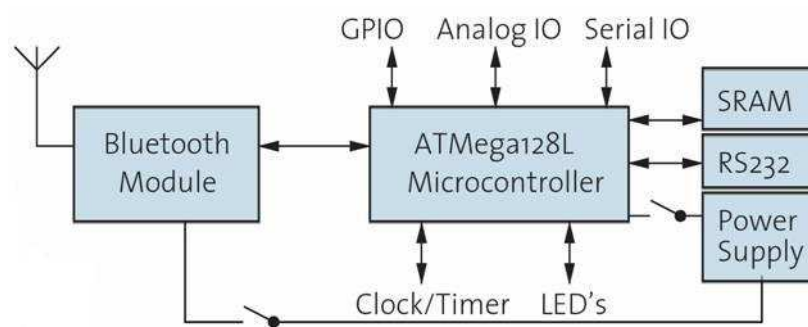


Figure A.2: BTnode System Übersicht

zwerksystemen zu ermöglichen.

Aufgabenstellung

1. Erstellen Sie einen Projektplan und legen Sie Meilensteine sowohl zeitlich wie auch thematisch fest [11]. Erarbeiten Sie in Absprache mit dem Betreuer ein Pflichtenheft.
2. Machen Sie sich mit den am Institut und bei Prof. Mattern bereits durchgeführten Arbeiten [12, 1, 14] vertraut. Es sollten möglichst viele Synergien aus schon durchgeführten Arbeiten genutzt werden. Führen Sie eine Literaturrecherche durch. Suchen Sie nach relevanten neuen Publikationen.
3. Arbeiten Sie sich in die Softwareentwicklungsumgebung der BTnodes sowie von TinyOS ein. Machen Sie sich mit den erforderlichen Tools vertraut und benutzen Sie die entsprechenden Hilfsmittel (online Dokumentation, Mailinglisten, Application Notes).
4. Für TinyOS wurde an der Universität Kopenhagen ein Bluetooth Protokoll Stack entwickelt [15], der auf Sourceforge unter `contrib\tinybt` verfügbar ist. Dieser kann auf den BTnodes oder auf einem Mote mit angeschlossenem Bluetooth Modul betrieben werden. Versuchen Sie ob dieser Stack zusammen mit dem standard CC100 Interface auf einem Mote betrieben werden kann. Versuchen Sie ausserdem die remote programming option des neuen TinyOS release 1.1.
5. Mit der `TreeNet` Applikation haben wir eine Backbone Infrastruktur geschaffen, die es ermöglicht flexibel und spontan viele BTnodes in einem transparenten Netzwerk zu verbinden. dazu wird ein baum zwischen allen BTnodes aufgebaut über den dann alle Knoten erreicht werden können. Solch ein baum könnte in der Zukunft als “Access Backbone” zu verschiedenen Teilen eines Sensornetzwerk Experimentes dienen. Versuchen Sie Teile des Netzwerkes mit neuem Bootcode neu zu programmieren.
6. Erarbeiten Sie ein Konzept das den flexiblen Zugriff und die Verteilung von Bootcode in einem heterogenem Netzwerk aus Motes und BTnodes ermöglicht. Beachten Sie hierzu das die Funktionen *Bootcode update*, *remote monitoring* und *remote debug* auf beliebigen Knoten eines Netzwerkes möglich sein muss. Hierzu gibt es verschiedene Möglichkeiten mit dem BTnode `TreeNet` als Backbone: (i) BTnode/Mote Sandwich Knoten bei denen ein programmier/debug bus die Verbindung herstellt oder (ii) ein dual-radio System mit einem CC1000 und Bluetooth Frontend and einem Host Controller.

7. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

Durchführung der Semesterarbeit

Allgemeines

- Der Verlauf des Projektes Semesterarbeit 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 Semesterarbeit 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 *6. Februar 2004* 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.

Bibliography

- [1] Urs Frey. *Topology and Position Estimation in Bluetooth Ad-hoc Networks*, Diploma Thesis, ETH Zurich, Winter Semester 2002 / 2003.
- [2] ATMEL. *ATmega128 Preliminary Complete*
http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf.
- [3] 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'I Conf. Mobile Computing and Networking (MobiCom 99)*, pages 271-278. ACM Press, New York, Aug. 1999.
- [4] 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, Jan. 2001.
- [5] L. Blazevic, L. Buttyan, S. Capkun, S. Giordano, P. Hubaux-J, and Y. Le-Boudec-J. Self organization in mobile ad hoc networks: the approach of terminodes. *IEEE Communications Magazine*, 39(6):166-174, June 2001.
- [6] NCCR-MICS: Swiss National Competence Center on Mobile Information and Communication Systems. <http://www.mics.org>.
- [7] J. Hill et al. System architecture directions for networked sensors. In *Proc. 9th Int'I Conf. Architectural Support Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93-104. ACM Press, New York, Nov. 2000.
- [8] D. Culler et al. TinyOS: An operating system for Networked Sensors. <http://webs.cs.berkeley.edu/tos>.
- [9] Crossbow Technology Inc. <http://www.xbow.com>.
- [10] Chipcon. *CC1000, Single Chip Very Low Power RF Transceiver*, April 2002.
- [11] E.Zitzler. Studien- und Diplomarbeiten, Merkblatt für Studenten und Betreuer. ETH Zürich, TIK, Mar. 1998.

- [12] R. Semadeni and L. Wernli. Bluetooth Unleashed, Wireless Netzwerke ohne Grenzen. Term thesis, Computer Engineering and Networks Lab, Swiss Federal Institute of Technology (ETH) Zurich, July 2001.
- [13] U. Frey. Topology and Position Estimation in Bluetooth Ad Hoc Networks. Master's thesis, Dept. Information Technology and Electrical Engineering, ETH Zurich, 2003.
- [14] BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. <http://www.btnode.ethz.ch>.
- [15] M. Leopold, M.B. Dydensborg, and P. Bonnet. Bluetooth and Sensor NETWORKS: A Reality Check. In *Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003)*. ACM Press, New York, Nov. 2003.