Experimental Sensor Network: Lessons from Hogthrob

by **Klaus Skelbæk Madsen**

Dept. of Computer Science, University of Copenhagen
Spring 2006

Abstract

This thesis is a part of the Hogthrob project, which aims to use sensor network technology for sow monitoring. A sensor network is defined as a collection of sensor nodes, each having sensing and communication capabilities.

Having a way to detect the start of a sows heat-period, will enable the farmer to inseminate the sow at the most beneficiary time. Performing the insemination at the right time will increase the chance of impregnating the sow. If the sows does not become pregnant, the farmer will have to feed the sow for 3 weeks before she enters heat again, thus raising the production costs. An increased activity of the sow have previously been shown to be a good indication of the sow being in heat.

In this thesis we design and deploy- a data gathering application to obtain data, that allows a method to detect the start of a sows heat-period to be devised. For this purpose we develop and deploy a sensor network application that can gather activity data for a sow. In the deployment we must monitor the sow for 20 days, to gather data showing both the non-heat and heat activity levels. To be successful this application needs to collect enough data that the detection model can be established. Establishing this detection model is beyond the scope of this thesis, but we show that the data collected during the deployment indicates an increased activity during the heat-period.

Furthermore we explore the possibility of using data compression for similar data gathering experiments, by evaluating 2 general purpose algorithms and one that is specifically geared towards our experiment. We show that using compression would be beneficiary for our application, but that the choice of compression algorithm can have a huge impact on the lifetime of the application.

Contents

1 Introduction **9**
- 1.1 Hogthrob . . . 9
- 1.2 Problem Definition . . . 10
  - 1.2.1 Compression . . . 11
- 1.3 Contribution . . . 12
- 1.4 Outline . . . 12

2 Related Work **15**
- 2.1 Node Hardware . . . 15
  - 2.1.1 UC Berkeley Motes . . . 15
    - The Mica Mote . . . 15
    - Mica2, Mica2Dot and MicaZ Motes . . . 16
    - Telos/T-Mote Sky . . . 16
  - 2.1.2 ETH Zürich BTnodes . . . 17
    - BTnode 2.2 . . . 17
    - BTnode 3 . . . 18
  - 2.1.3 Node Summary . . . 19
- 2.2 Node Software . . . 19
  - 2.2.1 TinyOS . . . 19
  - 2.2.2 BTnut . . . 20
- 2.3 Experimental Sensor Networks . . . 20
  - 2.3.1 Great Duck Island . . . 20
    - Lessons for Hogthrob . . . 22
  - 2.3.2 ZebraNet . . . 22
    - Lessons for Hogthrob . . . 23
  - 2.3.3 A Macroscope in the Redwoods . . . 23
    - Lessons for Hogthrob . . . 23
  - 2.3.4 Deployment of Industrial Sensor Networks . . . 24
    - Lessons for Hogthrob . . . 24
  - 2.3.5 Wired Pigs . . . 25
    - Lessons for Hogthrob . . . 25
- 2.4 Previous Farm Experiments . . . 26
  - 2.4.1 Automated Oestrus Detection on Sows . . . 26
  - 2.4.2 Health State Monitoring on Cows with Bluetooth . . . 26
- 2.5 Summary . . . 27

3 The First Hogthrob Experiment **29**
3.1 Goal . . . 29
3.2 Method . . . 29
3.2.1 Ground Truth . . . 30
3.2.2 Back Pressure Test . . . 31
3.2.3 Vulva Reddening Score . . . 31
3.3 Summary . . . 31

4 Node Hardware **33**
4.1 Sensor Board . . . 33
4.1.1 Accelerometers . . . 33
Range Options . . . 34
Interface Options . . . 34
Energy Consumption and Startup Time . . . 34
4.1.2 Accelerometers from Analog Devices . . . 35
4.1.3 Accelerometers from Freescale Semiconductors . . . 35
4.1.4 Accelerometers from STMicroelectronics . . . 35
4.1.5 Comparison of the different Accelerometers . . . 35
4.1.6 Other Possible Sensors . . . 36
4.1.7 Manufacturing the Sensor Board . . . 37
4.2 BTnode Modifications . . . 38
4.2.1 Reducing Energy Consumption on the Nodes . . . 38
4.2.2 Voltage Regulator for the Bluetooth Module . . . 39
4.2.3 Battery Charge Indicator . . . 39
4.3 Summary . . . 39

5 Low Level Software **41**
5.1 Accessing the Accelerometers . . . 41
5.1.1 The ADXL320 . . . 41
5.1.2 The LIS3L02DS . . . 42
5.2 Power Management . . . 42
5.2.1 Implementing Power Management . . . 43
5.2.2 Power Management on the Mica Motes . . . 44
5.3 TinyBT . . . 44
5.3.1 Problems in the original TinyBT Stack . . . 45
5.3.2 Duty Cycling the Bluetooth Module . . . 46
5.4 Storing Sensor Data . . . 47
5.4.1 Accessing the Flash . . . 47
Disabled Interrupts . . . 49
Busy Waiting . . . 50
5.4.2 Placing Code in the Boot Loader Area . . . 50
5.4.3 Finding Unused Flash Pages . . . 50
5.4.4 A TinyOS Component for Accessing the Flash . . . 51
5.5 Summary . . . 51
5.5.1 Further Work on TinyBT . . . 52

6 The Application **55**
6.1 Time Synchronization . . . 55
6.2 Flash Page Layout . . . 56
6.3 Offloading Data . . . 56
6.3.1 Memory Considerations . . . 57
6.3.2 Initiating Bluetooth Communication . . . 57
6.3.3 Choosing a Packet Type . . . 58
6.3.4 Bluetooth Communication Problems . . . 59
6.3.5 Transfer Protocol . . . 60
6.3.6 Duty Cycling . . . 62
6.4 In Field Debugging . . . 62
6.5 Reliability . . . 63
6.5.1 Making the Protocol Robust . . . 63
6.5.2 Automatic Reboot . . . 63
6.6 Size of the Final Application . . . 64
6.7 Summary . . . 65
6.7.1 Possible Improvements . . . 65

7 Energy Budget **67**
7.1 Battery Choices . . . 67
7.2 Battery Experiments . . . 69
7.2.1 Discharging the Batteries . . . 69
7.2.2 Capacity Results . . . 70
7.3 Energy Consumption of the Node . . . 71
7.3.1 Measuring Energy Consumption . . . 72
7.3.2 Current Consumption Estimations . . . 73
7.4 Summary . . . 74

8 Field Experiment Setup **75**
8.1 Sow Marking and Node Pairing . . . 75
8.2 Nodes . . . 75
8.3 Cameras . . . 77
8.4 Servers . . . 79
8.5 Bluetooth . . . 79

9 Field Experiment Results **81**
9.1 Results of Manual Heat Detection . . . 81
9.2 Problems With Node Check-in . . . 82
9.3 Unexpected Node Reboots . . . 83
9.4 Server Problems During the Experiment . . . 83
9.5 Data Extraction . . . 84
9.5.1 The Original Extraction Algorithm . . . 84
9.5.2 Taking the Experiment Problems into Account . . . 85
9.5.3 Anomalies in the Extracted Data . . . 86
9.6 Validating the Collected Data . . . 87
9.6.1 Verifying the Correctness of the Dataset . . . 87

9.6.2 Correlating the Acceleration Data to the Video 89
9.7 Node Lifetime 91
9.8 Lessons Learned 93
9.9 Summary 94

10 Compression **95**
10.1 Overview of the Data 95
10.2 Choosing Compression Algorithms 96
10.3 Huffman Coding 98
10.3.1 Generating the Code Table 98
10.3.2 Implementation Notes 100
10.3.3 Choosing the Best Code Tables 101
10.4 Lempel-Ziv 77 101
10.4.1 Implementation Notes 102
10.5 A Simple Data Specific Algorithm 102
10.6 Compression Framework 102
10.6.1 Compression Algorithm Interface 103
10.6.2 Testing the Algorithms on the PC 103
10.6.3 Testing the Algorithms on the Node 104
10.7 Testing the Compression Algorithms 104
10.7.1 Compression Ratio 104
10.7.2 Compression Time and Energy Consumption 105
10.8 Results 105
10.8.1 Expected Results 105
10.8.2 Code Size and Memory Usage 105
10.8.3 Compression Ratio 106
10.8.4 Compression Speed and Current Consumption 107
10.9 Would Compression Have Helped? 109
10.10 Summary 109
10.10.1 Further Work 110

11 Conclusion **111**
11.1 Future Work 111
11.2 Future Work on Similar Applications 112

Appendix **113**

A Bibliography **113**

B Sow Movements **119**

Chapter 1
Introduction

We will start by presenting the goals and requirements of the Hogthrob project. We will then in detail present the problems in the part of the Hogthrob project, that this thesis covers, and outline the approach we will take. We describe the contributions of this thesis, and present an outline for the rest of the thesis.

1.1 Hogthrob

The Hogthrob project[1] is a research project, with the goal to build a sensor network infrastructure for sow monitoring. The project is a collaboration between the following entities:

- Dept. of Informatics and Mathematical Modeling, Technical University of Denmark (DTU)
- Dept. of Large Animal Science, The Royal Veterinary and Agricultural University (KVL)
- The National Committee for Pig Production
- IO Technologies
- Dept. of Computer Science, University of Copenhagen

Currently the systems used for sow monitoring are primarily based on RFID ear tags. Such a tag can be used to identify the sow, and to control the amount of food dispensed to the sow at the feeding stations. Furthermore the feeding stations can be programmed to lead specific sows outside the pen, making it easy for the farmer to round up them up.

Some of the problems with these tags are:

No easy way to locate a specific sow - In some cases, e.g. when a sow is sick, the feeding station software can alert the farmer that a specific sow has not been at the feeding station for an extended period. In such a case, the farmer has to manually find the sow in the pen, using a hand-held RFID tag reader. This reader must be close to the ear tag to read the RFID tag, making this process impractical.

No reliable way to detect heat - When a sow enters heat, she must be inseminated within a short period of time. If this does not happen, there is a 3 week period before the sow is in heat again. The currently available solution, is to place a box with a boar (i.e. a male pig), inside the pen. When a sow approaches the boar, her ear tag is read. The closer to the heat-period, the more

[1] http://www.hogthrob.dk

often the sow approaches the boar. However, sows establish themselves in a strict hierarchy, and the sows low in the hierarchy might not approach the boar at all.

It is these two problems that the Hogthrob project strives to resolve, through the use of sensor network technology.

With regard to the heat detection, it has previously been shown, that the activity of the sow can be a good indicator of when it is in heat, as a statistically significant change in the activity of the sow occurs[26]. However this experiment was carried out on sows that were housed individually. In Denmark, the legislation requires that sows are housed together in large pens for most of the time. Therefore we will need to re-validate the results from the previous experiment. We will also need to devise a detection algorithm, that can be used to detect the heat-period.

We will need an experiment collecting data, to devise this detection algorithm. This requires that we monitor the activity patterns of several sows, in their pen. The more data is collected, the more certain we can be of being able to devise a model.

A way to collect these activity patterns would be to monitor the pen using video cameras, and track the sows this way. However it will not be easy to discern the sows from each other. Instead we opt to use sensor nodes to collect the data. Using sensor nodes has the advantage that we can simply attach nodes to the sows we wish to monitor.

This data collection experiment will be the main focus of this thesis.

1.2 Problem Definition

Using sensor nodes to collect activity patterns from sows requires that we design and build and an application for this purpose. However before we can start to design the application, we will have to provide answers to some basic questions:

How to measure the activity? Others have successfully used accelerometers[26] for this purpose, so we will choose the same approach.

For how long should we measure? We will have to gather data, both in the heat-period and outside the heat-period, to be able to compare. We need both periods from the same sow, as the activity patters might not be the same across sows. If we collect data from the end of a heat-period and 20 days forward, we can be fairly certain that the sow will enter the next heat-period.

How often should we obtain measurements? We want to collect as much data as possible, to make it easier to devise the model for heat detection. On the other hand, the amount of data we gather will affect the lifetime of the node, as we will spend more energy on sampling. So this will have to be a tradeoff.

Once we have these questions answered, we can begin to answer the questions about the design of the data gathering application:

Which sensor node to use? There are many available. However we already have experience with the BTnode 2.2 from ETH Zürich, and have enough nodes available to carry out the experiment. Therefore we wish to use this node.

How should we power the sensor node? Since the node will be attached to the sow, it has to be powered by some sort of battery. It is an indoor stable, so

using solar panels or the like is not possible. We will have to find a way to provide enough power for the node, so that it can function throughout the duration of the experiment.

How do we protect the node? Sows are very curious, and will play with almost anything they can get hold of. Since the sows are housed many together, we need to protect the node from the other sows. The packaging must also be able to protect the node from the ammonia in the air, and the manure on the floor.

How do we protect the sow? While the packaging protects the node, we also need it to protect the sow. The chemicals in batteries, or the electronics in the node can potentially hurt the sow. This must of course not happen.

Do we store the data on the node or do we offload it? If we make the assumption that we store one bit each second to indicate if the sow have been active or not in the last second, we will end up with 210 KiB of data. This should be possible to store on the node. However we do not know how large the acceleration should be, before we can consider the sow to be active. If we instead assume that a byte is needed each second, this will result in 1.6 MiB, more than we can reasonably expect to store on a sensor node.

How can we offload the data? We will need some kind of radio, and the obvious choice is the Bluetooth module on the BTnode. There should be plenty of power in the stables, to establish an infrastructure that can we can offload the data to, and which covers the entire pen.

Will the sensor nodes last throughout the experiment? If the node can survive for the duration of the experiment, depends on the answers to many of the other questions. The energy consumption will depend on how often we sample, how we offload the data and how often we offload data. However if we use the Bluetooth to offload the data, we will not be able to leave the radio on for the duration of the experiment, because it uses too much power.

How should the radio be duty cycled? This in turn depends on how much data we gather, and how much we will be able to store on the node. How much we can store on the node will depend on how we store it, and compression might be practical for this.

We will explore all of these problems detail, and conduct the data collection experiment. However we will not include compression in the deployed application for two reasons:

1. We do not know much about the data we collect. Therefore it can be hard to choose a compression algorithm that fits it.
2. We want the deployed application to be as deterministic as possible. Compressing the data will affect the duty cycle of the radio, as we either will have less data to offload, or will offload it at unknown points in time.

Instead we will evaluate the compression algorithms when the field experiment is over, in order to evaluate how it would have affected the experiment.

1.2.1 Compression

It has been argued that compression could be used in sensor networks to lower the energy consumption, especially on nodes where data is aggregated from several

sources[41]. However few compression algorithms are available for the applications in typical sensor network operating systems such as TinyOS.

Compression have been used as a way to increase the bandwidth[60], and with the goal of lowering energy consumption[17, 48], but a through analysis of how compression algorithms affect the energy consumption have not been done.

A data collecting application such as ours, is an obvious candidate for compression. Therefore we wish to evaluate how using compression to store and transfer data would have affected our field experiment. Performing this evaluation after the experiment allows us to test different compression algorithms against each other, with real data.

We will test the compression algorithms with regard to compression ratio, energy consumption and speed of the compression they can provide when compressing the data from the field experiment. All of these properties can affect how useful a compression algorithm is in a sensor network.

To perform these tests, we will develop a framework, that allows us to test the algorithms both on a sensor network node, and on a PC. Being able to run the algorithms on the PC is important as it will speed the development of new algorithms, as debugging them becomes easier.

1.3 Contribution

The contributions made by this thesis are as follows:

- We introduce a buffer management system in TinyOS, to allow the radio on the node to function at full speed (Section 5.3).
- To duty cycle the radio, we present a novel approach to data storage on the nodes (Section 5.4).
- We design a sensor network application, driven by the needs of the data collection experiment (Chapter 6).
- We deploy the sensor network, and describe the lessons we have learned from the deployment (Chapter 9).
- We design and develop a framework for testing different compression algorithms (Section 10.6).
- We show that a compression algorithm specifically designed for the collected data performs better than generic algorithms (Section 10.8).
- We show that compression could have a positive effect on our experiment, but the choice of compression algorithm can severely affect the lifetime of the node (Sections 10.8 and 10.9).

Designing the actual heat-detection algorithm from the gathered data is future work, and will be carried out by Cécile Cornou at KVL.

1.4 Outline

We will start by looking at previous contributions to the sensor network area, in order to gather as many lessons as possible, before proceeding with our own experiment (Chapter 2). We then describe the goal and method of our experiment,

and how to establish the ground truth (Chapter 3). We proceed to cover the hardware for the experiment. We select sensors, design a sensor board, and we lower the energy consumption of the hardware as much as possible (Chapter 4).

We then focus on the software required to support our application, such as drivers for the sensor board and Bluetooth radio (Chapter 5), before designing the application and protocols to use in the experiment (Chapter 6). We provide an estimation of the energy consumption of the node, to find out if our application can run for the entire experiment (Chapter 7). We describe how the infrastructure for the experiment have been installed in the stables, and how to protect it and the node from the environment (Chapter 8). We extract the data gathered by the nodes during the experiment, and we verify that it can be correlated to the ground truth (Chapter 9). Lastly we evaluate different compression algorithms, and look at how they could be used to improve the application (Chapter 10).

Chapter 2
Related Work

In this chapter we will describe the hardware, applications and deployments others have created, to extract the lessons and experiences that we can use for our deployment.

We start by looking at the most popular hardware platforms, together with the platform that we will specifically be using, to provide a view of the forces and weak points of different platforms. After describing the platforms, we will describe two software systems that specifically targets sensor networks.

Then we will describe several recent sensor network deployments that resemble our deployment, or from which we can learn important lessons. Lastly we will describe other farm experiments that have not been performed within the sensor networking scope.

2.1 Node Hardware

In this section we will describe a small part of the available nodes, their capabilities, forces and weak points. Currently many different platforms exists, but we will focus on the newer and most popular UC Berkeley motes, and the BTnodes developed at ETH Zürich.

2.1.1 UC Berkeley Motes

UC Berkeley have been at the center of sensor network research, since the field started to attract attention. The original idea of "Smart Dust"[58], i.e. a large collection of one cubic-millimeter computers communicating through the use of optics and lasers, has given way for larger nodes with more processing power and sensing capabilities, organized in smaller networks than originally envisioned.

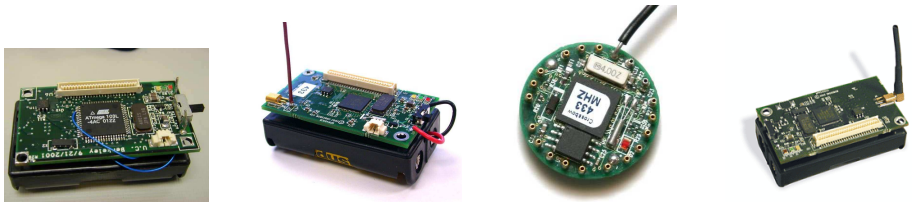
The Mica Mote

The Mica series of sensor network nodes — called motes — are the most popular nodes currently available. They are manufactured by Crossbow Technology[1].

The original Mica design (see Figure 2.1(a)) is no longer in production but consisted of an Atmel ATMega103 micro-controller, combined with a TR1000 radio transceiver[29]. 4 Mbit of external flash was also connected to the ATMega103, which could be used to store sensor measurements, or as temporary storage for a new program image. The Mica was fitted with a co-processor that could reprogram the ATMega103 from an image stored in the external flash. Power was provided

[1] http://www.xbow.com

Figure 2.1 – *The Mica Mote Family*



(a) Mica (b) Mica2 (c) Mica2Dot (d) MicaZ

from a battery holder, that could hold two AA batteries. To allow the mote to function as the batteries were depleted, a boost converter was included, which provided the node with 3 V, from input voltages as low as 1.1 V.

The Mica mote did not have any sensors built in. Instead sensor boards could be attached through a 51-pin I/O expansion connector. This connector provided access to most of the ATMega103's I/O-pins, including I^2C, SPI and Analog to Digital, enabling the use of many common sensors.

Mica2, Mica2Dot and MicaZ Motes

The Mica2, Mica2Dot and MicaZ motes are all descendants of the original Mica mote. All three feature the Atmel ATMega128 and 4 Mbit of external flash[16]. However they differ in the choice of radio and form factor.

The Mica2 — shown in Figure 2.1(b) — uses the ChipCon CC1000 radio, which can operate in different bands, depending on the regional requirements. The Mica2 features the same I/O expansion connector as the original Mica mote, making it possible to reuse sensor boards.

The Mica2Dot — shown in Figure 2.1(c) — is basically a Mica2 the size of a quarter (25 mm diameter). The mote is powered by a single 3 V coin cell battery, which is attached directly to the mote. The mote has 18 expansion pins, to which sensor boards can be attached.

The MicaZ (see Figure 2.1(d)) is the newest member of the Mica family. This mote features the ChipCon CC2420 radio, which is IEEE 802.15.4 compatible. Otherwise the node is similar to the Mica2, in that it uses the same micro-controller, has the same external flash, the same I/O expansion connector and from factor.

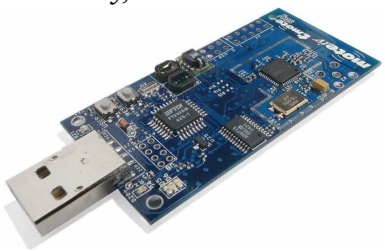
Neither of these motes feature a boost-converter. The boost-converter was omitted, because it will cause a higher power consumption as the input voltage drops. This causes the nodes to use more power as the batteries are depleted, which again leads to a lower lifetime[49]. The implication of this choice, especially on measurements from the ADC, does not seem to have been explored in depth.

Telos/T-Mote Sky

The Telos mote is the newest mote developed at UC Berkeley, and is manufactured by Moteiv[2] under the T-Mote brand (see Figure 2.2). The focus of the Telos mote have been to lower the power consumption, make it easier to use the node, and

[2] http://www.moteiv.com

Figure 2.2 – *The T-Mote Sky, the commercial version of the Telos mote*



make the node more robust[50].

To lower the power consumption, a micro-controller from Texas Instruments called the MSP430 is used instead of the Atmel ATMega128. The reasoning behind this choice is that normal sensor network applications spend most of their lifetime in sleep-mode, and the MSP430 only uses 1 μA in sleep-mode, where the ATMega128 uses about 20 μA[50]. Apart from the lower power consumption, the MSP430 is capable of operating on a supply voltage of 1.8 V, where the ATMega128 needs 2.7 V. This means that it is possible for the MSP430 to operate off two AA batteries, until these are completely depleted.

To make the mote easier to use, it features a built in USB programmer, so that it can be connected directly to the USB port of a PC, and programmed this way. The main advantage of this is in the development stages, where the mote does not require external programmers.

To make the mote more robust, sensors are integrated on the mote. It features an optional SHT11/SHT15 humidity and temperature sensor, and optional light sensors. The fact that the sensors are integrated with the mote, is supposed to give the design additional robustness. If there is need for other sensors two expansion connectors are provided.

The Telos mote uses the ChipCon CC2420 IEEE 802.15.4 compatible radio, just as the MicaZ.

2.1.2 ETH Zürich BTnodes

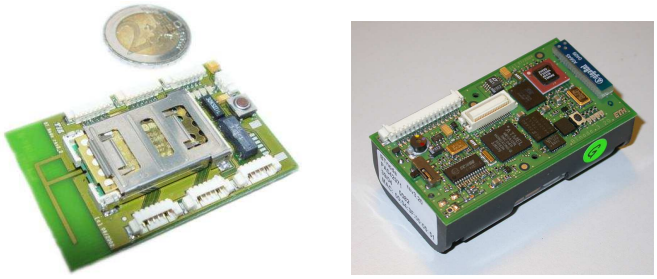
At ETH Zürich, they created the BTnode platform, to prototype wireless sensor network applications quickly[32]. The BTnode platform uses Bluetooth for the wireless communication. The benefits of using Bluetooth is that a very high bandwidth is available, compared to both the CC1000 and IEEE 802.15.4 compliant radios. But Bluetooth also enables easy communication between the nodes and a PC, eliminating the need for a gateway device.

The BTnode 1 was a prototype for use in the Smart-Its project[32]. We will not go into detail about this, as it resembles the BTnode 2.2 quite a lot, both in the choice of components and in the capabilities.

BTnode 2.2

The BTnode revision 2.2 — shown in Figure 2.3(a) — is based on the same Atmel ATMega128 that is used by the Mica motes. The ATMega128 is connected to an Ericsson ROK 101 007 Bluetooth module, which was the first commercially available Bluetooth module[32]. To the ATMega128 an additional 60 KiB of memory is con-

Figure 2.3 – *The newer BTnodes*



(a) BTnode 2.2 (b) BTnode 3

nected, making a total of 64 KiB available to applications. Some nodes are equipped with in total 240 KiB of external memory, which is made accessible as 4 separate memory banks.

The node has a built in voltage regulator, that makes it possible to use batteries to power the node. The voltage regulator can handle voltages in the range between 3.0 V to 16 V. A second voltage regulator controls the power to the Bluetooth module. This is necessary to completely power off the Bluetooth module when it is not in use.

The ROK 101 007 is connected to one of the ATMega128's two UARTs. The two can communicate at a speed of 460.8 kbps, which should make it possible to use the full bandwidth of the Bluetooth protocol.

The BTnode does not have any built in sensors. However most of the unused pins on the ATMega128 are available through 6 connectors on the node, making it easy to attach sensors to the node.

The default OS for the BTnode is the BTnut operating system, which is a simple OS written in C[10] (see Section 2.2.2 for more details). At DIKU a port of TinyOS, complete with a simple Bluetooth stack have been created[36].

BTnode 3

The BTnode 3 is a descendant of the BTnode 2.2. It still uses the Atmel ATMega128 as the MCU, but instead of the old Ericsson Bluetooth module, it includes both a ChipCon CC1000 radio — as is also found on the Mica2 node — and a Zeevo ZV4002 Bluetooth module. Both radios can be switched on and off independently by the MCU. The node is shown in Figure 2.3(b).

The inclusion of the ChipCon radio makes the BTnode 3 able to both participate in networks with Mica2 nodes, and run the same TinyOS programs with very few changes. While the Bluetooth module have been changed, it still provides the same HCI interface,[3] so BTnut applications designed for the BTnode revision 2.2, should also work with few changes.

Other changes to the design includes a built in battery holder for two AA-sized batteries. Since the node needs 3.3 V to operate, a boost converter is included

[3]Host Computer Interface, the communication protocol between the Bluetooth module and the host computer. Since most Bluetooth modules use this interface, it is simple to replace one module with another.

Table 2.1 – *Comparison of the different sensor network nodes*

Node	MCU	Clock (MHz)	RAM (KiB)	Radio	Bandwidth (kbps)
Mica	ATMega103	4.00	4	TR1000	115.0
Mica2	ATMega128	7.37	4	CC1000	76.8
Mica2Dot	ATMega128	7.37	4	CC1000	76.8
MicaZ	ATMega128	7.37	4	CC2420	250.0
Telos	MSP430	8.00	10	CC2420	250.0
BTnode 2.2	ATMega128	7.37	64	ROK 101 007	433.9
BTnode 3	ATMega128	7.37	244	ZV4002, CC1000	433.9, 76.8

that can deliver 3.3 V from an input voltage of down to 0.5 V. Furthermore the 6 connectors on the BTnode 2.2 have been replaced with a single connector, much like the I/O expansion connector on the Mica-family motes, that makes it possible to stack sensor boards on to the node.

2.1.3 Node Summary

In Table 2.1, the main characteristics of the different nodes we have described, is listed. As can be seen from this table, most of the nodes use similar hardware. The advantages of this is that it allows code reuse, and enables the nodes to communicate, even across different platforms.

Apart from the described nodes, many others exist which are either specialized to a specific task (such as the node developed for ZebraNet, see Section 2.3.2), or with hardware that differs greatly from the norm (such as the Intel Mote with a 32-bit MCU, see Section 2.3.4). However the Mica family are without a doubt the most popular.

2.2 Node Software

Several operating systems which specifically target wireless sensor network applications exists. The range of hardware supported by each operating system differs greatly, just as the services provided by the individual operating systems does. We will present the two OS's that are most important for our work, TinyOS and BTnut.

2.2.1 TinyOS

TinyOS[28] is a very simple event driven operating system. Its main focus is on the Mica family. Amongst other things it includes several different networking protocols, and various applications, such as TinyDB which presents the sensor network through a SQL-like interface. TinyOS have been ported to several different platforms, including the BTnode. This port includes an implementation of the lower layers of the Bluetooth stack, and makes it possible to communicate with other Bluetooth devices at the ACL level.[4]

TinyOS was developed with modularity and resource optimization in mind. Therefore all functionality in TinyOS is encapsulated in components. At compile-

[4]The ACL level is the lowest communication protocol in a Bluetooth module, which accessible from software. Higher level protocols are implemented on top of the ACL protocol.

time, only the components requested by the application programmer is included in the compiled program, thus lowering the memory and power requirements as much as possible.

TinyOS is written in the C-extension called nesC[25]. nesC enables the component model of TinyOS, and adds some valuable tools, such as compile time warnings for possible races in the code, and in-code documentation of interfaces. Furthermore it supports the annotation of commands and events, so that it is possible to see which commands and events are allowed to be executed in interrupt context — called `async`. Commands and events that are marked with `async` should not be used at the application level, but only in the low level software — such as drivers — that directly interacts with interrupts.

While the event-driven component-based model makes it easy to replace parts of the system, it also makes it harder to perform other tasks. Since the application only can respond to events, it is often necessary to create state-machines to keep track of what the application should do next.

2.2.2 BTnut

At ETH they have developed an operating system for use with the BTnode called BTnut[10]. BTnut is based on Nut/OS[5], an operating system for the Ethernut system. An Ethernut system is an Atmel ATMega128 connected to an Ethernet controller. Some of the main features of Nut/OS is cooperative multitasking, events, timers and dynamic heap application.

BTnut keeps this feature set, but adds Bluetooth support, by implementing the high level Bluetooth protocols, such as RFCOMM and L2CAP. BTnut is, just as Nut/OS, written in C.

Because of its design, BTnut is not a very flexible solution. If there are features of the system that are unneeded for a specific application, it is not easy to drop these. If one wants to minimize the memory requirements, this can critical. However, it provides a high-level system interface, which can help the application programmer to focus on the application.

2.3 Experimental Sensor Networks

There have been much research in the sensor network field, but it is only recently that real world deployments of sensor networks have begun to gather speed. In this section we will describe some of these deployments, and gather what we can learn and relate to the Hogthrob project from each deployment.

2.3.1 Great Duck Island

The Great Duck Island project[6] (GDI) is a habitat monitoring application, developed for use on the Great Duck Island[41]. The goal of the project is to monitor the nesting habits of the Leach's Storm Petrel, which nests on this island. Ordinarily researchers have visited the island outside the breeding season, to inspect the abandoned nesting burrows without disturbing the Petrels. This way it was possible to determine which of the burrows had been in use. However the researchers were interested in how the birds behaved during the breeding season.

[5] http://www.ethernut.de/
[6] http://www.greatduckisland.net

To find out, a Wireless Sensor Network was deployed on the island in 2002. This first deployment consisted of 32 Mica motes, equipped with the Mica Weather Board, and was deployed in a 6 hectare area. The weather board contained a light sensor, a humidity sensor, a temperature sensor, a pressure sensor, and a passive infrared sensor[49]. Most of the motes were placed outside the burrows, to measure the weather conditions, while 9 of the motes were placed in the burrows, so that they could measure the conditions close to the nest. All the motes were coated with a thin parylene sealant to protect them against water. The sealant was tested by submerging a running node in water for a week, which did not reveal any problems. The sensors on the Weather Board were not coated, as this would have prevented them from operating correctly. The motes placed outside the burrows was also enclosed in a ventilated acrylic tube, to give them further protection against the weather.

Because of the size of the deployment area, the system was divided into several networks. The sensor nodes were grouped into patch networks, each of which contained a gateway node. This gateway node communicated both with the patch network, and a transmit network. The transmit network transfered all the data from the different patch networks to the base station. The base station was connected to the Internet through a satellite link. The entire network was operated "off-the-grid", i.e. the energy for all parts of the deployment came from either batteries, or solar power.

The major problems during the first deployment were:

- The sensors on some nodes reported out of bounds values. This seemed to be caused by the sensors getting wet.
- Packets from a few nodes in the system never arrived, except when there was packet loss from many of the other nodes in the network.

A close correlation exists between nodes where the sensors reported out of bound values, and nodes which failed later in the experiment[56]. Some of the of the networking problems were likely caused by clock skew. Nodes that experienced excessive clock skew often failed later on in the experiment, so this might have been caused by malfunctioning hardware too[56].

The second GDI deployment was carried out in the summer and autumn of 2003. This deployment was of a rather larger scale than the first, consisting of in total 98 motes[55]. All of these motes used the Mica2Dot platform. 62 were burrow motes, which were deployed in the burrows, while the remaining 36 were weather motes. The burrow motes were equipped with a passive infrared temperature sensor and a humidity/temperature sensor. The weather motes measured temperature, humidity and barometric pressure. The network infrastructure was in large parts similar to the infrastructure used for the first experiment.

Some of the lessons from the second deployment was:

- When designing the packaging for the nodes, one should take the antenna into account. Either by having it integrated on a PCB, or by integrating it into the packaging. That makes it easier to make the packaging waterproof.
- When deploying the node, a externally visible signal, such as turning on a LED, is a help in determining if the node is turned on, and can save time.

- When using batteries with a constant operating voltage, the remaining capacity cannot be estimated by measuring the battery voltage. Therefore other measures are needed, such as energy counters in the application.
- Integrated data logging can help to recover data that is not transmitted off the node properly. However the energy consumption implications needs to be considered thoroughly, as writing to the external flash is a costly operation.

Lessons for Hogthrob

An important lesson to take from the GDI project, is that we should take care when packaging the node, to protect it against the environment. Especially as the pig-pen will be more hostile, with the ammonia in the air, than the out-doors at GDI.

The networking problems does not relate to our project, as we will use Bluetooth for the communication, and the networking problems within the GDI project relates to the radio stack used on the Mica motes.

From the second deployment, we can use the first two lessons, i.e. remembering the antenna when designing packaging, and having a visible indication that the node is turned on. We do not expect to use constant voltage batteries, and we gather more data than we can reasonably store on the node, so the last two lessons do not relate to our deployment.

2.3.2 ZebraNet

The goal of the ZebraNet project is to track the movement of wild animals, specifically zebras[31]. Three revisions of custom sensor nodes have been developed and tested, before settling on a node consisting of a GPS receiver, a long range (approximately 5 km) radio, a 4 Mbit flash and a Texas Instruments MSP430[62]. This node is powered by a 2 Ah Lithium-Ion polymer battery which can power the node for 5 days, and which is recharged by solar cells.

The node obtains a position from the GPS receiver every 8 minutes. This position is stored in the 4 Mbit flash. To conserve RAM on the micro-controller, these measurements are stored in flash directly after they have been obtained. To do this, they make use of the way writes to flash work. Once a page in the flash have been erased, all bits in it is set to high. A write can only change bits from high to low. Since a single measurement only consumes 28 bytes and the flash must be written in 264 byte blocks, the part of the page that should not be altered is filled with high bits, so that it is unaffected by the write.

Once every two hours, the radio is turned on, to search for other nodes within communication range. If one is found, as many samples as possible is transfered to it. To prevent collisions each node has a designated time-slot, in which it is allowed to offload data. The time-slot synchronization is maintained using the time information embedded in the GPS signal. This time-slot system is only possible because of the low number of nodes that are expected to be deployed.

An initial test deployment on 7 zebras have been carried out, at the 100 km^2 Sweetwaters game reserve in central Kenya. The key problem observed during this deployment was that while the range of the radio was specified to 5 miles, and tests had confirmed that it would work up to a range of 1 mile, shorter ranges was experienced in the test deployment. Also the duty cycling of the radio proved much to restrictive.

Lessons for Hogthrob

From ZebraNet we can learn that we should test the range of the radio, in an environment that resembles the deployment environment as close as possible. This will be important for us, as we need to have Bluetooth coverage for the entire pen.

The way the flash memory is used — writing several times to the same page — might also be interesting for our application.

2.3.3 A Macroscope in the Redwoods

UC Berkeley and Intel research have jointly deployed a sensor network to provide information about the environment in a 70 m tall redwood tree. The deployment consists of 33 Mica2Dot motes, all installed in a single redwood tree, and each fitted with sensors for humidity, temperature, photo-synthetically active radiation (PAR), both direct and reflected. The nodes are programmed with TinyDB, and measurements are obtained from all sensors every 5 minutes for 44 days[57]. To provide additional security against lost readings, the nodes store the measurements in the on-board flash.

The nodes are protected by an enclosure, that exposes the direct PAR sensor at the top, while protecting the mote and the rest of the sensors from the environment. However both the temperature and the humidity sensors require a certain amount of airflow to produce accurate readings, so these are exposed at the bottom of the enclosure. A wide cap provides protection against the environment for the bottom of the enclosure.

The data is transfered to a Stargate system,[7] which is connected to a GPRS modem, offloading the results to an off-site database. Not all measurements were offloaded in this way, some were offloaded manually by connecting a laptop computer directly to the Stargate system.

Of the 1.7 million expected data points, only 820.700 were collected, giving a 49% yield. Many of the deployed nodes did not deliver any data to the system. This is attributed mainly to the nodes running out of power during the pre-deployment calibration. The batteries were not replaced prior to deployment, due to fears that the disassembly of all the nodes would increase the likelihood of failure during the experiment. Another problem was that some of the nodes ran out of flash storage during the deployment, as the flash had not been cleared after the pre-deployment calibrations, as this would also require disassembling the nodes. When this happened the nodes would still offload data to the Stargate system.

Lessons for Hogthrob

The major cause of node failure in this experiment, was because the batteries had not been replaced since the pre-deployment calibration. Also, more data would have been available, if the flash on the system had been cleared. To avoid making the same mistakes, we should ensure that the batteries are at the full capacity when we deploy the nodes. We will not be able to store all our measurements on the node, so the lesson about clearing the flash is not important for our deployment.

[7] The Stargate system, is a PC-class single board computer featuring a 400 MHz ARM compatible processor

2.3.4 Deployment of Industrial Sensor Networks

At Intel they are investigating the use of sensor networks for predictive maintenance[33]. Predictive maintenance is a term used to describe a family of technologies that provide information about the health of a piece of industrial equipment.

The focus at Intel have been on vibration analysis. To measure vibration, sensor boards have been developed that can be connected to industrial class vibration sensors. Two different node designs are used for the sensor network. One is the Mica2, which is described in Section 2.1.1, and the other is the Intel Mote. The Intel Mote, is a Bluetooth based sensor node, like the BTnode. However the Intel Mote features a 32-bit ARM micro-controller, operating at 12 MHz[44], making it one of the most powerful sensor network nodes available, with regard to processing power.

The Mica2 and the Intel Mote cannot communicate with each other, because of the different radios. Instead the network is grouped into clusters, which communicate with one or several Stargate gateway nodes. The Stargate gateway nodes are equipped with an IEEE 802.11b wireless networking card, through which they relay the data from the mote, to a root Stargate node.

The sensor network is deployed in two industrial settings: In the central utility building at a semiconductor fabrication plant, and aboard an oil-tanker sailing in the North Sea. The oil-tanker is an especially harsh environment for a sensor network, as it is constructed mainly of metal, limiting the reach of the sensor networks.

In both cases 5 vibration sensors are connected to each node. These are sampled at 19.2 kHz, collecting 3000 samples in a short burst, which results in approximately 6 KiB of data. For the Mica motes, this is more than can be kept in its 4 KiB of memory. To solve this problem, an additional ATMega128 with 64 KiB of external memory was used as a buffer between the Mica mote, and the accelerometers. This was not necessary for the Intel Mote.

How often the motes samples the vibration sensors, depends on the deployment. In the deployment at the central utility building, the 3000 samples are obtained once each hour. For the oil-tanker experiment the nodes are grouped into two deployments, where the starboard deployment waits 18 hours between sampling the sensors, and the center deployment waits 5 hours.

In the central utility building, the Intel Mote with the Bluetooth network provides an average transfer time 10 times lower than that of the Mica motes. This can in part be attributed to the difference in bandwidth between the two radio designs, but the transport protocol also plays a part in the low transfer rate, due to an aggressive throttling algorithm.

For the oil-tanker deployment, results were delivered for at least 80% of the duration of the experiment. The shortest lasting node operated for almost a month, which was more than the estimated 21 days.

Lessons for Hogthrob

From these deployments we can learn that it is possible to use Bluetooth for sensor networks, and that it in some cases is the better choice. The Intel Mote, ends up using less current than the Mica2 motes, due to the higher transfer rate[44]. While this comparison is not completely fair — the protocol used to offload the data, played in favor of the Bluetooth radio — it never the less shows that for high

bandwidth applications Bluetooth is able to compete.

2.3.5 Wired Pigs

Wired Pigs[42] is a study of existing wireless sensor network software and hardware, to evaluate the usability of these when computer science expertise is limited.

The project seeks to measure the body temperature of pigs, to find out if the pigs suffer from temperature induced stress. The body temperature is measured by sensor nodes on the pigs. To show if a change in body temperature is caused by the surroundings, a network of sensors in the stables, measure the humidity, light and temperature.

The sensors on the pigs — called the PiggyBack network — are Mica2Dot motes, with two thermistors attached. These thermistors are surgically implanted into the pig, one just under the skin, and one deeper to measure the core temperature of the pig. The thermistors are attached to the mote with wires, and the mote is placed in a bandage collar around the neck. 4 male pigs were selected for the PiggyBack network.

The network of environmental sensors — called the Environment Network — consists of 4 Mica2Dot motes, measuring the temperature close to the pigs, two Mica2 motes measures the humidity and temperature for the stable in general, and a single Mica2 mote placed outside the stables, measures light and temperature.

The PiggyBack network used TinyDB for the data acquisition, while the Environment Network used the commercial Sensicast Developers Version, a software package that allows the user to create sensor networks in a user friendly fashion. Because the motes use different software packages, the two networks were not able to communicate with each other. Therefore two base stations were required. The base station for the PiggyBack network, was placed so that the maximum communication distance was 5 m. This was required as the individual stalls housing the pigs, were constructed of steel tubes, and thus lowered the communication range a lot. The base station for the Environment Network was placed outside the stables, with the longest communication range required, approximately 10 m.

The performance of the PiggyBack network was in general poor, as the highest rate of obtained sensor readings experienced was 30%. Also some of the surgically implanted thermistors came loose during the experiment, forcing the premature take down of the PiggyBack network. The researchers speculate that the low amount of sensor readings is caused in part by the steel tubes around the pigs, and in part by software problems. On the other hand, the performance of the Environment Network was good, as it experienced over 90% sensor reading reception, during the deployment.

The conclusion of the paper is that the TinyDB software package is difficult to use without a computer science background. It is however much more flexible than the Sensicast Developers Version.

Lessons for Hogthrob

The environment for our network will be quite different, with a large pen housing several sows. This means that we will not have the same problems with the radio range, as there are no metal around our sows. It also means that our solution for mounting the node needs to be more durable, as the sows fight with each other.

The Sensicast software package seems to be primarily oriented around temperature and energy monitoring, and predictive maintenance. Furthermore it does not seem to be compatible with our BTnodes, so this makes it unsuited for Hogthrob. Using TinyDB is not an option either. TinyDB relies on being able to offload the measurements, immediately after they have been obtained. We need a rather high sample rate, meaning that we would not be able to turn off the Bluetooth radio, which again means that we would not be able to power the nodes for the duration of the experiment.

2.4 Previous Farm Experiments

Apart from directly sensor network related experiments, others have performed experiments that resembles what we want do to closely. In this section we will discuss two experiments: A previous heat detection experiment on sows, and a health state monitoring experiment on cows.

2.4.1 Automated Oestrus Detection on Sows

The inspiration to the heat detection approach employed in this project, comes from an experiment that sought to automatically detect the start of the oestrus of sows using either body temperature, vaginal temperature or physical activity[26].

For the body temperature experiment a thermistor was implanted in the ear base of the sows, and connected to a data-logger that measured the temperature every 30 s. This experiment was carried out on 21 individually housed sows.

For the vaginal temperature experiment, a small data-logger was inserted into the sows vagina, sampling the temperature every 16 seconds. This experiment was carried out on 8 individually housed sows.

For the physical activity a small accelerometer[59], was attached to the sow with a neck collar. The acceleration was measured 255 times per second, and a counter was incremented each time the acceleration was greater than 10 $\frac{m}{s^2}$. This experiment was conducted on 4 individually housed sows.

It was found that the both of the temperature based experiments showed significant changes in temperature close to the onset of oestrus. However the activity experiment showed a significant rise in the activity of up to 1000%, at the start of the heat-period. We hope to be able to show a similar change in the activity when sows are housed together.

2.4.2 Health State Monitoring on Cows with Bluetooth

The Danish Institute of Agricultural Sciences[8] have in collaboration with Blip Systems[9] developed a system for monitoring the health state of cows. Nothing about the infrastructure have been published, apart from newspaper articles[34]. The system contains 3 different kinds of nodes. Two to monitor the state of the cow, and one to provide the infrastructure for the experiment.

Of the two nodes monitoring the cows, one node measures if the cow is standing up, or laying down, using a gyroscope. This node only reports when the state changes. The other node measures the pulse and body temperature of the cow, and

[8] http://www.agrsci.dk
[9] http://www.blipsystems.com

offloads these results regularly. This node is also used for determining the position of the cow, using measurements of the transmission strength from the infrastructure network. Both of these nodes are powered by batteries. For the gyroscope node, a normal primary battery is used, as this node does not transmit often. It is expected that the battery can sustain the node for 2 years. The pulse and body temperature nodes have a rechargeable battery that should last for 6 months. The nodes are mounted in a standard casing at the neck of the cow.

The infrastructure network consists of 15 nodes, placed in the ceiling of the 1,200 m^2 stable. These nodes are organized in a mesh-network, where only 4 of the nodes are connected to the data-logging server.

It is hoped that the system will be able to tell the farmer about the health state of each cow, alerting him to potentially sick cows. Also, just as with sows, the activity levels of cows change during the heat-period, so this system could also be used to detect heat-periods of the individual cows.

The goal is to develop the system commercially, with an expected cost of 700 DKK per cow.

2.5 Summary

In this chapter we have provided an overview of some of the available sensor network nodes, and described the TinyOS and BTnut operating systems that are specifically targeted for sensor networks.

We have provided an overview of some of the interesting sensor network deployments that have been carried out, and described the lessons that we can use from these deployments.

The lessons we take with us from the previous deployments are:

- Make sure to protect the node properly against the environment. If possible avoid external antennas, and other stuff that would require holes in the packaging for the nodes.
- Test the range of the communication where the application is going to be deployed, as the range very much depends on environment.
- Make sure that the batteries used when the nodes are deployed, are at their full capacity.

The farm experiments, described in the previous sections, tells us that our application is interesting, as others have tried to do the same. The cow application in particular tells us that there might be a commercial potential in our application. However the price dynamics are not the same when breeding pigs, as they are when breeding cows. So the success of an integrated heat detection chip for sows, is entirely dependent on the price being in the 10 DKK range.

A trend in these deployments is that most of them are data collecting experiments, that allows researchers from other fields to get otherwise unavailable data. While this is a worthy cause, it does not provide a good business case for sensor networks. However deployments such as the Industrial Sensor Network deployment (see Section 2.3.4), could be the first steps towards this. Likewise the future stages of the Hogthrob project could provide such a business case.

Chapter 3
The First Hogthrob Experiment

In this chapter we will describe the goals and approach of the first Hogthrob experiment. The main goal is to collect data to construct a model, which can be used to detect heat from the activity patterns of the sows. We will also describe the different kinds of ground truth we are going to collect during the experiment.

3.1 Goal

The overall goal of the Hogthrob project is develop a method whereby sensor network technology can be used to track sows, and detect the start of their heat-period.

In the first phase of the project we need to establish a model for detecting the start of the heat-period for the sows. Others have observed a 1000% increase in the activity for individually housed sows, during their heat-period[26]. However, in Denmark the current law prohibits individual housing of sows, so we need to re-validate this result for sows housed together in large pens. The sows establish themselves in a hierarchy, and a sow's position in this hierarchy might affect how clearly she displays signs of heat. It is therefore not given that the previous results can be re-used.

To re-validate the previous experiment we want to monitor the activity of the sows, with the purpose of establishing a model that can be used to determine when sows are in heat. Establishing this model is beyond the scope of this thesis, but will instead be carried out at KVL. To make it easier to establish this model we wish to gather as much data as possible. To validate our measurements we also need to collect some absolute form of truth — our ground truth. This ground truth will — amongst other things — provide information about the heat-periods of the sows.

The experiment will be conducted at Askelygaard, outside Roskilde, and will run from the 21st of February 2005, to the 21st of March 2005. The nodes will be installed the 28th February 2005, i.e. on day 7 of the experiment, and they will have to collect data for at least 20 days. Five sows will be selected for the experiment.

3.2 Method

For the experiment we will use the BTnode revision 2.2 developed by ETH Zürich, to gather the data. We have chosen these nodes, as we already have previous experience in using these, but also because we have enough of these nodes to carry out the experiment. This choice helps keeping the cost of the project low.

If cost was not an issue, it would make sense to choose another platform, as the radio on the BTnode, is very energy consuming, has a high startup time, and

high discovery and connection times[32]. While much of this is true of Bluetooth in general, the ROK 101 007 modules used on our BTnodes were the first commercially available Bluetooth modules from Ericsson[32]. The energy consumption and startup time have been improved since then, making Bluetooth a better choice in sensor networks[44], but it is still not as flexible as the simpler radio systems used on other sensor nodes.

In the previous heat detection experiment a device that measured the acceleration was attached to a collar around the neck of the sow[26]. The acceleration was sampled with a rate of 255 Hz. Every time the acceleration was above 10 $\frac{m}{s^2}$, one unit was added to a counter. It is not clear from the publication how often this counter was reset, nor when the data from this counter was offloaded.

As we need to re-validate the findings of this previous experiment, we choose to approach the problem in the same way: We will use accelerometers to detect the activity of the sows, and place these in a collar around the neck of the sow. However, we have decided against aggregating the measurements as described above, as this will discard a lot of data, that might be useful when designing an algorithm to detect the heat-period. Instead we want to sample at 4 Hz, and keep all of the samples. The reasoning behind this sample rate, is mainly to lower the energy consumption. At a sample rate of 255 Hz, we would gather an enormous amount of data which needs to be offloaded from the node, all of which will cause a higher energy consumption. Also it has been shown that frequency of the acceleration in the human upper body while walking, is in the range of 0.8 – 5 Hz[12]. As the sows does not move very fast most of the time, we decided that a sample rate of 4 Hz should be sufficient.

3.2.1 Ground Truth

Besides the acceleration data, we want to collect several other data sources to establish a ground truth. First of all we need to know when the heat-period occurs, but we would also like to be able to determine the cause of specific events in the gathered acceleration data.

To find the heat-period, we use 3 different manual heat detection methods, which together will provide a very good accuracy of when the heat-period occurs. The 3 methods are:

- Back Pressure Test (BPT)
- Vulva Reddening Score (VRS)
- Rectal Temperature (RT)

The BPT and VRS methods will be described in detail in the following sections. For the rectal temperature test, a slight change in temperature indicates that the sow is in heat.

The manual detection methods is conducted from day 21 of the experiment (the 13th of March) at 07:00, 14:00 and 21:00, and until one day after the sow stops showing the standing reflex.[1]

Another source of ground truth is the boar pen visit. A separate pen is located so that the sows have all but physical access to a boar. When they approach the boar, their RFID ear tag is read, and stored together with the time of their visit.

[1] The standing reflex is when the sow stands stiffly, ready for the boar to mount her. Sows show this behavior when they are close to, or in heat.

An increase in the number of times a sow approaches the boar during the day, is a good indication of heat. The results from this test will be used to further improve the accuracy of the heat detection.

To be able to explain specific patters in the data, we also wish to monitor the pen using 4 cameras. These cameras will cover the parts of the pen, where the sows are active, such as the feeding station etc. The camera installation will be described in detail in Section 8.3.

3.2.2 Back Pressure Test

The back pressure test (BPT)[15] is a simple test, and also the one the farmer uses to determine if a sow is in heat. The test result is a score between 1 and 6, and the scores are given as follows:

1. The sow flees when touched, or within 5 seconds of being touched.
2. The sow allows the person performing the test to press the knee against her side, and to press her on the back. She does not allow the tester to sit on her back.
3. The sow shows the standing reflex when the tester sits on her back, but flees during the test.
4. The sow shows the standing reflex without the ears erected, allows the tester to sits on her back, but vocalizes.
5. The sow shows the standing reflex without the ears erected, allows the tester to sit on her back, and does not vocalize.
6. The sow show the standing reflex with the ears erected.

When the score is in the range from 4 to 6, the sow is considered to be in heat.

3.2.3 Vulva Reddening Score

For the vulva reddening score (VRS)[15] the color of the vulva is used to detect heat. The score is divided into 4 categories:

0. Pale red
1. Pink
2. Red
3. Dark red

The colors are matched against previously selected pictures of other sows.

3.3 Summary

In this chapter, we have established the goal of the first Hogthrob experiment, i.e. to gather data about the activity of sows, so that a heat detection model can be established. We have decided that the node platform used to monitor the activity will be the BTnode revision 2.2, and the sensors that measure the activity will be accelerometers. We have also described the 5 sources of ground truth will be collected during the experiment:

- Back Pressure Test

- Vulva Reddening Score
- Rectal Temperature
- Boar Pen Visit
- 4 cameras that cover the parts of the stables where the sows are active.

In the next chapter we will evaluate different kinds of accelerometers, and in general prepare the node hardware for deployment in the stables.

Chapter 4

Node Hardware

In this chapter we will explore the different kinds of accelerometers that are available, and how these can be connected to the node. We will also explore how to modify the node, to bring down its energy consumption.

For the accelerometers we will have to choose which accelerometers to use, and we will need to design and produce a sensor board containing these.

To make the node ready for deployment, we will have to solder new components onto some of the nodes, and replace other components. The goal is to make all the nodes similar, and to minimize their energy consumption as much as possible.

4.1 Sensor Board

In this section we will discuss the requirements of the sensor board needed for the experiments. We will select the sensors we wish to have on the board, and select the specific components to be used. To select the components, we look at following characteristics:

- The resolution and range of the sensor.
- The availability of the sensor.
- The energy consumption of the sensor.
- The voltage required for the sensor to function.
- The interface that allows the sensor to communicate with the node.

From these key characteristics it should be possible to decide which components to use on the sensor board.

4.1.1 Accelerometers

To measure the activity of the sows, we choose to use an accelerometer. An accelerometer measures the acceleration, and are used in a multitude of applications. In cars for the airbag system, as vibration sensors in washing machines, etc. Several types of accelerometers are available. Some have mechanical acceleration detection, some use piezo-electric materials, but recently integrated circuits (IC) containing accelerometers are being produced as well. We will focus on these, as they provide the smallest form factor, and because they will be easier to integrate with the node.

When creating an accelerometer in an IC, a process called Micro Electro-Mechanical System (or MEMS) is usually employed[45]. In a common design the sensing part of the accelerometer is a block of silicon that is suspended inside the IC so that

it moves on a spring, when the IC is exposed to acceleration. Some "fingers" are attached to the block that moves, and in conjunction with the rest of the IC, these create a capacitor. The capacitance of this capacitor changes when the "fingers" moves, and this change is measured and converted inside the IC.

To make it simple to connect the accelerometer to the BTnode, we will focus on accelerometers that can be powered by 3.3 V, as this is the voltage provided by the voltage regulator on the node.

In the following sections, we will discuss the most important of the characteristics for the accelerometers, i.e. the range, the interface, the energy consumption and startup time.

Range Options

The most important parameter of the accelerometer for our experiment, is the range of acceleration it can measure. The acceleration experienced when a human walk are in the range of -2 – 2 g [12, 18] when measured at the tibia. We assume that normal movements of a sow will be in the same range, as we measure on the head, where the accelerations are usually lower. However when the sows are fighting or startled we expect a higher acceleration. So if we get an accelerometer with a range of approximately -5 – 5 g, there should be few cases where the acceleration would be out of range for the accelerometer.

Interface Options

The interface determines how to read the acceleration measured by the accelerometer. The most common types of accelerometers are analog, where the voltage output of the accelerometer is proportional to the acceleration. This output can then be converted to a digital value, either by an external Analog-to-Digital Converter (ADC), or by an ADC included in the MCU.

A few accelerometers have a Pulse Width Modulated (PWM) output. In this case, the output from the accelerometer is digital. The MCU must measure the time from a rising flank in the digital signal, to a falling flank. This time, in relation to the time between two rising flanks can then be used to calculate the acceleration.

The accuracy of both of these interfaces are determined by the accelerometer, but also by the circuits that convert the signal to a digital reading. The accuracy of an analog accelerometer is limited by how good the ADC in question is, and the resolution of the timer in the MCU will limit the accuracy of a PWM based accelerometer.

The last option is to have a digital interface, such as Serial Peripheral Interface (SPI), or Inter-IC Bus (I^2C). If the MCU does not have support circuits for these interfaces, they can be emulated at the software level, through the use of general purpose I/O pins (GPIO). A digital interface has the advantage that the accuracy is not limited by parameters outside of the IC.

Energy Consumption and Startup Time

The energy consumption of the accelerometers is also important. The energy consumption when a sample is obtained is of course important, but in order to obtain the lowest possible energy consumption we must duty cycle the accelerometer by powering it down, or putting it into a sleep-mode, if one such is available. When the

accelerometer is turned on we must wait for a short period, before reading samples from it. This is because the accelerometer needs to initialize, and charge its filter capacitors. Therefore the amount of time we can leave the accelerometer powered off, depends on the amount of time it needs to charge its filter capacitors. An optimal accelerometer for us, has as short a startup time as possible.

4.1.2 Accelerometers from Analog Devices

Analog Devices produces MEMS accelerometers with analog output or with PWM output. In the summer of 2004, when we selected the accelerometers, the only accelerometer with a range of 5 g, the ADXL320[4], was not yet released. The closest was either a 2 g (the ADXL202[2]) or a 10 g (the ADXL210[3]) accelerometer. However Analog Devices was kind enough to provide us with pre-release samples of the ADXL320. The ADXL320 is has two analog outputs, one for the X-axis and one for the Y-axis of the accelerometer[4].

4.1.3 Accelerometers from Freescale Semiconductors

Freescale Semiconductors (previously Motorola) produces analog accelerometers, which resemble the ones from Analog Devices. The only ones that were available in the summer of 2004, had a range of 1.5 g (the MMA6260Q[23]) or 10 g (the MMA6231Q[22]). Both of these are dual-axis accelerometers.

4.1.4 Accelerometers from STMicroelectronics

STMicroelectronics' line of accelerometers with analog output resemble the offerings from both Freescale and Analog Devices. However all the accelerometers from STMicroelectronics have an electronically selectable range, which can be set to either ± 2 g or ± 6 g. The accelerometers come in 2-axis or 3-axis versions. At the time, STMicroelectronics was the only manufacturer we found, which had a 3-axis accelerometer available.

The LIS3L02DS is a 3-axis 2 g/6 g accelerometer which can communicate with the MCU either through the I^2C bus, or through the SPI bus[54]. Both these busses are supported by the ATMega MCU on the BTnode. Apart from having 3 axes, this accelerometer has other interesting features: It can be programmed to issue an interrupt, whenever the acceleration of one of the axes are above or below a certain threshold, and it also has a power down mode. The LIS3L02AS4 is basically the same accelerometer as the LIS3L02DS, but with an analog interface[53].

4.1.5 Comparison of the different Accelerometers

In Table 4.1 the accelerometers described above are listed together with their most important characteristics, taken from the respective datasheets.

As can be seen from this table, the specifications does not differ greatly. We choose to use both a 2-axis and a 3-axis accelerometer. The 2-axis because these are the most common parts, and therefore the cheapest, and the 3-axis in order to have an accelerometer that can measure all three axes.

For the 2-axis accelerometer we chose to use the ADXL320, because it matched the range we wanted to measure. For the 3-axis we chose the LIS3L02DS, because it was digital and because it has 12-bit precision, where the ADC in the BTnode only has 10-bit precision. The LIS3L02DS has a high startup time, and if we bandwidth

Table 4.1 – *Comparison of different accelerometers*

Product Code	Axes	Range	Interface	Energy consumption	Startup time
ADXL202[2]	X, Y	$\pm 2\ g$	PWM	0.6 may	See Note
ADXL210[3]	X, Y	$\pm 10\ g$	PWM	0.6 mA	
ADXL320[4]	X, Y	$\pm 5\ g$	Analog	0.48 mA	
MMA6260Q[23]	X, Y	$\pm 1.5\ g$	Analog	1.2 mA	14 ms
MMA6231Q[22]	X, Y	$\pm 10\ g$	Analog	1.2 mA	2 ms
LIS3L02DS[54]	X, Y, Z	$\pm 2\ g/\pm 6\ g$	Digital	1 mA	50 ms
LIS3L02AS4[53]	X, Y, Z	$\pm 2\ g/\pm 6\ g$	Analog	0.85 mA	See Note

Note: For some of the accelerometers, the startup time depends on the size of an attached capacitor. For the ADXL202 and ADXL210 the formula to calculate the startup time is $160\ ms/\mu F \times C_{filt} + 0.3\ ms$*, where* C_{filt} *is the value of the capacitor. For the ADXL320 the formula is* $160\ ms/\mu F \times C_{filt} + 4\ ms$*. For the LIS3L02AS4 the startup time is* $550\ ms/\mu F \times Cload + 0.3\ ms$*.*

the ADXL320 to 10 Hz, it will also have a high startup time. However their power consumption is low enough that it should not matter.

4.1.6 Other Possible Sensors

Since we are going to create a sensor board specific to our application, we might as well consider if we wish to include more sensors, than just the accelerometers. As the node and sensor board needs to be very well protected from manure, water and the like, we do not wish to have any sensors that needs to be exposed to the surroundings. This requirement narrows the range of relevant sensors.

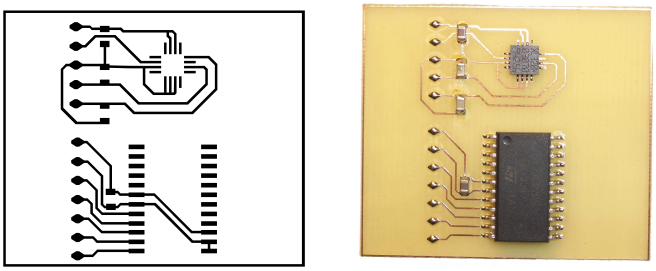
A sensor that still would be relevant is a humidity sensor. A humidity sensor could be used to detect if the node electronics is exposed to water. If this happens, the node could send out a warning that it most likely would have to be replaced soon.

Many different humidity sensors exist, but most of them have an analog interface. To measure the humidity, power must be pulsed through them while the resistance is measured. The measured resistance of the humidity sensor is proportional to the relative humidity of the air around the sensor. Such a sensor was used on the Mica Weather Board v1.0 which was used in the first deployment of Great Duck Island project[49]. To interface the sensor to the MCU, a lot of control electronics had to be used. Also for the Great Duck Island project, they experienced a lot of problems with this kind of sensor, ranging from crashing the node to draining the battery. This was most likely caused by the sensor getting wet, but still the external electronics required makes this solution impractical for our project.

Another solution is to use a digital humidity sensor such as the SHT11 from Sensirion, which is the humidity sensor used for the Mica Weather Board v1.5[49]. This sensor provides readout of the sensor value through an I^2C interface, making it much more suitable for integration with a MCU.

As the cost of this sensor is around 150 DKK, and its use is limited, we decided against using it.

Figure 4.1 – *The accelerometer sensor board*



(a) The PCB layout for the sensor board

(b) The actual sensor board, with the two accelerometers soldered on

4.1.7 Manufacturing the Sensor Board

We initially expected to use a simple bread board for the sensor board with the two accelerometers soldered onto. However the ADXL320 is packaged in a 4×4 mm lead frame chip scale package (LFCSP[1]) with 16 solder pads. It is not possible to solder these chips using a soldering iron. The LIS3L02DS is packaged in a 24-pin SMD package, which can be soldered by hand.

The Technical University of Denmark (DTU) kindly agreed to manufacture both print boards, and to solder the accelerometers onto these. The resulting PCB layout and sensor board can be seen in Figure 4.1.

The way the two accelerometers are placed, the X-axis of the ADXL320 points upwards in the picture, and the Y-axis points to the left. For the LIS3L02DS, the X-axis points to the right, while the Y-axis points upwards. To make the two accelerometers easier to compare, we switch the wires for the X and Y-axis on the ADXL320, so that the X-axis on the ADXL320 corresponds to the X-axis on the LIS3L02DS, but with the sign switched, and the Y-axis will correspond directly to each other.

For the ADXL320 we needed to choose the value of the two filter capacitors. The filter capacitors filter out noise from the accelerometer readings, and thus selects the bandwidth of the accelerometer. Since we needed to obtain samples from the accelerometers with a sample rate of 4 Hz, we choose capacitors with a value of 0.45 μF, which gives us a bandwidth of 10 Hz.

We decided to make connections on the sensor board for the self test pin on both of the accelerometers, since this could aid us when testing the boards. For the LIS3L02DS we also decided to leave connections for all the data-pins, so that we did not have to choose which of the interfaces to use, when manufacturing the board.

We connected the power supply of each of the accelerometers to a spare general purpose I/O pin on the ATMega128. This way we could turn each accelerometer off completely. The current that can be sourced from a single pin of the ATMega128 micro-controller is well within what is required to drive the accelerometers[4, 8, 54].

4.2 BTnode Modifications

To make the BTnodes ready for deployment, we had to make some changes to them. These changes were made to ensure that all nodes behaved the same way, and to lower their energy consumption as much as possible.

All of the necessary changes to the nodes were carried out by DTU.

4.2.1 Reducing Energy Consumption on the Nodes

During the initial phase of the project, we conducted some simple experiments, in order to determine if the goal of having the node run for 20 days on battery power, was obtainable.

This initial experiment was conducted by connecting the BTnode to a 4.5V power supply, and an ampere-meter. A modified `Blink` application was uploaded to the node, and the current consumption was measured. The `Blink` application simply toggles one of the LEDs on the node, and our modification was to have a longer interval between the toggles, to allow the ampere-meter to settle in between.

The current consumption measured from this experiment was in the range 12 – 13 mA. If we assume that we are going to have 4 rechargeable AA batteries on the node with a capacity of 2000 mAh the batteries will last for about 7 days, if we keep the node idle all the time. As the experiment should last 20 days, we need to cut the energy consumption down to less than a third, to have energy for both sampling the accelerometers, and offloading the results.

The first problem we located was that the port of TinyOS to the BTnode only entered the idle sleep-mode[36]. Since the `Blink` application only uses the timer, the ideal sleep-mode for the node would be power-save[8]. Fixing this problem in the TinyOS port (see Section 5.2 for the implementation details) lowered the current consumption to 7.3 mA. Disconnecting the BTnode from the programmer lowered the current consumption further to 5.5 mA.

According to the datasheet the ATmega128 should only use about 9 μA[8] when in power-save mode at 3.3 V. The reason we looked at the current consumption at 3.3 V is because this is the voltage supplied by the voltage-regulator. When the MCU draws 9 μA, the voltage regulator powering the node uses around 0.11 mA, because of the ground current in the voltage regulator[46]. So we should be able to lower the current consumption further.

Martin Leopold discovered that if the 0 Ω resistor providing power to the external memory was removed, the current consumption would drop. Removing this resistor will disable the external memory, leaving the node with only the 4 KB internal memory. However, the current consumption dropped to 0.5 mA so this was a necessary tradeoff.[1] With this energy consumption the node should be able to function for about 160 days. This estimate will drop when the node needs to sample the accelerometers and use the Bluetooth radio.

While the node is only using 0.5 mA with the listed modifications, it still does not compare too well with the 0.11 mA we assume it would use. However there are other components that consume energy:

- There is a second voltage regulator used to power the Bluetooth module, how-

[1] We discovered after the deployment that if the external memory was initialized properly, the current consumption would go down almost as much as when removing the 0 Ω resistor.

ever it is turned off in this experiment, and thus should draw only 1.5 μA.

- There is also a voltage divider, used for the battery charge indicator. This voltage divider is made of a 2.7 kΩ and a 10 kΩ resistor, and will therefore use 0.26 mA at 3.3 V.

With these components adding to the energy consumption, it does not seem unreasonable that the node uses 0.5 mA when in power-save mode.

So to sum up, the only hardware change necessary, was to disconnect the 0 Ω resistor to the memory module. With this modification in place, we could not reasonably expect the energy consumption to be any lower.

4.2.2 Voltage Regulator for the Bluetooth Module

On some of our nodes, there was a second voltage regulator, which controls power to the Bluetooth module. The Bluetooth module itself has a pin called ON, which can be used to turn the module on and off. According to early versions of the datasheet, it should be enough to pull this pin to a logical low, to turn off the module.

However the designers of the BTnode noticed that the module still consumed power, even when it should be turned off. Therefore they included the second voltage regulator, to ensure that the module could be turned off completely. If one looks at later versions of the datasheet, they state that the pin called VCC_{io} should be pulled low together with the ON pin. But since the VCC_{io} pin is connected directly to the power source for the Bluetooth module, we will need the extra voltage regulator in order to turn off the module properly.

4.2.3 Battery Charge Indicator

As previously mentioned the BTnode features a battery charge indicator. This indicator is a voltage divider that is attached to an ADC pin on the MCU. The voltage divider is made of a 2.7 kΩ and a 10 kΩ resistor. Since the ADC on the node can measure up to 3.3 V, the maximum voltage that can be measured by the voltage divider is 4.2 V. If we use 4 cells to power the node the initial voltage from rechargeable batteries is going to be around 5.7 V, and when the battery voltage is down to 4.2 V, they are close to being completely depleted. Because of this we want to replace the voltage divider. If we replace the 2.7 kΩ resistor, with a 10 kΩ — so the voltage divider halves the voltage — we can measure up to 6.6 V.

It is not possible to turn off the voltage divider, so this change will also give us a small benefit in the energy consumption. The original voltage divider was using $\frac{3.3\text{ V}}{12.7\text{ k}\Omega} = 0.260$ mA, but if the resistors are both 10 kΩ, the voltage divider will only consume 0.165 mA.

4.3 Summary

In this chapter we have selected two accelerometers for our custom sensor board, and manufactured sensor boards for use in the experiment. The accelerometers were chosen primarily because they match the range of accelerations we expect to encounter in the experiment.

We have also lowered the current consumption of the BTnode, from initial measurements of 13 mA, down to 0.5 mA. The high current consumption was due

in part to inefficient sleep-mode usage in the TinyOS port to the BTnode, and in part due to the external memory, which we disabled.

Lastly we have discussed some changes which are needed to make the nodes ready for deployment. We have ensured that all nodes are equipped with a separate voltage regulator so that the Bluetooth module can be powered off completely, and we have changed the battery charge indicator to support a wider range of battery options.

In the next chapter we will look at the different software parts we will need to support the application.

Chapter 5

Low Level Software

TinyOS have been ported to the BTnode by Martin Leopold[36]. In this chapter we will describe the changes and additional components that is needed to use this port for our application. We need the following to support the application:

- Components to obtain measurements from the two accelerometers.
- A method to control the power-management features of the ATMega128, as described in Section 4.2.1.
- Components to control power to the Bluetooth module, and communicate with a base station PC.
- A way to store data on the node, so that we can duty cycle the Bluetooth radio.

In the following sections we will address each of these points in turn.

5.1 Accessing the Accelerometers

To allow the application to obtain readings from the sensors, we need to create an interface to access the sensors. But first we need to choose how to interface with the sensors.

5.1.1 The ADXL320

The ADXL320 outputs the acceleration as a voltage. Therefore we connected the output pins for the X and Y-axis to two of the ADC pins on the MCU. So to obtain a reading from the accelerometer, we need to query the ADC circuit in the MCU. A component that does exactly this is already present in TinyOS.

However this component is not very well documented, and very complicated. It requires the user to create an interface for each axis of the accelerometer, and during the initial experiments with it, it seemed to return wrong results. Given the complexity of the code, we decided to implement a simpler solution, which is easier to debug and customized for our needs.

The interface to our component is as simple as possible, and can be seen in Figure 5.1. Apart from the `TwoAxisAccel` interface the `ADXLAccelM` module also implements the `init`, `start` and `stop` commands from the `StdControl` interface, which is used to turn the accelerometer on and off. It also ensures that the startup time requirements are honored. Because we use 0.45 μF filter capacitors, the startup time is 80 ms.

Figure 5.1 – *The interface to the 2-axis accelerometer*

```
interface TwoAxisAccel {
  command result_t getData();
  async event result_t dataReady(uint16_t xaxis,
                                 uint16_t yaxis);
}
```

5.1.2 The LIS3L02DS

The LIS3L02DS can be connected to the MCU in two different ways. Either through the I^2C interface (a subset of what Atmel calls TWI) or through the SPI interface. Again there are components for both of these interfaces included with TinyOS.

We decided to use the TWI interface for the communication, as the SPI interface is used to communicate with the radio on the Hogthrob V0 platform. Since we will want to connect the accelerometer to this platform later in the Hogthrob project, using the TWI interface was the obvious choice. However the TWI interface caused serious reliability issues[1], causing us to switch to the SPI interface instead.

Again a component for using the SPI interface, already existed within the TinyOS project. The component in this case was very specific, and would have to be modified somewhat in order to work with the SPI interface on the accelerometer. When one communicates with a SPI device, the $\overline{SS}$ line (Slave Select) to the device one wishes to communicate with must be set to a logical zero. This signals to the device that we wish to communicate with it. Usually the device listens for as long as the $\overline{SS}$ line is low, but this accelerometer, only listens long enough to receive one command, and send back the result. This did not blend well with the TinyOS component.

Because of this, we opted to develop our own component, which suited our needs. The interface for the digital accelerometer is kept just as simple as for the analog accelerometer, and can be seen in Figure 5.2. Again the `STM3DigiSPIM` module also provides the `StdControl` interface, allowing the application to control the power to the accelerometer. And this interface also ensures that the accelerometer is not accessed before the startup time of 50 ms have passed. We selected the 2 *g* range on this accelerometer, so that we will measure both with a 2 *g* range and a 5 *g* range.

5.2 Power Management

As described in section 4.2.1, there is no support for power management in the BTnode port of TinyOS, and this is required for the node to survive the 20 days of the experiment.

The ATMega128 has 6 different sleep-modes. Of these only 3 are interesting for our purpose. These are idle, ADC Noise Reduction, and power-save. The others

[1] Usually the problems would start with the Bluetooth communication speed dropping to 2 KiB/s, where it normally was around 20 KiB/s. The node would continue with its tasks, but after a while it would get stuck entirely. When first encountered these symptoms started happening after approximately one hour. After changing parts of the TWI interface code it would take approximately a day for the problem to appear. We did not have a JTAG interface (which is used to perform in-system debugging), so it was impossible to find the cause of these problems.

Figure 5.2 – *The interface to the 3-axis accelerometer*

```
interface ThreeAxisAccel {
  command result_t getData();
  async event result_t dataReady(int16_t xaxis,
                                 int16_t yaxis,
                                 int16_t zaxis);
}
```

are not interesting either because they disable the timer (which we need in order to wake up), or because they are used when one wishes to keep the wakeup time low which is not necessary for our purposes. For more information see the datasheet for the ATMega128[8].

Idle mode is used when any of the integrated I/O interfaces are in use, e.g. when the UART is communicating, or when the SPI or I^2C interface is in use. It only stops the clocks that drive the CPU and the program memory (i.e. the flash).

ADC Noise Reduction is used when the node is performing an analog to digital conversion, to reduce the noise from other parts of the MCU. Like the idle mode, it stops the clocks that drive the CPU and the flash, but this mode also stops the clock driving the I/O interfaces.

Power-save mode is used when the MCU should wait on the external timer. In this mode, the only functioning parts of the MCU is Timer0 and the external interrupts. Timer0 is the timer that is driven by the external oscillator, and the one used for the central timing component — called `TimerC` — in TinyOS. This mode should be entered when the MCU has no work left to do, and when it does not need to communicate with external devices. This mode stops all clocks, except the one driven by the external oscillator.

In the following sections we will describe how we have implemented the power management, and how it is implemented for the other ATmega128 based ports, as our approach is different than the one used by the other ports.

5.2.1 Implementing Power Management

The choice of sleep-mode depends on the currently active I/O interfaces on the MCU. Therefore we choose an approach that resembles having a counting semaphore for each available state.

When a component turns on an I/O interface, and needs to make sure that the node enters the idle mode, it calls the `TOSH_ENTER_IDLE_MODE` macro. This macro increments a counter, which remains non-zero for as long as one of the components in the system needs idle mode. When the component no longer requires idle mode `TOSH_LEAVE_IDLE_MODE` is called. A similar counter is implemented for the ADC Noise Reduction mode.

When TinyOS puts the node to sleep, it determines the best sleep-mode based on the values of the two counters, and puts the node to sleep in this mode. Interrupts needs to be enabled when the sleep-mode is entered, otherwise the node will never wake up from sleep.

The selection of the correct sleep-mode, requires several instructions, and ends up setting the `MCUCR` register of the node. To make sure that the correct sleep-mode

is selected this needs to be done with interrupts disabled. Once this register is set, the interrupts are enabled again, and the `sleep` instruction is executed. However this leaves a small window where an interrupt could change the sleep-mode needed by the application, making the node enter the wrong sleep-mode, which in the worst case could cause the node to lock up.

The `MCUCR` register has a bit which controls if the `sleep` instruction of the MCU will be executed or not. We can use this bit prevent the race. When one of the `TOSH_LEAVE` or `TOSH_ENTER` macros are executed, we disable the execution of the `sleep` instruction. This means that if one of these macros are executed in the small window between selecting the correct sleep-mode, and issuing the `sleep` instruction, the `sleep` instruction will not be executed. The TinyOS scheduler will then end up calling the `sleep` function again, and then the correct sleep-mode will be entered.

5.2.2 Power Management on the Mica Motes

The power management implementation used on the Mica motes does not use counting semaphores. Instead the power management adjustment is done in a component called `HPLPowerManagement`. This component provides a command called `adjustPower`, which uses the MCU registers to determine which interfaces are currently active. From this it determines the correct sleep-mode and sets it.

The race in our solution is also avoided by the Mica implementation. When `adjustPower` is called it posts a task to perform the actual power management adjustment. If an interrupt fires while this adjustment is in progress, which also calls `adjustPower`, TinyOS will not put the node to sleep, but instead execute the new `adjustPower` task after the current adjustment, and thus select the correct sleep-mode.

This method would also work for us. However, the port of TinyOS to the BTnode did not include the `HPLPowerManagement` component, and we did not become aware of its existence until after the experiment was finished.

5.3 TinyBT

TinyBT[36] is an implementation of a very simple Bluetooth-like stack.[2] The Bluetooth interface is separated into several layers.[3] The lowest layer which we can access from the node is the HCI (Host Controller Interface). As can be seen from Figure 5.3, the HCI layer provides an interface to the protocols ACL (Asynchronous Connection-Less link) and SCO (Synchronous Connection-Oriented link). The ACL protocol is a point-to-multipoint link, and even though it is called connection-less, a node still has to initiate a connection to another node to send data. The SCO protocol is a point-to-point link, well suited for voice transfer.

ACL is the underlying protocol used by the L2CAP protocol (which is implemented purely in software), which provides multiplexing of data connections, while the SCO protocol is used to provide streaming support for voice data. The L2CAP protocol is normally the lowest layer exposed by the OS. However TinyBT only

[2] As TinyBT does not implement the full Bluetooth stack, and as it is not recognized by the Bluetooth SIG, it cannot claim to support Bluetooth. We will however refer to it as Bluetooth for the sake of simplicity.

[3] We will not describe the Bluetooth protocol in detail, but instead refer the reader to books on the subject[13, 43], and the Bluetooth specification[11].

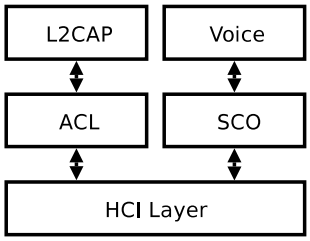


Figure 5.3 – *Part of the Bluetooth protocol stack*

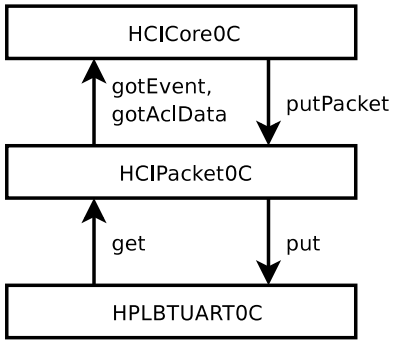


Figure 5.4 – *Components of the TinyBT stack*

provides access to the ACL layer.

Working at the ACL level represents a challenge when communicating with a PC. As already mentioned, the lowest point of entry for a standard compliant Bluetooth stack is through another protocol on top of the ACL protocol. Therefore many of the Bluetooth libraries does not support interfacing directly with the ACL protocol. The Bluez stack (which is the default Bluetooth stack in Linux), can be made to work reliably with ACL communications provided that one disables the software implemented ACL-based protocols in the operating system.

5.3.1 Problems in the original TinyBT Stack

At the beginning of this project the TinyBT stack was not very well tested. It was written as a proof-of-concept and it had only been used to communicate between two BTnodes, for short periods of time. No support for node to PC communication was provided. This is however a requirement for this project.

In the TinyBT stack it is possible to choose the baud-rate used between the Bluetooth module and the MCU. At the start of the project, this was set to 57.6 kbps, resulting in a low bandwidth, but the TinyBT code worked reliably. When the communication speed was raised to 460.8 kbps, which is the highest data-rate supported by the Bluetooth module, problems started showing. The Bluetooth stack started to report errors about receiving unknown commands etc. When such an error was encountered, the stack usually ended up in an unusable state.

To explain the cause of problem, we need to take a look at how the components that make up the TinyBT stack are connected to each other. As can be seen from Figure 5.4, the component that communicates with the Bluetooth module is called `HPLBTUART0C`. This component can communicate with the Bluetooth module (actually the UART) one byte at a time. Once it receives a byte, it signals the `get` event. This event is received by the `HCIPacket0C` component, which splits the data-stream from the UART into individual HCI packets. When it detects that an entire packet have been received, this packet is delivered to the `HCICore0C` component, through one of the `gotEvent` or `gotAclData` signals.

The TinyBT code did not work very well when the communication speed was raised, because a single byte buffer was used internally in the `HCIPacket0C` component. When a byte was received from the UART, the it was stored in a buffer,

and the `data_ready_task` was scheduled to run. This task copied the byte to the receive buffer, and checked if a whole packet was ready. However if another byte was received, before the `data_ready_task` had been run, the previously received byte would be overwritten.

We fixed this problem by storing the data directly in the packet receive buffer, instead of storing it in a intermediate buffer. This required that we check if an entire packet have been received while still in interrupt context, but this check is simple enough to be feasible.

Solving this problem, gave rise to a new problem. Instead of silently overwriting bytes, entire packets were being dropped instead. This was bad, as some of the packets received from the Bluetooth module are event packets, i.e. information packets concerning state changes and other information, which is needed to correctly interface with the module. If such a packet is lost, the stack can end up in a non-working state, which causes the application to fail.

The packets were dropped because of another queue that only holds one item. When `HCICore0C` delivers a packet to the application, it does so through an event which is not in interrupt context, but in task context. The event is delivered in task context because it is easier for the application programmer to handle such events, as they are serialized by the system, eliminating many race conditions. To exit interrupt context `HCICore0C` has to post a task, which then delivers the received packet to the application. When the application handles the event, it must return a free packet, which `HCICore0C` and `HCIPacket0C` can receive the next packet in. But while the application handles this packet, the `HCICore0C` and `HCIPacket0C` components have no place to store any new packet that is being received. This means that they will have to drop data until the application exits the event handler.

The solution to this problem is to implement a buffer-manager[40] which the lower layers can ask for new buffers when they need them.

With these two changes the Bluetooth module works consistently, even when the transfer rate is 460.8 kbps.

5.3.2 Duty Cycling the Bluetooth Module

Another problem with the TinyBT stack, was that it did not allow the application to turn off the Bluetooth module, once it had been powered on. This was a major problem, as the power consumption of the Bluetooth module when idle is 30 mW[38]. Therefore we need to be able to turn off the Bluetooth module, when we are not using it to offload data.

To turn off the Bluetooth module, we need to be able to reset the stack so that it can initialize the Bluetooth module more than once. Because the version of TinyBT we started with used a buffer-swapping technique, it allocated 3 – 4 internal buffers. These buffers could be returned to the application, and end up as a part of the applications buffer pool. Therefore it was not possible to just reinitialize the pointers to these buffers, when the stack needed to be reinitialized.

Again the solution is a buffer-manager. The buffer-manager allows us to remove all the internally allocated buffers, thus making the re-initialization problem the trivial job of evicting all buffers from the stack.

5.4 Storing Sensor Data

To duty cycle the Bluetooth module, we have to store the data we sample from the accelerometers somewhere, while the Bluetooth module is turned off. The most obvious choice is to store it in the data memory of the node. However as described in Section 4.2, we disabled the external memory to conserve energy, leaving us with 4 KiB. Since the Bluetooth specification requires that the buffers used for communicating with the Bluetooth module are larger than 255 bytes[11, Part H:1], we will at least have to allocate 512 bytes of memory for these buffers: one for receiving and one for sending. This leaves us with 3.5 KiB, some of which is used by the application variables, and some is used for maintaining the state of the Bluetooth stack. So at best we will have 3 KiB available to store sensor data in.

As we decided sample at 4 Hz (see Section 3.2), and each sample from the 2-axis accelerometer is 10 bits wide and each sample from the 3-axis accelerometer is 12 bits wide, we are going to use 30 bytes each second, if we simply store the data as tight as possible. This will allow us to store about 100 seconds of sensor data in memory. This will not allow us to duty cycle the Bluetooth module enough to expect that the node will last for 20 days. Therefore we have to consider other options.

With the BTnode we have two options:

- We can use the unused parts of the program memory on the ATMega128.
- We can attach external flash to the BTnode, as is done on the Mica motes[29].

If we choose to use external flash, we are free to choose the size of the flash, and since we already are developing a sensor board, we could simply attach the flash to the same sensor board. However if the program memory of the ATMega128 could be used, this would be preferable from a cost and simplicity point of view.

The ATMega128 MCU has 128 KiB of flash for the program memory. Of these the top 8 KiB are reserved for code that writes to the program memory. Also some of the program memory is going to be used for the application. So if we assume that there is 100 KiB program memory available for us to store sensor data in, we actually have room for just above 55 minutes of sensor data. This will hopefully be enough to keep us within the energy budget, but we will explore this further in Section 7.3.

There are several interesting problems related to using the program memory on the ATMega128 as storage. First of all we need to know how to make the MCU overwrite its program memory. Secondly we need to ensure that we do not overwrite the application itself. Thirdly we need to design an interface that allows us to access the flash from within TinyOS. In the following sections we will discuss these problems.

5.4.1 Accessing the Flash

The AVR architecture used in the ATMega128, is a so-called Harvard architecture, which means that the program and data memory are separated[27]. This means that in order to access the program memory from within a program, special instructions will have to be used. For the ATMega family the instructions are called `LPM` (Load Program Memory), `ELPM` (Extended Load Program Memory) and `SPM` (Store Program Memory). The `SPM` instruction only has an effect when issued from the

highest 8 KiB of the memory, while the `LPM` and `ELPM` instructions can be issued from any point in the program memory.

The `SPM` instruction can only be issued from the No Read While Write memory segment[4] (also called NRWW). The rest of the memory is called Read While Write or RWW. If a page is written in the NRWW part of the flash, the MCU stops all processing until the page have been written. This is not the case if the page is written in the RWW part. While such a write is in progress only program memory in the NRWW part of the memory can be accessed.

Reading from flash is simple. One can read both bytes and words anywhere in memory using one of the `LPM` or `ELPM` instructions. Writing to flash is more complicated. It is performed page-wise, with pages of 256 bytes. Writing a page is a 3 step process:

1. The temporary page buffer is filled. This is a 256 bytes write-only area of the chip. Filling this buffer is done two bytes at a time, by calling the `SPM` instruction.
2. The page to be written is erased. This is once again done by issuing the `SPM` instruction. Once the page is erased, all the bits in it are set.
3. The data stored in the temporary page buffer is transferred to flash using the `SPM` instruction. In effect what happens is a bitwise `AND` operation between the values in temporary page buffer, and the values in the flash.

Step one and two are interchangeable. The flash is guaranteed to have a data retention time of 20 years[7]. Once more than 10,000 erase/write cycles have been performed on a single page, this retention time is no longer guaranteed. Writing or erasing a page in flash takes between 3.7 ms and 4.5 ms[8].

In the standard library for the AVR platform, several functions are defined, which makes access to the program memory easy from C[9]. The interesting functions for our purpose are:

`pgm_read_word_far` - Reads a word from an address in the program memory. The word can be read from the entire 128 KiB of the memory. A similar function called `pgm_read_word_near` is available, but this function is only capable of reading from the first 64 KiB of memory.

`boot_page_erase` - Erases a page in the program memory, setting all bits in the page back to 1.

`boot_page_fill` - Writes a word to the temporary page buffer.

`boot_page_write` - Writes the contents of the temporary page buffer to a page in the flash.

`boot_rww_busy` - Checks if the Read While Write memory is busy. Whenever the functions `boot_page_erase` or `boot_page_write` are executed, the RWW section of the memory is set to busy, meaning that it cannot be accessed.

`boot_rww_enable` - Tries to enable the Read While Write memory again. This must be done once the flash operation is finished. Otherwise the MCU will not allow access to the RWW memory.

[4] Actually the SPM instruction can only be issued from the Boot Loader Area, the size of which can be selected by setting fuses on the MCU. However the Boot Loader Area can at most be the entire NRWW memory, or 8 KiB.

WRITING TO FLASH (5.1)

```
1:  Input: The address to write to (addr)
2:  Input: A 265 byte array containing the new contents of the flash (data)
3:  WRITETOFLASH()
4:      cli();
5:      for (int i = 0 ; i < 128 ; i++) {
6:          boot_page_fill(i, data[i * 2]);
7:      }
8:      boot_page_erase(addr);
9:      while (boot_rww_busy()) {
10:         boot_rww_enable();
11:     }
12:     boot_page_write(addr);
13:     while (boot_rww_busy()) {
14:         boot_rww_enable();
15:     }
16:     sei();
```

The pseudo code describing how to write to the flash, can be seen in Algorithm 5.1. This code shows two problems with the process:

Interrupts are disabled throughout the process. As it takes at least 3.7 ms to erase a page in flash, or to write to it, the above pseudo code will disable interrupts for a total of at least 7.4 ms. This is bad, especially when the Bluetooth module is active, as we can miss interrupts from the UART.

The code busy waits for the page to be written or erased. The busy wait results in the MCU using more energy than necessary. If we could enter a sleep-mode, while the flash is being written, we would be able to reduce the energy consumption.

Disabled Interrupts

Interrupts are disabled during the flash operations, as the `SPMEN` bit in the `SPMCSR` register needs to be set for the `SPM` instruction to be executed[8]. This bit is automatically reset after 4 clock cycles. If interrupts are not disabled, an interrupt could fire between the instruction setting the bit, and the `SPM` instruction, resulting in the `SPM` instruction not being executed. The `SPMEN` bit needs to be set, no matter which action the `SPM` instruction needs to perform. So we absolutely need to disable interrupts while the `SPM` instruction is executed.

The interrupts could still be enabled during the busy wait stages. But as all the interrupt handlers are placed in the RWW part of memory — which is disabled during the write to flash — we would not gain anything from this.

This could be solved by moving the interrupt handlers to the NRWW section, together with the flash writing code. However this would also require that we move all the code that is called by the interrupt handlers, into the NRWW section. This is impractical, as it requires changes to all components used by our application. Furthermore we cannot be certain that all the required code can fit into the 8 KiB available for the NRWW section.

Busy Waiting

To get rid of the busy wait, the MCU supports raising an interrupt whenever the `SPM` instruction completes. So we might be able to disable all other interrupts, and enter a sleep-mode until the write or erase is finished.

However all of the supported sleep-modes for the ATMega128, even the Idle mode disables the clock that drives the flash[8]. So if the MCU enters sleep-mode, it will not wake up when the write or erase is finished, but first when an interrupt from an external device fires. Therefore we decided against removing the busy wait.

5.4.2 Placing Code in the Boot Loader Area

Writing to the flash requires, as previously discussed, that the code issuing the `SPM` instruction resides in the boot loader area. The boot loader area is at least 2 KiB large, so if the code is placed at the 126th KiB in the flash, it can use the `SPM` instruction no matter how the fuses on the MCU are programmed.

We are not going to be able to store the entire application in the boot loader area, so we need some way to inform the TinyOS tool-chain that certain functions needs to be placed at specific locations in the memory.

The GNU GCC[5], can place a function in a different section of the object file. If `__attribute__((section(".bootloader")))` is added after the function declaration, the function is placed in the `.bootloader` section. The GNU Binutils linker can then place the function from the `.bootloader` section at the memory address `0x1FC00` if the parameter `--section-start=.bootloader=0x1FC00` is given on the command line. The address `0x1FC00` is the beginning of the last 2 KiB of program memory.

The nesC pre-compiler complicates this, as it inlines all functions. Therefore chances are that the `.bootloader` annotated code will be inlined inside another function which is not located in the `.bootloader` section. However if the string `__attribute__((noinline))` is added to the function declaration, GCC will not inline that function. nesC does not recognize the `noinline` attribute, so in the code it delivers to GCC, all functions still receives the `inline` keyword. This causes GCC to inline the function, while warning that the function both is specified to be inlined and not to be inlined. We created a patch to address this problem.[6] The patch makes nesC respect the `noinline` attribute, by not asking for such functions to be inlined. This patch have been integrated into nesC 1.1.2 and higher.

5.4.3 Finding Unused Flash Pages

When storing the sensor data in the flash, it is very important not to overwrite the program code. Therefore we need a way to find the pages which is not used by the program.

Finding the upper bound is simple, as we define it ourselves. The last usable page is simply the one just before the page where the `.bootloader` section starts.

Finding the first unused page is a bit more problematic. A simple solution would be to hard code the value. The problem with this solution is that it is very error prone, and the errors might not be easy to discover: If we overwrite program

[5] GNU GCC is the compiler used by TinyOS for the BTnode.
[6] Available at `http://sf.net/mailarchive/forum.php?thread_id=4955062&forum_id=27020`

code, and the application actually calls this code, we will discover it. But initial values for global variables etc., are also stored in the program memory, and copied to RAM during the initialization of the node. Overwriting a page containing this data will be harder to discover, as the error only shows itself when the node is rebooted. Therefore the ideal solution would be a way to determine the first usable flash page from within the application.

The linker helps us to determine the first unused page. It inserts a symbol into the finished program code, which marks the end of the program and initial values. This symbol is called `__data_load_end`. Normally symbols are used to mark the location of a function or a shared variable, so to access this symbol from C, we need to declare a variable, like so:

```
extern const prog_int32_t __data_load_end;
```

This declaration tells the compiler, that a symbol called `__data_load_end` is defined somewhere else, and we know it is a constant 32-bit integer stored in program memory. The value or type of this variable does not matter. To figure out where the program code ends, we simply look at the address of `__data_load_end`, which will point to the end of the program code. Using this address it is simple to find the first usable page.

5.4.4 A TinyOS Component for Accessing the Flash

Components to access flash already exists in TinyOS. If we could use the interface `PageEEPROM`, which is the same interface used to access the external flash on the Mica motes, it would be easier to make things such as Matchbox[24] (a simple flash file system) use the internal flash.

However the `PageEEPROM` interface is modeled very closely to the fact that the flash is external. The interface requires that the implementation keeps a cache of the last accessed page in memory, which allows an application to write to the same page in several calls, without writing to the flash in between.

Since the `PageEEPROM` interface needs to use 256 bytes of RAM to hold a page of the flash, and since the it would require a lot of work to support it, we decided against using it, opting instead to construct our own interface. This interface is flexible enough that it should be possible to create a new component to emulate the original `PageEEPROM` interface, if a need for this arises later on.

The interface for the `FlashAccessM` module can be seen in Figure 5.5. The two commands `firstUsablePage` and `lastUsablePage` can be used to retrieve the first and last page that can be overwritten without overwriting parts of the program. The `read` and `read_some` functions read from a specific page. There are functions for this in the Standard Library for the AVR platform but, it is advantageous to be able to perform these functions with the same addressing as the `erase` and `write` functions. The `erase` and `write` functions respectively erases and writes a page to flash. Both of these functions disable interrupts for periods of time up to 4.5 ms.

5.5 Summary

In this chapter we have created or adjusted the low level software to the needs of our application:

- We have designed interfaces for the accelerometers, and implemented these.

Figure 5.5 – *The `FlashAccess` interface*

```
interface FlashAccess {
  async command uint16_t firstUsablePage();
  async command uint16_t lastUsablePage();

  async command result_t read(uint16_t page_no, void *page);
  async command result_t read_some(uint16_t page_no,
                                   uint8_t start,
                                   uint8_t count, void *data);

  async command result_t erase(uint16_t page_no);
  async command result_t write(uint16_t page_no, void *page);
}
```

- We have devised new a power management interface for TinyOS, which will help us in reaching the goal of deploying the node for 20 days.
- We have made changes to the TinyBT stack, to make it ready for production use, including stability fixes and code clean up.
- And we have presented a novel approach to data storage, using the built in flash on the ATMega128. This allows us to duty cycle the Bluetooth module, without adding external flash to the node.

With these pieces in place, we can proceed to defining how the data gathering application should work. This will be the subject of the next chapter.

5.5.1 Further Work on TinyBT

The state of TinyBT after the changes described in Section 5.3 is quite good. However there are still many points where it could be improved.

It is possible to use the stack in a production environment now, provided one adds some timeouts to the application code which, if reached, restarts the TinyBT stack. This can happen if we receive bogus data from the Bluetooth module, or if we drop a byte or packet. This can happen, if interrupts are disabled for long periods of time.

The solution for this problem is to add error handling as per the Bluetooth specification[11, Part H:4]. This states that the node should send a `HCI_Reset` command to the Bluetooth module, whenever it encounters an invalid HCI packet indicator (which is the first byte of a packet), or if the length field is out of range. This will catch most problems, and get the node and Bluetooth module communicating correctly again. However a timeout for how long the node can spend receiving a packet also seems in order. Otherwise, if the node misses a single byte in a packet, it has to wait for the next packet to arrive, before it detects the error. In our deployment this error-recovery was implemented in the application, and was timeout based (see Section 6.5.1 for details). Whenever one of the timeouts triggered, the application simply power-cycled the Bluetooth module.

To make the stack more flexible, it would also be a good idea to move all the node/Bluetooth module specific code into its own component. This would make it easy to port the stack to other Bluetooth modules or nodes, as one would only have

to replace this one component. This is very relevant, as a BTnode revision 3 exists, which have switched to a Zeevo Bluetooth module. Currently no port of the TinyBT stack have been made to this module, but that is an interesting task.

A more cosmetic issue with the current TinyBT implementation, is that it is implemented as one interface. This means that every application has to implement 10 – 15 signal handlers, even though the application might not use all that functionality. For example, an application that never scans for other Bluetooth devices, needs to implement the events `inquiryCancelComplete`, `inquiryResult` and `inquiryComplete`. If all the inquiry commands and events was put into one interface, the application could simply choose not to use that interface, causing the default implementations of the events to be used instead. This would also make the application source code much easier to read.

One last thing that is worth considering is that Intel have released a node, called the Intel Mote[44]. This node features a Zeevo Bluetooth radio, and Intel have provided a Bluetooth stack for this in the TinyOS tree. Furthermore they have implemented a automatic network assembly protocol, for these nodes. It might be useful to merge this stack with the TinyBT stack, especially if they have implemented the L2CAP protocol.

Chapter 6

The Application

With the accelerometers selected, the sensor board manufactured, and the low level software in place, we are ready to design the main application for the experiment. For this we are going to need the following:

- A way to time-stamp the data gathered by the node, so we can find the time when a specific sample was obtained.
- An on-flash storage layout, that optimizes the amount of data we can store in the flash.
- A protocol that will allow us to efficiently offload data to a PC.
- A way to debug the nodes in the field.

Apart from these 4 problems, we will look at several ways to ensure the reliability of the application during the experiment. We will also look at the memory requirements of the final application, both with regard to program memory and data memory.

6.1 Time Synchronization

To correlate the data gathered on the node with the data from the video cameras, we need to have some way of time-stamping the data.

We know that the time between two consecutive samples will be the same through out the experiment for each node, even though we do not know how long this period will be. This is because the different nodes have different clock drifts. Therefore we introduce a sample counter. When the timer used to obtain samples fires, this counter is incremented by one. The value of this sample counter is then written in the four first bytes (32 bits) of each page we store in the node's flash memory. This will provide us with redundancy, making it possible to find the sample number of a specific sample even if some pages are missing.

Whenever we offload data to one of the PC's, we include the current value of this sample counter in the initial packet (which also contains a dataset identifier and the number of available pages, more on that in Section 6.3.5). When the PC receives this packet it immediately assigns a time-stamp to it, and stores this information.

To figure out when a specific sample was obtained, we can look at the pairs of time-stamps and sample counter values, and calculate the average time between two samples. We can then use this average to calculate the time a specific sample was obtained.

Since this method does not take the time it takes to transmit a packet into account, there will be a small difference between the real time, and the time at which a sample is obtained. But this difference is going to be much smaller than the time between two samples. So this will provide accurate enough time-stamping for our purpose.

6.2 Flash Page Layout

As described in the previous section, we store a 4 byte sample number at the beginning of each page of data we store on the node. In this section we will discuss how we store the accelerometer measurements in the remaining 252 bytes. For this experiment, we want the application to be as deterministic as possible, which is why we will not use compression to store the samples. Instead compression will be explored thoroughly after the field experiment, in Chapter 10.

We cannot be certain of the order in which we receive the measurements from the analog and digital accelerometer. Also in a few cases, a measurement is never returned by one of the accelerometer components.[1] A way to solve this is to lead a sensor reading with a number that identifies if it is an analog or digital reading. After the identifier, the actual sensor reading is written.

The initial straight forward way to store the data, is to use a byte for each identifier, and 2 bytes for each axis. This means that we would have to store 12 bytes 4 times each second, and that a page is filled in 5.25 s. There are 512 pages available on the node in total, but some of these are used by the application. A conservative estimate is that we will have 400 pages available for data storage, and therefore will fill the memory in 35 minutes.

Some simple observations can save us a lot of space: If we start with the identifier, we only need two bits to represent it. One is not enough, because we need to support three states. One for each of the accelerometers, and one we can use to mark the end of data in a page, when there is no more room for measurements. For the digital accelerometer we need 12 bits per axis, and for the analog 10 bits per axis. This means that we will only have to write 60 bits (or 7.5 bytes) 4 times each second, filling a page in 8.25 seconds, and all the available pages in 55 minutes.

To ensure that we do not destroy the flash, we will store the measurements in a cyclic buffer. This will ensure proper wear-leveling of the flash, and make sure that no page is written to more often than others.

6.3 Offloading Data

Offloading data from the node should be done as fast as possible, as the radio is extremely energy consuming. We also need make the offloading algorithm as error resistant as possible, so data cannot be corrupted during the offload. The last goal is that the algorithm should be simple, to make it easier to implement. This in turn should lower the risk of programming errors, which is especially important on the node, as errors there will be hard to correct, once the experiment is running.

We will not concern ourselves with multi-hop routing, as this will complicate the offloading mechanisms. Furthermore there is plenty of power available in the

[1] We have not explored this in depth, as we did not have a JTAG interface to assist debugging the problem. However it is most likely caused by a timing issue.

stables, so we should be able to have Bluetooth coverage everywhere.

In the following we will discuss solutions to the different problems in discovering nodes and offloading data, while keeping the above in mind.

6.3.1 Memory Considerations

As described in Section 4.2.1, we decided to disable the external memory. This leaves us with only 4 KiB, for the stack, application variables, a buffer to store the sensor data in, and buffers for the Bluetooth communication.

We will need a buffer to store the sensor data in, so that we only need to write each page of flash once, to fill it. This buffer needs to be at least 256 bytes large. The alternative is to write several times to the same page, taking advantage of the fact that a write to flash, in effect is a bitwise `AND` operation[62]. However this solution will result in a higher energy consumption, in part due to the approximately 4 ms busy wait needed for each write (see Section 5.4.1 for details).

The buffers for the Bluetooth communication will have to be at least 256 bytes large, to satisfy the requirements in the Bluetooth specification[11, Part H:1]. However to be able send an entire page in one packet, we will for the sake of this argument say that a packet should be 300 bytes long.

We have performed tests with the TinyBT stack that have shown that we will need at least 3 free buffers when sending at full speed, to not miss control packets from the Bluetooth module. This means that we at least will have to use 4 buffers, as we will also need one for the data to send. Having only one send buffer, will impose a limit on the transfer rate, so 6 buffers in total would be more ideal.

We will need 1456 bytes of memory for 4 Bluetooth buffers, and the flash buffer, i.e. more than one third of the available memory. In an ideal situation where we have 6 packets available for Bluetooth communication we are going to use 2056 bytes, or just over half of the available memory. If we are going to use 6 buffers for the Bluetooth communication, the rest of the application should not use more than 1.5 KiB, because we also need some memory for the stack.

6.3.2 Initiating Bluetooth Communication

Bluetooth communication is initiated with a device discovery, followed by the establishment of a connection.

Device discovery is done by sending “inquiry” packets. The devices that wishes to be discovered needs to be in a complementary “inquiry-scan” mode, in which the Bluetooth module listens after and responds to inquiry packets.

During discovery, the devices are not synchronized, so the device doing the inquiry will need to send enough packets that a device in inquiry-scan will be discovered. Measurements using a older version of the BTnode, but with the same Bluetooth module, shows that performing the inquiry costs twice as much power as listening in inquiry-scan mode[32]. An inquiry must last for at least 10.24 s in order to collect all responses from devices[11, Part B:10]. However 99% of the devices can be expected to be discovered in approximately 5.5 s[32, 35].

So to conserve as much energy as possible on the node, the PC should do the inquiry, and the nodes should only turn on the Bluetooth module, put it into inquiry-scan mode and wait for a connection from a PC.

Once a PC have performed an inquiry, it should check if the result matches

Table 6.1 – *The different ACL Packet types, and their characteristics[11, Part B]. The transfer speeds in this table are theoretical maximum values.*

Packet Type	Max size (bytes)	Max symmetric data rate (kbit/s)	(KB/s)	Error correction
DM1	17	108.8	13.6	FEC & CRC
DH1	27	172.8	21.6	CRC
DM3	121	258.1	32.3	FEC & CRC
DH3	183	390.4	48.8	CRC
DM5	224	286.7	35.8	FEC & CRC
DH5	339	433.9	54.2	CRC

any of the nodes participating in the experiment. If so, the PC should create a connection to the node. Once the connection have been created the two devices can communicate with each other.

6.3.3 Choosing a Packet Type

Each connection has a packet type associated. The packet type affects how large packets can be before they are fragmented (actually how many protocol time-slots are used to send the packet), and it also controls the error correction algorithms used for the packets. The fragmentation of packets is handled automatically by the Bluetooth module. As a larger packet-size only to a certain extent means higher bandwidth[38], we want to select the packet type that fits best with the amount of data we want to send.

There are two types of error correction for ACL packets[11, Part B], FEC (Forward Error Correcting) and CRC (Cyclic Redundancy Check). The 2/3 rate FEC check is capable of correcting all single errors, and detecting all double errors within 10 bits. It achieves this by coding each 10 bit block into 15 bits. The CRC check is a 16 bit CRC-CCITT, but instead of an initial value of 0xFFFF, the UAP (Upper Address Part, bits 24 to 31 of the device address) is used as the lower 8 bits. For the packets using both CRC and FEC checks, the FEC encoding is applied after the CRC have been added.

Table 6.1 contains a list of the different types of ACL packets, and their characteristics. As can be seen, all of the packets have error correcting capabilities, which means that our evaluation will have to be based on the other characteristics.

When we offload data from the node, the natural chunk-size is 256 bytes, as this is size of a page in the flash. The only packet type that will encompass a whole flash page is the DH5 packet. This packet type has CRC, but not FEC. If we wish to use a packet with FEC, the most obvious choice would be the DM5 packet. We would then have to split a flash page into 2 packets, which would make the solution more complex, as we would have to be able to re-request each part of the page. But as DH5 is the packet with the highest data rate, this seems like the obvious choice.

We have performed a simple test using different package types, with different packet sizes, and with different UART speeds. To make the test resemble the real scenario, we only send data from the node to a PC, and we send it as quickly as possible. The three different packet sizes we have tested are, the maximum size that can be sent in a single packet, 135 bytes (the size of half a page, plus some

Table 6.2 – *TinyBT Transfer Speed*

Packet Type	Transfer rate in *kbps*					
	UART @ 230.4 kbps			UART @ 460.8 kbps		
	Max size	135 B	270 B	Max size	135 B	270 B
DM1	53.25	105.01	105.12	51.73	105.36	105.62
DH1	80.77	167.83	167.04	81.73	169.05	168.82
DM3	173.35	175.56	179.73	313.84	211.47	282.30
DH3	177.60	174.91	179.33	334.12	319.74	345.68
DM5	179.09	175.10	179.68	340.27	281.81	282.40
DH5	180.73	175.10	179.92	347.01	281.54	345.54

bookkeeping information) and 270 bytes (an entire page plus bookkeeping). The two UART speeds we have tested are 460.8 kbps and 230.4 kbps. The results are available in Table 6.2.

If we start by looking at the low UART speed, we see that the UART is quickly becomes the bottleneck. Even when using DM3 packets, we are limited by the transfer speed of the UART. If we look at the high UART speed, it is obvious that we should use either DH3 or DH5 if we want to have the highest possible transfer rate. We do not reach the transfer speeds from the specification, but this is most likely caused by limiting factors on the node.

These results contradict previous findings, where the best bandwidth is measured with the DH3 packet type[38]. With that packet the throughput is measured as 304.8 kbps, which is lower than our results. But their tests are also performed with a packet size of 668 bytes, where our best case for that packet type use a packet size of 270 bytes. Another cause of this difference can be the amount of noise in the 2.4 GHz band during the experiment.

We choose to use the DH5 packet, because this will be the easiest option when developing the application and protocol. We do not expect a lot of noise in the 2.4 GHz band in the stables, so choosing this packet should also give us the best transfer speed.

6.3.4 Bluetooth Communication Problems

During our tests with the BTnode and with an early implementation of the application we noticed several problems with the Bluetooth communication. The initial tests were done with an extremely simple protocol. The PC initiates the connection to the node, and once the node receives the connection, it starts offloading all its data, one page in each packet, and closes the connection once finished. Testing with this protocol revealed the following problems:

- When offloading a node with full memory, the connection would sometimes fail halfway through the offloading process.
- Once in a while a single packet would be missing in the data stream.
- The application would in rare cases receive many errors from the TinyBT stack, indicating half received packets and the like. Usually the only way to get the communication back on track was to power-cycle the Bluetooth module.

The first two problems seems like problems with either the Bluetooth module or with the TinyBT stack. However we have spent much time looking at the code to see if there was any potential problems, and found none. So in the end we simply chose to make a workaround for these problems.

The problem with many errors from the TinyBT stack is caused by writing to the flash. As explained in Section 5.4.1, writing or erasing takes from 3.7 ms to 4.5 ms. In this period all interrupts are disabled. But with the high UART communication speed, we receive 15 bits in approximately 0.04 ms[40]. So when we write to flash while receiving a packet from the Bluetooth module, we will miss data.

Lowering the UART communication speed to 230.4 kbps, makes these problems occur much less frequently, because we are less likely to miss data, and this makes the stack stable enough for production use. Therefore we decided to use this UART speed for the deployment, even though the transfer-rate is significantly lower.

However the other two problems highlight the need for a more elaborate transfer protocol, which we will present in the following section.

6.3.5 Transfer Protocol

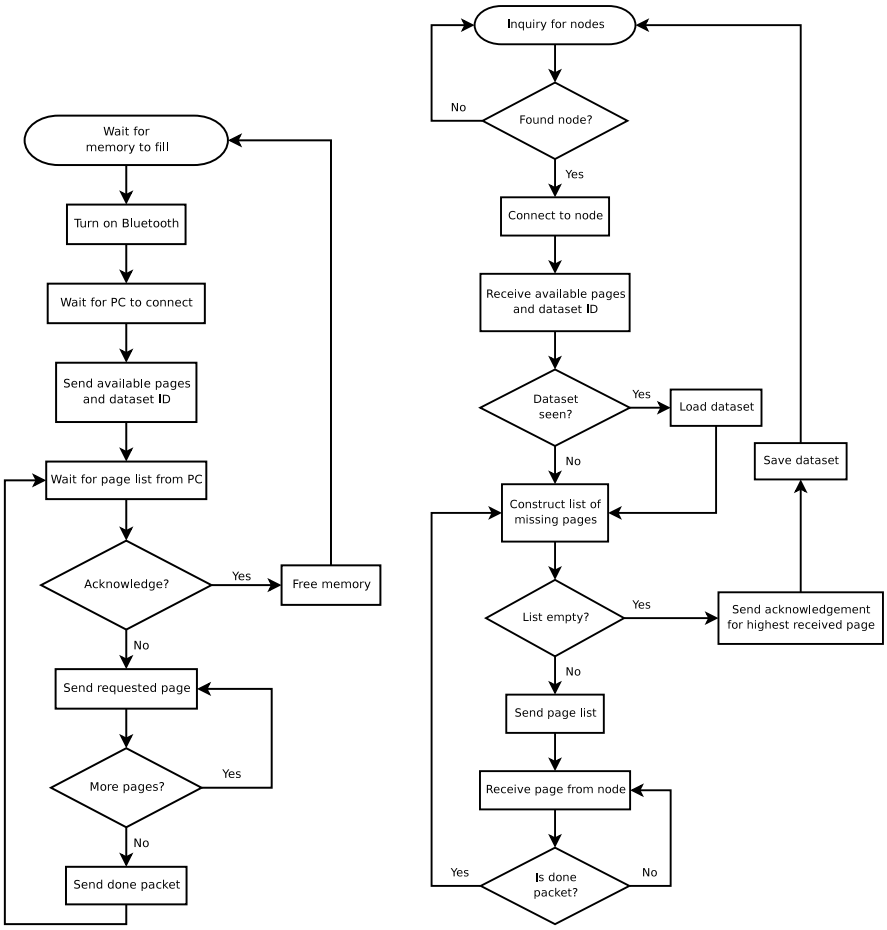
Now that we have decided on how to discover nodes, and the packet type of the connection, we need to decide on a protocol for transferring the data from the node to the PC. We want to keep the protocol as simple as possible, and to push as much of the bookkeeping to the PC side. The PC is much easier to debug than the node, so having the complex part of the communication protocol on the PC side, will make it easier to implement the protocol. Once the experiment is deployed it will also be much easier to upgrade the software on the PC, than the software on the node.

We decided to implement the following protocol. Once the connection have been established, the node sends the number of available pages to the PC, together with a dataset identifier. From the dataset identifier the PC is able to tell if it already has seen the dataset on the node, as the identifier should be unique for as long as the program is running on the node. We choose to use the sample counter value from the first flash page for the identifier, as it has the these properties. From the dataset identifier and the number of available data pages, the PC creates a list of all the pages that it does not have currently, and sends this to the node. For a new dataset, it simply requests all pages from 0 to the number of available pages $-$ 1. The node then sends all the requested pages, each in their own packet, and when it is done, it sends a packet to signal that no more packets will be sent. Now the PC looks through its list of received pages, and checks if there is any missing. If there is, a new list of pages is created. Otherwise a packet which acknowledges all the pages received by the PC is sent. When the node receives this packet, it frees the memory used by the pages that are being acknowledged, and updates the dataset identifier. A detailed state diagram for the protocol can be seen in Figure 6.1.

There are obvious problems with this protocol. One of the major ones is that no retransmission of control packets takes place. So if the PC misses the done packet, or if the node misses an acknowledge or page request packet, the offloading process deadlocks. To solve this, we introduce timeouts on the node, whenever it expects to receive a packet from the PC. We do it on the node, because we need to make sure that a failed communication will not leave the Bluetooth module turned on.

This very simple protocol, allows us to re-initiate an offload at the point where a previous offload stalled. It also allows us to resend a single missed packet, without

Figure 6.1 – *State diagram for the protocol between the node and PC*



(a) The protocol on the node

(b) The protocol on the server

much overhead.

6.3.6 Duty Cycling

In Section 6.2 we calculated that the node will use 8.25 s to fill a page, and that it will fill the memory in approximately 55 minutes, depending on the size of the application.

Our main goal is to make the node last the entire 20 days of the experiment. Therefore we choose an aggressive duty cycling of the Bluetooth module, waiting until only 15 pages of flash is left. This gives the node almost 80 seconds from it turns on the Bluetooth module the first time, till the memory is filled entirely.

While very aggressive, we decided that this would be reasonable, as the PC's inquiry takes about 10 s, and the offload with a full memory takes 6 seconds. So this means that the PC's can scan for the node 6 times in a row before it discovers the node, without the node loosing data, provided that no other errors occur. Also a simple lab test with two PC's and a single node checking in for a week did not reveal any problems with this assumption.

6.4 In Field Debugging

As a way to help determine the cause of failure when the nodes are deployed, we included some statistics about the health of the node in the initial packet. These statistics include, amongst other things:

- The number of failure events received from the Bluetooth stack, since the last received acknowledgement.
- The number of automatic reboots, since the last acknowledgement.
- The number of results received from the analog and digital accelerometers.
- A measurement of the battery charge indicator.

During the stability tests of the node, we introduced a simple debugging mechanism. When the node is only sampling, all the LEDs on the node are turned off. Once it turns on the Bluetooth module, the first LED on the node is turned on. Once the Bluetooth module is in inquiry-scan mode, the second LED is turned on, and when a PC is connected the third LED is turned on. This made it very easy to see what the node was doing.

We did not want to do this in the experiment, as a single LED consumes about 20 mA. So to provide this debugging information in the stable, we created a replacement component for the `LedDebugC`, used to control the LEDs on the BTnode. The `FakeLedDebugM` component, controls some unused pins on the MCU, instead of controlling the LEDs. The status of these pins can then be checked by using a voltmeter. Even better, a simple LED array can be constructed, which can be attached to the node when we want to know the state of the node. While the node still consumes a small amount of energy to keep the pin high, it is much less than if a LED was used.

To make it easier to determine if the node was functioning properly, we used the LEDs during the first offload. Also, we ensured that the first offload would happen after 10 minutes, so that we could package the node and would not have to wait an hour to find out if it worked properly.

Another valuable debugging feature would be the ability to turn on the debug UART. In normal usage, we have to turn it off, to conserve energy. But one could construct the application so that it samples the value of a single pin, at the same time it obtains a sample from the accelerometer. If 4 consecutive samples of the pin indicates that the pin is low, it activates the UART.

We decided that this would be too much work, as it required many changes to the `StdOutC` component, which we used to write debug information to the serial port.

6.5 Reliability

In this section we will describe the measures we have taken to make sure that the application works reliably when deployed. This includes making sure that the node cannot end up with the radio turned on for a prolonged period, and guarding ourselves against programming errors as much as possible.

6.5.1 Making the Protocol Robust

We implemented the simple protocol described in Section 6.3.5. However in the event that no PC connects, we created a back-off algorithm, to ensure that the node would not deplete its batteries completely.

When the Bluetooth module is turned on, the application waits for 30 s. If it does not receive a connection within this time, an internal counter is incremented, and the Bluetooth module is powered down for 10 seconds. Once this counter have been incremented 3 times, the period where the Bluetooth module is turned off is raised to 1 minute. After three more passes the turned off time is raised to 15 minutes, and after three more the Bluetooth module is turned on every 30 minutes. At this point the turned off period is not raised any more. The connection timeout is not the only time the counter is incremented. It is also incremented if a connection is established, and the server does not send the page list within 2 seconds, or when other timeouts that could point to a problem in the communication are triggered. To return to the 10 seconds turned off period, the node must receive an acknowledgement packet from the PC, indicating that all the pages have been offloaded successfully.

This back-off algorithm is very aggressive, but this is a deliberate choice. We want to preserve as much energy as possible, in the event that both PC's or the off-loading program on the PC's fails.

In addition, we put a Cyclic Redundancy Check (CRC) on all packets. This is somewhat superfluous, as the ACL packets already contain a CRC check. However there is no CRC check on the UART communication between the MCU and the Bluetooth module. And since the CRC was already implemented for use in the other radio stacks in TinyOS, we decided that this was a small enough change that we wanted this extra protection.

6.5.2 Automatic Reboot

As mentioned in Section 5.1.2, we initially had some problems where the node simply started misbehaving, when we used the I^2C interface to the digital accelerometer.

This experience caused us to implement a controlled reboot of the node. If the application received a lot of Bluetooth errors, or if it did not receive a connection from a PC within a reasonable time period, even though the Bluetooth module already had been power-cycled three times, the node initiates an automatic reboot. This also happens when the application notices that it does not receive readings from one of the accelerometers.

Before the node is rebooted, the state is written to flash. The first available flash-page is reserved for this purpose. All the flash-memory management state is written there, together with a mark, so that the application can detect if a reboot was initiated by the application or by other means. Also if the flash is not filled up, the temporary buffer we use to store the accelerometer readings in is also saved, as is the pointers and counters, which tells the application where to write.

Once we have written all the state information to the flash, we trigger the watchdog timer. The normal use of the watchdog timer is to perform a reboot if the application have crashed. That is, the application has to "pet the dog" (execute the `WDR`-instruction) at intervals to prevent the node from rebooting. Since we are interested in rebooting the node, we simply enable the watchdog timer, and refrain from "petting the dog".

When the node is rebooted, the cause can be read from the `MCUCSR` register. So as a precaution we will only check for the mark in the first unused page of flash memory, if the node was reset by the watchdog. As another precaution, we erase the page that holds the state, as soon as we have read it in. This way we are certain that if something goes wrong, the node will not continually reboot, as the second reboot will be detected as a power-on, because the mark is erased.

Even though this code was originally written to deal with the I^2C problems, we decided to keep it for the deployment. Potentially it can save the node from a corruption in the RAM. Once the node reboots all the memory is reinitialized, eliminating any RAM corruption, and any values read from the flash are checked to make sure that they are valid.

6.6 Size of the Final Application

For the TinyBT project a simple script was created which could calculate the code size and memory requirements of each of the components in a TinyOS application. This is not to be considered a precise measurement of the size of a specific component, as many functions are inlined when the code is compiled, and thus may feature in another components code size.

A list of component sizes for the final application can be seen in Table 6.3. The interrupt routines are folded in to their respective components, and all the basic code from TinyOS is aggregated into a TinyOS component.

By far the largest component in the table is the `PowerTesterM` component, which is the application that controls all the other components. The three components `HCICore0M`, `HCIPacket0M` and `HPLBTUART0M` together implement the TinyBT stack. So the code size of the new and modified TinyBT stack is 2830 bytes, where it before our improvements was 3178 bytes[36]. The difference is not enough to decide if our version is lighter than the original version. Our code have been compiled with GCC version 3.4.3, while the original code have been compiled with an earlier version of the compiler. So this difference might as well be attributed to the

Table 6.3 – *Code size and memory usage of the different components*

Component	Code size (bytes)	Memory usage (bytes)
ADXLAccelM	464	6
BTPacketHandlerM	226	1836
CRC16M	562	0
FakeLedDebugM	50	0
FlashAccessM	348	5
HCICore0M	568	10
HCIPacket0M	1490	18
HPLBTUART0M	722	4
PowerTesterM	3330	360
STM3DigiSPIM	486	11
TimerM	2040	64
TinyOS	1328	20
Total	*11614*	*2334*

new compiler, as to the changes in the code. But the possible inlining of code into other components can also cause such a difference.

If we look at the total code size, the entire application only uses 11.3 KiB. This means that we have 458 flash pages left to store accelerometer measurements in, which will give us 63 minutes before the application needs to offload data.

As to the memory usage, the two biggest consumers are — as expected — the `BTPacketHandlerM` (which declares all the buffers for the Bluetooth communication), and the application `PowerTesterM`. The `BTPacketHandlerM` contains 6 buffers for the communication, so we reached the ideal situation described in Section 6.3.1. Furthermore we have only used 2334 bytes in total, so there is still a lot of memory left that could be used in the implementation of a compression algorithm.

6.7 Summary

In this chapter we have described how we have implemented our application. We have designed a simple system, that can give us the time when a specific sample is obtained. We have also devised a storage format, that we can use to store measurements in the flash. We have designed a simple protocol that is well suited for communicating over Bluetooth, and which works around the problems we have discovered with the Bluetooth communication. The final application and the supporting software is available from `http://hogthrob.42.dk/application`.

The application ended up using so little data memory that we can allocate 6 buffers for the Bluetooth communication, ensuring the best transfer rate.

6.7.1 Possible Improvements

Because of the way the application evolved, and the limited time we had to prepare the application there are several things that could be made better.

All the protocol code should be contained in its own component. As the protocol code does not have any state that needs to be saved across a reboot, this

should be simple. A component with a two simple commands is all that is needed. One that tells the component to start the offloading protocol, and one which updates the amount of available pages. The component would then use an event, to signal when it had received an acknowledgement and memory could be freed.

The flash memory should be managed by a separate component. This is not as simple, as the component contains some state information that should be saved when the node is rebooted. Apart from that, a simple command which would take a buffer, and write it to the next free flash-page, a command to free memory, and a command to retrieve a specific page, is all that would be needed. The command to write a buffer to flash, should signal an event to tell the application how many pages is left after the write.

Chapter 7

Energy Budget

In this chapter we will discuss the different aspects of the energy budget. We will select and test batteries for use in the experiment, and estimate how long the node will be able to run the final application.

When looking at the energy budget, the first thing to decide is how to power the node. In our case, since the nodes are located on sows, we need some kind of battery to go with it. The literature about batteries differentiates between a cell and a battery[39]. A cell is a single electro-chemical cell, while a battery is a collection of cells. So a normal 1.5 V "battery" is really a single cell, where a 9 V battery is a battery, consisting of 6 cells. In the following we will use these terms.

We have some limitations affecting the choice of batteries. First of all, the cells should be sealed, so that they cannot leak, even when placed upside down. The chemicals inside cells are dangerous, and we do not want hurt the sows. Secondly, the weight of the battery should be reasonable, as the sow will have to carry the battery in a neck-collar. Thirdly the output voltage of it must stay above 3 V, until the battery is completely depleted. We need this as most batteries have a declining voltage during discharge, and we have seen the node misbehave when powered by a 3 V transformer.

7.1 Battery Choices

The literature[39] divides batteries and cells into two groups, primary and secondary. Primary cells are use-once or disposable cells, while secondary cells are rechargeable.

We limit ourselves to secondary cells. We will have to perform several experiments with them, which makes primary cells an impractical and a more expensive choice. Of the secondary cells, the 4 most readily available to the average consumer are:

- Lead-Acid
- Nickel-Cadmium (NiCd or NiCad)
- Nickel-Metal-Hydride (NiMH)
- Lithium-Ion (Li-Ion)

The Lead-Acid cell is one of the oldest types of secondary cells. Its most popular use is as car batteries, to drive the starter-motor. This is because one of the properties of a Lead-Acid cell is that it is well suited to provide a high load. Another characteristic property is the low energy to weight ratio. Lead-Acid cells are also difficult to manufacture in small sizes, and these are not readily available.

Table 7.1 – *Comparison of cell types[14]*

Cell Type	Nominal voltage	Energy capacity per kg	Self discharge per month	Cycle life
Lead-Acid	2.0 V	30 – 50 Wh/kg	5%	200 – 300
NiCd	1.25 V	45 – 80 Wh/kg	20%	1500
NiMH	1.25 V	60 – 120 Wh/kg	30%	300 – 600
Li-Ion	3.6 V	110 – 160 Wh/kg	10%	500 – 1000

The self discharge of NiCd and NiMH are highest immediately after they have been fully charged. Part of the self discharge of Li-Ion batteries stems from the protection circuits.

The Nickel-Cadmium cells are most often encountered as a rechargeable replacement for primary cells in our household appliances. They are available off the shelf in R03, R6, R14 and R20 form factors (also known as AAA, AA, C and D), and are inexpensive. The energy to weight ratio is low, but better than Lead-Acid. Contrary to Lead-Acid cells, NiCd cells experience a so called memory effect, where a shallow discharge[1] will lower the capacity of the cell. It is possible to reverse this effect though a couple of full discharge cycles.

The Nickel-Metal-Hydride cells, are just as the NiCd cells, sold as a rechargeable replacement for primary cells. In recent years they have almost completely replaced the NiCd cells, because of their higher capacity. Also, they do not contain the toxic Cadmium as NiCd cells does. The NiMH cells also shows the memory effect, but in a lesser degree than NiCd cells.

The Lithium-Ion cells are known from notebook computers and cell-phones. Li-Ion cells have both the best energy to weight and the best energy to volume ratios of the cells described here. Another important property of Li-Ion cells is their high nominal voltage at 3.6 V. This makes it possible to drive modern electronics directly without a boost converter using a single cell. Also Li-Ion cells does not exhibit any memory effect, making them easier for the consumer to use. The disadvantage of Li-Ion cells are their high price, and the fact that they need protective circuits to prevent both deep-discharge and over-charging. Either of these events can cause overheating inside the cell, potentially leading to explosion.

A sibling to the Lithium-Ion cells are the Lithium-Ion-Polymer cells. Their properties resemble the ordinary Lithium-Ion cells, but it is possible to manufacture them in almost any form. The price of this flexibility is a little loss in the energy to weight ratio. However they are obvious candidates for cell-phones, and other devices that need high capacity, but has little space for cylindrical cells.

Table 7.1 lists the most important properties of these four types of cells. As can be seen from this table, Li-Ion cells are best in every category. However Li-Ion cells are very expensive, and the additional protection circuit also adds to the price. Furthermore the cells are not easily obtainable to consumers. Lead-Acid batteries are difficult to buy in sizes that are small enough, that we can use them for this experiment. Both NiCd and NiMH cells are readily available, but we choose to use NiMH cells for our experiment, because of their higher capacity.

We choose three different cells, from two manufacturers. One from Panasonic called HHR210A, and two from Ansmann with a capacity of respectively 2300 mAh

[1] A shallow discharge is a discharge which does not deplete the battery completely, before it is recharged again.

Table 7.2 – *Comparison of experiment cells*

Cell	Nominal Voltage	Manufacturer Capacity		Weight
Panasonic HHR210A[47]	1.2 V	2080 mAh	2496 mWh	29 g
Ansmann 2300 mAh[5]	1.2 V	2300 mAh	2760 mWh	28 g
Ansmann 2400 mAh[6]	1.2 V	2400 mAh	2880 mWh	30 g

and 2400 mAh. Table 7.2 lists the properties of the different cells. We will use 4 cells to power the nodes, as this will give us a nominal voltage of 4.8 V, and a voltage of 4 V when the batteries are close to completely depleted.

7.2 Battery Experiments

To validate our energy budget we wish to perform a simple test to see if we can validate the capacity claimed by the manufacturer.

When listing the capacity, the manufactures use the C-rate to specify how the capacity of the cell have been tested. Discharging a cell or battery at a C-rate of 1 C, means that it will be depleted in an hour. Likewise discharging a battery at 0.2 C (also written as C/5) means that the battery will be depleted in 5 hours[39]. During the discharge, the current drawn from the battery is constant.

7.2.1 Discharging the Batteries

The datasheet capacity for the 3 different cells we use, are all given in 0.2 C. The rate of discharge will affect the amount of energy the cells can provide[5, 6, 14], and a higher discharge rate will result in a lower capacity for NiMH cells[5, 6, 19]. To replicate the capacity the experiment, we have to create a load that will deplete the cells in 5 hours, which requires that the current drawn is approximately 400 mA.

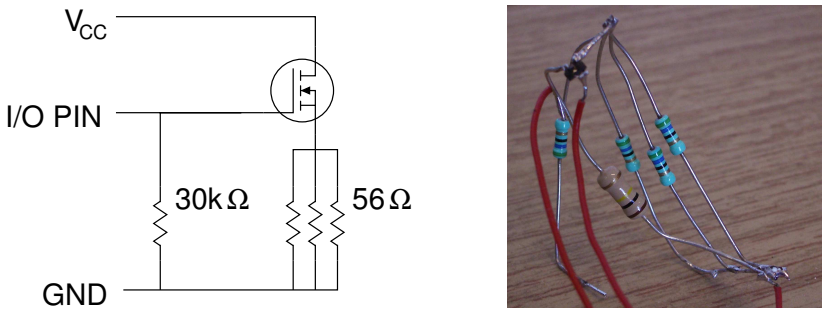
To perform this discharge, we need a way to draw a constant current from the cells. We also need a way to stop the experiment once we have depleted the cells. Otherwise we risk damaging them, making it difficult to get repeatable results when performing several tests in a row with the same cells.

We can use the BTnode for this purpose, as its battery charge indicator can be used to determine when we should stop discharging the attached battery. We simply construct an application which turns on all the LEDs on the node, and samples the battery charge indicator continuously, until the charge drops below a certain level. When this happens we turn off all the LEDs, and put the CPU to sleep, so it uses the least possible amount of energy.

Using the node also solves the other problem. The voltage regulator on the node stabilizes the current drawn from the batteries throughout most of the experiment, even though the voltage output of the batteries decrease.

A modified BTnode, running at full speed, with all 4 LEDs turned on, only consumes approximately 25 mA. At this rate it will take 4 days to discharge a 2400 mAh cell. To increase the current consumption we can attach a couple of resistors which the node can turn on and off. Since the MCU on the node cannot provide enough current to get a high enough current consumption, we use a MOSFET transistor to turn on and off the resistors. The diagram for this circuit can be seen in Figure 7.1.

Figure 7.1 – *Diagram and picture for the resistor circuit*



The circuit was built as a birds nest, as it was simple enough for this.

Three 56 Ω resistors in parallel will have a resistance of approximately 20 Ω. These will be provided with 3.3 V, and therefore they will draw 165 mA. The voltage regulator can at most provide a 200 mA continuous output[46], and if we added another parallel resistor, the current drawn by the resistors would be 236 mA.

With the resistors turned on, the node will draw around 140 mA, resulting in a rate of 0.06 C (or C/15), allowing us to deplete the cells in approximately 15 hours. This is relatively far from what the batteries have been tested with, but we cannot reach 0.2 C with a single node, because of the limitations set by the voltage regulator.

7.2.2 Capacity Results

For the experiment, we use two digital multi-meters, which continually measure the voltage of the batteries, and the current drawn from them. The multi-meters output the measurements 1 – 2 times per second, and a PC logs this data with a time-stamp in two files.

The test is performed three times on a 4 cell battery, and we continue discharging until the voltage of the battery is 4 V. We stop at this point, because the cells can degrade, if discharged below 1 V. Before the first test, the batteries were discharged using the discharge feature in the battery-charger. Once properly discharged they were charged, and left in the charger for at least 24 hours, to trickle charge.[2] After the 24 hours, the experiment was started. On the second and third run of the experiment, we did not discharge the batteries, as they already had been completely discharged during the previous experiment.

The results of the different tests can be seen in Table 7.3 along with the manufacturers specifications. For all three cells the capacity listed by the manufacturer comes pretty close to the capacity we measure. However as we discharge the batteries slower than the manufacturer, we would have expected to exceed their claims a bit. One interesting thing to note, is that we measure the capacity of the Ansmann 2300 mAh cell to be higher than the Ansmann 2400 mAh.

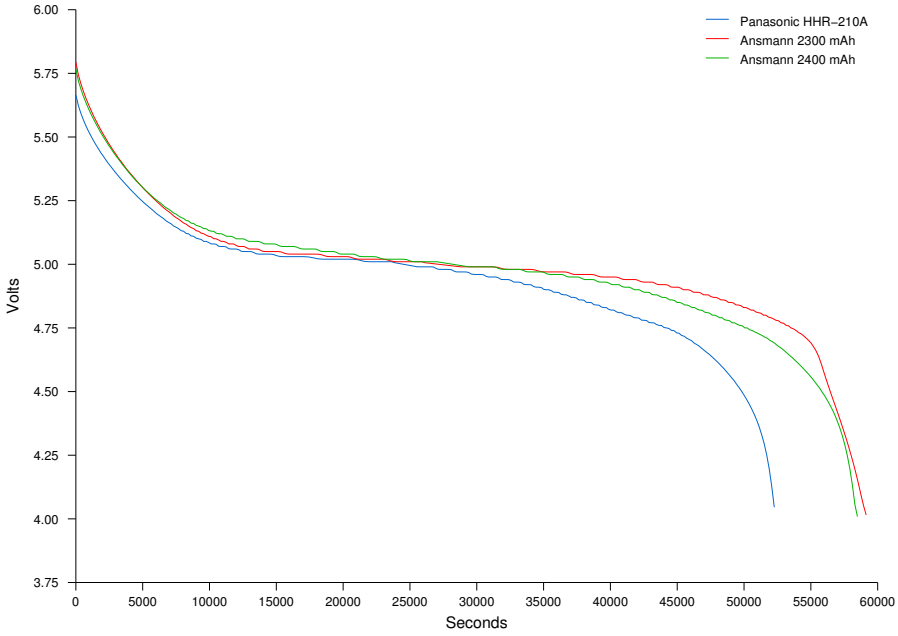
Discharge curves from a single discharge of the 3 batteries can be seen in Figure 7.2. The discharge curves for all of the batteries are close to the ones in the datasheets. The voltage is declining slowly after the initial quick drop, until the bat-

[2] When a battery is trickle charged, it is charged at a low rate (usually around 0.05 C) in order to compensate for the self-discharge. By trickle charging the battery for at least 24 hours, we ensure that it have reached its maximum capacity[14].

Table 7.3 – *Result of the capacity experiment*

Cell	Capacity in mAh				
	Run 1	Run 2	Run 3	Avg.	Datasheet
Panasonic HHR210A[47]	2026	2033	2023	2028	2080
Ansmann 2300 mAh[5]	2291	2280	2291	2287	2300
Ansmann 2400 mAh[6]	2263	2273	2282	2273	2400

Figure 7.2 – *Discharge curves for the 3 tested batteries*



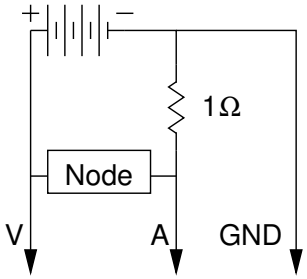
teries are almost completely depleted, where it again drops rapidly. This is really the best case scenario for us, as it will allow us to use the batteries for the longest possible period.

A major problem in testing the capacity this way, is that the load on the batteries does not resemble the load they are going to experience during the experiments. However it still gives us a pretty good idea of what to expect from the batteries, and especially the very sudden drop in voltage when the batteries are depleted, shows us that the node will be able to function until it has used all the energy in the batteries.

7.3 Energy Consumption of the Node

As mentioned in Section 4.2, we started by lowering the energy consumption of the node in idle state as much as possible. And in Section 5.2 we described how we could use the power management features of the MCU, to reduce the energy consumption as much as possible. We now want to estimate if the energy consumption is low enough that we can expect the node to last the 20 days of the experiment.

Figure 7.3 – *Diagram of how the node and battery is connected to the PCI-1202H*



7.3.1 Measuring Energy Consumption

The voltage regulator on the BTnode, ensures that the current drawn remains almost constant, even when the battery voltage changes. Therefore we will focus on the current consumption for our estimations. Using the power consumption will lead to wrong results, as the power consumed will change as the batteries are depleted.

To measure the current consumption of the node accurately, we need to obtain measurements at a high rate, or use an analog filter. The analog filter requires knowledge of analog electronics, so we choose to measure at a higher rate. To catch all the changes in the node's current consumption we need something that measures with a rate of at least 40 Hz. This is because the startup time for the digital accelerometer is 50 ms, and if we want to be able to see the changes we need to sample with twice the rate, according to the Nyquist-rate[21].

The digital multi-meters are unable to measure at this rate. Therefore we use a Data Acquisition card (DAQ) from ICP DAS called PCI-1202H. This PCI card is able to sample at a rate of 44 kHz. The card measures the voltage, so to measure the current drawn by the node, we need to place it in series with a 1 Ω resistor. See Figure 7.3 for a diagram of the setup. Since we can measure the voltage drop across the resistor, and we know the resistance, we can use Ohm's law[61] to find the current:

$$U = RI \Rightarrow I = \frac{U}{R}$$

When R is 1 Ω, the current equals the voltage drop. From Kirchhoff's junction rule[61], we can conclude that the current passing through the resistor equals the current that passes through the node. This is a very simple way to create an amperemeter. But as the nodes current consumption is in the mA range, the voltage drop we measure is going to be in the mV range. The ICP-1202H can measure in different voltage ranges, one of which is 0 V – 10 V. A pre-scaler on the board can be configured to a gain of 100. In this configuration, the board can measure from 0 to 100 mV (i.e. in our case 0 to 100 mA). As the ADC on the board has a 12-bit resolution, this means that it voltage is measured in steps of 0.0244 mV. This should be good enough to support our needs.

The PCI-1202H comes complete with drivers for Linux. Unfortunately the version of these drivers (version 0.6.5) that were available when we got the board was for the 2.4 version of the Linux kernel, and we were using the 2.6 version. Porting the driver was straight forward. As we started to use the board, we discovered that the programs communicating with the board often failed, if they were

Table 7.4 – *List of current consumption states for the final application*

Task	Duration	Current drawn	Frequency
Sleeping	N/A	0.47 mA	N/A
Sampling/storing data	27 ms	6.41 mA	4 times per second
Offloading data	36000 ms	52 mA	Once per hour

scheduled by the kernel in an unlucky way. To get around this, we had to move some of the measurement code from user-space to the kernel. After this the board was stable enough that we could use it for our experiments. The source code for the kernel-driver, and a simple user-space library to use the driver is available from http://hogthrob.42.dk/ixpci.

7.3.2 Current Consumption Estimations

Now that we have a method to accurately measure the current consumption of the node, we can proceed to estimate the lifetime of the node. To do this we have divided the application into separate states. The states and their associated current consumption can be seen in Table 7.4.

The sleeping state is what the node is in whenever it is not doing anything else. The Sampling/storing data state is when the node is reading from the accelerometers, and storing the data in flash. This really should be two states, but obtaining a current estimation for the combination was simpler. Offloading data includes turning on the Bluetooth module, waiting for a connection, and offloading the data. We have combined these steps into one, as it was difficult to get a repeatable measurement of the current consumption for the separate steps. The duration is set to 2 times 18 seconds. The 18 seconds can be split into 2 seconds for powering on the Bluetooth module, 10 seconds waiting for the PC to connect, and 6 seconds to offload the data. We use twice the 18 seconds to get a conservative estimate. The 6 seconds to offload the data is an estimate obtained by running the finished application, and measuring the offload time. As an estimate of the current consumption for this combined state, we have used the maximum current consumption the module can draw according to the datasheet[20]. It is a conservative estimation, but we would rather estimate a too high energy consumption than a too low.

From these values we can calculate the estimated average current consumption, while not sending:

$$\frac{0.47\text{ mA} \times (1000\text{ ms} - 4 \times 27.5\text{ ms}) + 6.41\text{ mA} \times 4 \times 27.5\text{ ms}}{1000\text{ ms}} = 1.12\text{ mA}$$

So without offloading the data, the node will use on average 1.12 mA. When we include offloading this becomes:

$$\frac{3600\text{ s} \times 1.12\text{ mA} + 36\text{ s} \times 52\text{ mA}}{3600\text{ s}} = 1.64\text{ mA}$$

So in total 1.64 mA on average, or 39.36 mAh per day. With 2100 mAh available the nodes should last more than 53 days. However we will not have 2100 mAh available because of the self discharge. As the self discharge is rather high for NiMH batteries (30% per month), a conservative estimate will be that we only have $\frac{2}{3}$ of the capacity available, i.e. 1400 mAh. This would give us an expected lifetime of 35 days, which is still 15 days more than needed for the experiment.

A problem with this estimation is that it does not take failed offloads and resends into account. The effect of not discharging the battery like in the capacity test, is also ignored. To get a better estimate of how the failed offloads and resends will influence the current consumption, we measured the current consumption of a single node checking into a single PC for 3 days. The average current consumption over these three days was 1.27 mA, somewhat lower than the previous estimate. This is most likely because the previous estimate is very conservative, especially with regard to the power consumption of the Bluetooth module. With an average current consumption of 1.27 mA the node should last almost 53 days, including the self discharge of the battery.

7.4 Summary

In this chapter we have selected the cells that we wish to use for the experiment. We have shown that the claims made by the manufacturers about the capacity of the cells are correct.

We have also measured the current consumption of the node in different states, in order to estimate the lifetime of the node. From these estimations it is clear that most of the current consumption comes from the basic sampling of the accelerometers. If we were to lower this energy consumption further, we would have to either remove the voltage regulator, replace it with one that performs better, or change the sample rate.

Our estimation based on measuring the node for 3 days, is that the node will have a lifetime of at least 53 days. However there are two problems with this estimation:

- The distance between the node and the Bluetooth receiver was less than 1 m, as they were both attached to the same PC. The distance between the node and the PC's Bluetooth module could very well affect the current consumption of the node during the offloading phase.
- Only one node is present during the experiment. If two or more nodes try to offload data at the same time, it will affect the amount of time the Bluetooth module is turned on, as the PC only offloads data from one node at a time. This will again result in a higher current consumption than actually measured.

While omitting these issues from the measurements will cause a lower current consumption estimate, our lowest and extremely conservative estimation conclude that the nodes will be able to last 15 days more than the 20 days of the experiment. So the conclusion must be that the nodes should last throughout the experiment.

Chapter 8
Field Experiment Setup

In this chapter we will describe how the experiment is setup in the stables. For the experiment we need the following:

- A way to distinguish the sows from each other.
- A way to protect the nodes from the environment in the stables.
- Positions for the 4 cameras, so that they cover as much as possible.
- To find a place for the PC's, which we need to capture video from the cameras and to offload data from the node.
- A position for the Bluetooth modules, so that the nodes can connect to them.

As previously mentioned in Section 3.1, the experiment will be performed from the 21st of February to the 21st of March. During the experiment we will attach nodes to 5 previously selected sows, to measure their activity.

8.1 Sow Marking and Node Pairing

We need a way to be able to tell the 5 sows from each other. This is needed both for the manual heat detection, and to tell the sows apart in the video data. Therefore a distinctive blue mark is sprayed on the back of each sow, which is large enough that it can be spotted in the video data. Also when we attach the nodes to the sows, the marking on the back is noted, together with the last 4 digits of the node's Bluetooth MAC address. The relation between nodes, sows and marks can be seen in Table 8.1.

8.2 Nodes

We need to attach the node and batteries to the sow, and we need to do it in a way that both protects the node from the environment in the pen, and prevents the sows from harming themselves by eating the chemicals in the batteries.

In consultation with KVL, we decided to use a slightly curved box, so that it would fit to the curve of the sow's neck. Centralværkstedet at the H.C. Ørsted institute manufactured the box, which can be seen in Figure 8.1. These boxes are the single most expensive part of the experiment, costing approximately 2000 DKK a piece.

To fasten the node inside the plastics box we attached Velcro-tape to the bottom of the node, and to the inside of the box. The accelerometer board was attached to the node using double sided carpeting tape. The antenna for the Bluetooth radio is

Table 8.1 – *The nodes and markings used on the 5 experiment sows*

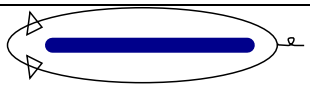
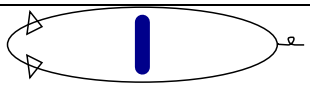
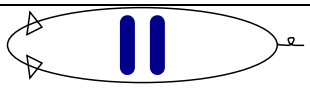
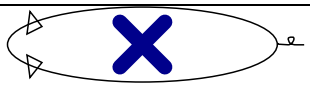
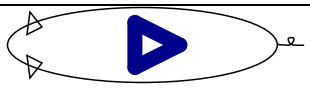
Sow	Node	Marking
Sow 1	4EBC	
Sow 2	4D08	
Sow 3	4EBF	
Sow 4	4D08	
Sow 5	4D09	

Figure 8.1 – *The box housing the nodes and batteries during the experiment*

Figure 8.2 – *The node, fitted with accelerometer board and Velcro*

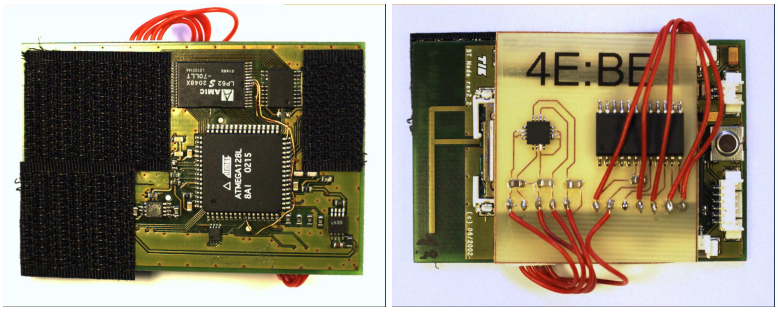
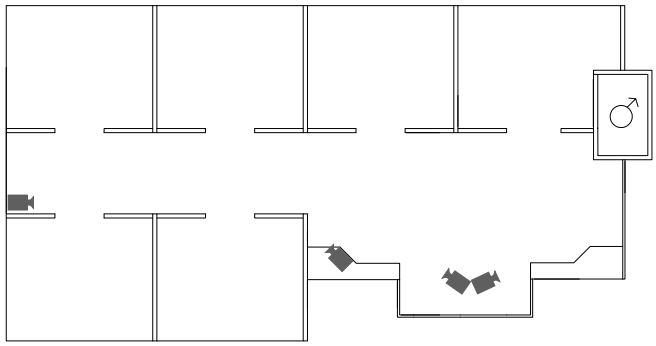


Figure 8.3 – *Position of the 4 cameras*



integrated on the node, eliminating the need for an external antenna. Furthermore by using plastic for the box, we do not hinder the wireless communication. A picture of a node, with the Velcro and accelerometer attached — ready for deployment — can be seen if Figure 8.2.

The batteries needed to power the node, also needs to be fixated inside the box. Fortunately the spring loaded battery holders we acquired fit perfectly inside the box, eliminating the need for further fixation.

Before the deployment we ensured that the sows were not able to break the box, by giving the box to the sows to play with. Small teeth-marks were all the damage they were able to do.

8.3 Cameras

We need cameras to establish the ground truth for the experiment (see Section 3.2.1). Therefore we installed the 4 cameras in the pen, and placed them so that they cover the areas where the sows are active. The cameras cannot cover the entire pen, but the areas that are not covered by the cameras are primarily the places where the sows are sleeping.

We use 4 AVC301A, which are black and white cameras with a composite output. The cameras allows coaxial cables to be connected, making it possible to use long cables without any noticeable effect on the image quality. Two of the cameras have a wide angle lens, the other two have a normal lens.

The position of the four cameras can be seen in Figure 8.3, and the installation of them can be seen in Figure 8.4. Figure 8.5 contains a picture from each of the four cameras. Note that one of the experiment sows can be seen in camera 1.

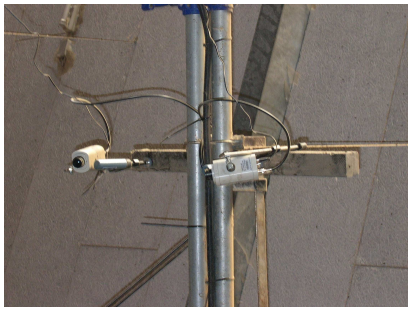
We had to get both power and the coaxial cable to each of the cameras. Fortunately a bar down the middle of the pen, used to cool the sows with water during the summer, could be used to hold the cables out of reach of the sows.

The cable from the cameras was split close to the two PC's, so that we could record the video on both PC's at the same time.

Figure 8.4 – *The 4 cameras*

(a) Camera 1

(b) Camera 2



(c) Camera 3 & 4

Figure 8.5 – *The view from the 4 cameras*

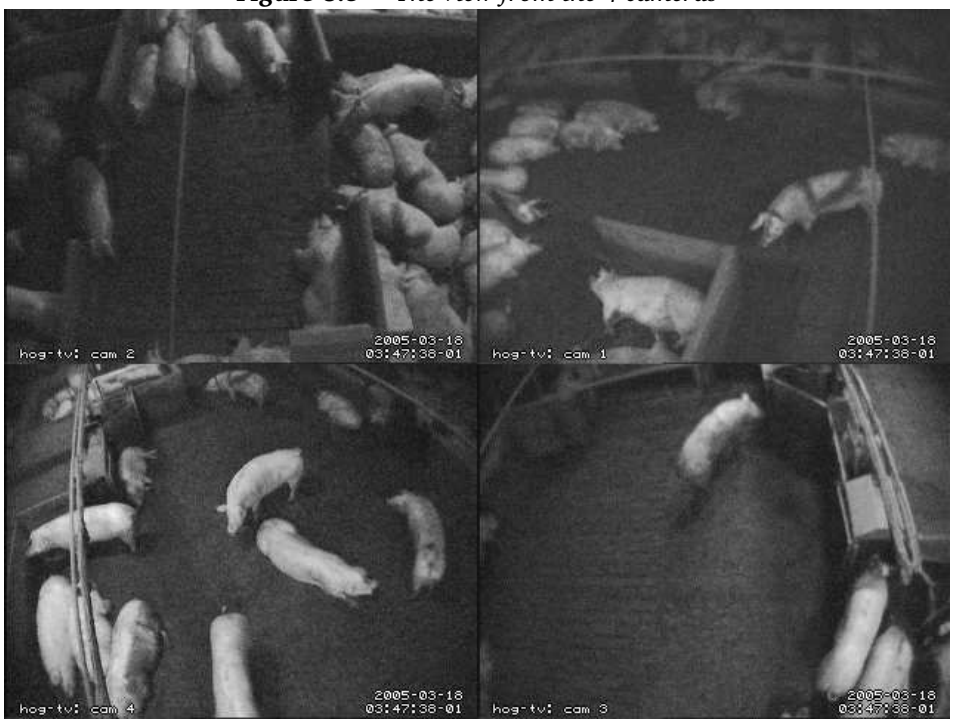


Figure 8.6 – *The container placed outside the stables*

8.4 Servers

The two servers have been installed by Eske Christiansen, and are almost completely identical. Both machines are fitted with 4 Pinnacle Studio PCTV Rave frame-grabber cards, and installed with Gentoo Linux. The open source program Motion[1] is used to grab video frames from the cards, and create video streams for offloading.

The two machines are connected to each other through a 100 Mbit/s network, and they are connected to the Internet through a 2048/512 kbit/s ADSL line, available at the farm.

Both of the servers run Motion all the time, so that all video data is gathered on both machines. One of the PC's offloads the video data through the Internet to a server at DIKU. A heartbeat is used to determine which of the machines should handle the upload. The offload is primarily a way to backup the data, but we also need it to make the videos available to the people at KVL, during the experiment.

The PC's also handles the offloading of data from the nodes. The PC that first discovers a node, is the one that handles the offload. The PC's continually syncs the offloaded node data between them, so that both has all data collected from the nodes. This data is also backed up at DIKU through the same mechanism as the video data.

Because of the harsh environment inside the stables, we decided to place the PC's outside the stables in a container, see Figure 8.6. This has the added benefit that the PC's can be installed and accessed during the experiment without having to enter the pen.

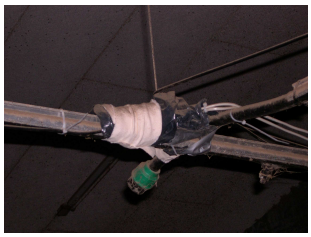
8.5 Bluetooth

To handle the Bluetooth communication we acquired two Sitecom CN-502, which are USB Bluetooth modules, with an external antenna, and a claimed range of up to 100 m.

Before we began installing equipment for the experiment, we tested the range of the Bluetooth module, by fitting it to a laptop, placed on one of the feeding stations in the pen at the corner of the pen. A packaged node in its protective box (see Section 8.2) programmed to check-in twice a minute, was then placed in all the

[1] Available from: `http://www.lavrsen.dk/twiki/bin/view/Motion/WebHome`

Figure 8.7 – *The Bluetooth modules in the middle of the pen*



corners of the pen. At each corner we made sure that a connection could be established and that data could be offloaded. This test did not reveal any communication problems. We decided to place the Bluetooth modules in the middle of the pen, thus halving the maximal distance between the nodes and the modules, just to be on the safe side.

Placing the Bluetooth modules in the middle of the pen, presented a new problem. Most commonly available USB cables are no longer than 5 m. Since the pen is approximately 12 m $\times$ 23 m large, we will need at least a 6 m cable, just to get from the wall of the pen to the middle, and even longer to reach the servers in the container. One could use USB hubs to extend the reach, but this will only buy us 5 m, per hub. Another solution is to use an USB Extender. An USB Extender consists of two small boxes, which can be connected to each other using standard Ethernet cables. One of the small boxes is attached to the USB port of the PC, while the Bluetooth module is connected to the other small box. The Ethernet cable between the two boxes can be up to 50 m long, so this is sufficient for our use.

With this extender we can attach the Bluetooth modules to the same bar that we use to carry the wires for the cameras. The finished installation is protected by a plastic bag and duct tape, and elastic bandages. It can be seen in Figure 8.7.

Chapter 9
Field Experiment Results

The video recording ran from the 20th of February to the 21st of March. The nodes were deployed the 28th of February, with as freshly charged batteries as possible. During the experiment we gathered approximately 240 MiB of raw data from the nodes, resulting in almost 21 million extracted samples. From each of the 4 cameras we collected between 4.5 GiB and 5.5 GiB of video data. In total this amounts to approximately 20 GiB of data, which was transfered to DIKU during the experiment.

During the experiment, we encountered check-in problems, nodes that rebooted unexpectedly, and various problems with the servers. We also had to replace a single node, because its Bluetooth module fell off.

In this chapter we will discuss these problems. We will also describe the algorithms used to extract the gathered acceleration measurements, and try to validate these measurements in different ways. We will look at the lifetime of the nodes, and compare this to our previous estimates. Lastly we will look at which lessons we can take with us to future deployments.

9.1 Results of Manual Heat Detection

As described in Section 3.2.1, manual heat detection was carried out during the experiment, so that we would know the exact heat-period of the 5 sows.

Table 9.1 lists the heat-period of the 5 sows in the experiment. The heat-period for Sow number 4 did not start until after the end of the experiment, which is why the table does not list a heat-period for her. Later, in section 9.6, we will take a cursory look at how these heat-periods relates to the activity data gathered by the nodes.

Table 9.1 – *The manually detected heat-periods*

Sow	Node	Start of Heat		End of Heat	
		Date	Time	Date	Time
Sow 1	4EBC	10/03	10:30	12/03	17:40
Sow 2	4D08	14/03	10:30	17/03	10:30
Sow 3	4EBF	16/03	02:00	17/03	17:30
Sow 4	4D0B	—		—	
Sow 5	4D09	14/03	10:30	16/03	17:30

9.2 Problems With Node Check-in

The first problem we discovered during the experiment was that the nodes would not check-in regularly, resulting in data-loss. However both servers were working correctly and this was not something that we had experienced in our test setup.

Looking at the logs from the check-in program on both servers, it became apparent that in many cases the PC's had to connect to the nodes at least two or three times in order to complete the offload. Sometimes even more connections were required.

The offloading problems seen in the logs on the PC's, could explain the irregular node check-in. The back-off algorithm on the node, as described in Section 6.5.1, was designed so that it would take a successful offload to make the node exit the back-off mode. This was a deliberate choice, as it would prevent the nodes from depleting their batteries, if something went wrong with the PC side of the communication.

Instead the back-off algorithm was making it very hard for the node to complete a full offload. Therefore we decided to change the algorithm, even though it meant that we had to collect all the nodes at the farm to reprogram them. We changed the back-off algorithm so that it went back to the 10 s interval, just as soon as a PC had connected to the node. This meant that no matter how many tries it took to do a complete and successful offload, it would still happen within a reasonable amount of time.

We reprogrammed the nodes the 11^{th} of March, or on day 18 of the experiment. While re-programming the nodes, we powered them with a replacement battery, so that the reprogramming would not affect the energy budget.

In Figure 9.1, the percent of gathered data per day, throughout the experiment is shown. The horizontal line at day 18, is the point where the nodes were reprogrammed. For some of the nodes, i.e. 4D09, 4D0B and 4EBC the change of back-off algorithm seems to have made a difference for the better. For 4EBF the amount of gathered data seems to have gone down.

The last node has a total dropout of data at day 21, leading us to replace it with a spare at day 22 (the 15^{th} of March) around 11:20. When we inspected the malfunctioning node, we found that the Bluetooth module had fallen off the node. This is likely a side effect of the reprogramming, as we have to be rather hard handed when removing the battery pack from the box protecting the node. During the reprogramming, the clamp holding the Bluetooth module have come loose, causing the module to fall out a couple of days later.

While the change in the back-off algorithm seems to have had some affect, it by no means explains why the problems offloading the data happens in the first place. As explained in Section 8.5, we had tested that the nodes were able to offload data without problems, over twice the distance in the same pen as the experiment was carried out in. We believe that the problem is caused by the sows sleeping on top of the nodes. When this happens the range of the Bluetooth communication is shortened, causing problems for the offloading algorithm. This explanation have been verified by comparing the video data with the periods of missing data.

Figure 9.1 – *Percent gathered data per day of the field experiment*

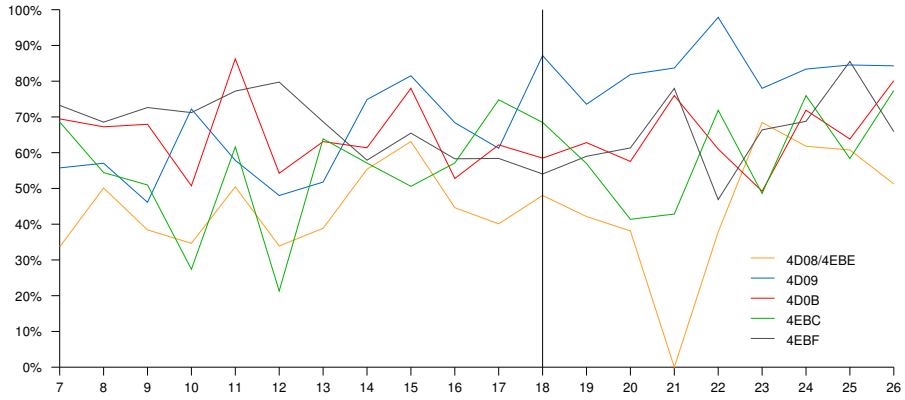


Table 9.2 – *The number of times the different nodes rebooted*

Node	Reboots
4D08	4
4D09	10
4D0B	22
4EBC	9
4EBE	0
4EBF	10

9.3 Unexpected Node Reboots

On the 2nd of March we discovered that one of the nodes had rebooted during the night. We had not seen this behavior in our test setup. Later it also happened to other nodes, but it did not happen often, at most once each day. Table 9.2 shows the number of times each of the nodes has rebooted.

The reboot was discovered, because the node started checking in samples numbered from zero again. Such a reboot should be easy to discover and correct when extracting the data after the experiment, and the data loss from it is small. Therefore we did not take steps to correct the problem.

When we performed the data extraction after the experiment, we discovered at a single node never rebooted. This was the spare node we deployed when the Bluetooth module came loose in one of the other nodes. This node used a custom-made battery-pack where 4 NiMH cells were welded together. We therefore concluded that the reboots were caused by the connection between the cells failing, because the springs of the battery packs was not tense enough.

9.4 Server Problems During the Experiment

Several times, the Bluetooth stack on one of the PC's stopped working correctly, with no apparent reason or anything in the log files. Either the Bluetooth module would never discover any other Bluetooth devices, or we would start getting errors back from operations that should work.

To correct this, we stopped the offloading program, removed the Bluetooth kernel-modules, and reloaded them. Usually this procedure corrected the problem. In a single case the server crashed, and was then manually rebooted. In the period from the crash to the reboot, the other server handled check-in and the gathering of the video data, as it should.

Another problem with the servers was that the time synchronization between the two servers did not work as expected. Both servers used NTP[1] to synchronize their time, but at the end of the experiment they were at least 10 seconds apart. Unfortunately this was not discovered during the experiment, so we do not have any explanation as to what caused this. Also we do not know if there was a 10 seconds difference all the time, or if the machines in periods had matching time.

9.5 Data Extraction

We have to extract the acceleration measurements from the collected raw data sent by the nodes, to a format that the people from KVL can import into their analysis tools. We will start by describing the simple algorithms that we originally expected to use, and will then go into detail about how the problems during the experiment affects these algorithms. Lastly we will present the final solution.

9.5.1 The Original Extraction Algorithm

We originally expected to be able to extract the data, by performing the following steps:

1. Merge the data from the two servers.
2. Convert the sample numbers to wall clock time.
3. Output the data as a .CSV file.

Merging the data from the two servers, is easily done. The sample number contained in each data packet (see Section 6.2 for details), can be used to uniquely identify the data packets. So if a node have offloaded half of its data to one PC, and then offloaded the whole to the other PC, we only use the page from one of the machines. Which one is used will not matter as they are identical. The packets that only exists on one of the servers is also included in the merge.

To convert the sample number to wall clock time, we must first recap how the time synchronization between the node and the PC is done. In Section 6.1 we decided that the time synchronization works by the PC time-stamping the initial packets. These initial packets contains the nodes current sample number. From these pairs of time stamps and sample numbers, we can calculate the average time between two consecutive samples using the algorithm in Algorithm 9.1.

With the result from this algorithm we can find the time when the first sample was measured ($epoch$), by looking at the information from the first initial packet (fip):

$$epoch = fip.server\_time - fip.sample\_number \cdot SecsBetweenSamples()$$

From this epoch, we can calculate the time when any sample was obtained:

$$sample\_time = epoch + sample\_number \cdot SecsBetweenSamples()$$

[1] Network Time Protocol

FINDING THE AVERAGE TIME BETWEEN TWO CONSECUTIVE SAMPLES (9.1)

```
1:  Input: Time-sorted list of all initial packets for both servers (init_packets)
2:  Output: The number of seconds between two consecutive samples
3:  SECSBETWEENSAMPLES()
4:      prev_sample = init_packets[0].sample_number;
5:      prev_time = init_packets[0].server_time;
6:      secs_between_sample_sum = 0;
7:      count = 0;
8:      for (i = 1 ; i < length_of(init_packets) ; i++) {
9:          sample_diff = init_packets[i].sample_number - prev_sample;
10:         time_diff = init_packets[i].server_time - prev_time;
11:         secs_between_sample_sum += sample_diff / time_diff;
12:         count++;
13:         prev_sample = init_packets[i].sample_number;
14:         prev_time = init_packets[i].server_time;
15:     }
16:     Return secs_between_sample_sum / count;
```

Now that we know how to merge the data from the two servers, and how to calculate the time for a specific measurement, we can extract all the data. We take the merged data, and sort it according to sample number. For each packet, we read the sample number. We then output all the samples, together with the current sample number. The sample number is incremented each time we have read both an analog and digital measurement, or when two analog or two digital measurements occurs after each other.

9.5.2 Taking the Experiment Problems into Account

The procedure described in the previous section could have been used if the nodes had not rebooted during the experiment, and if the time synchronization between the two servers had worked. The worst of these two problems is the node reboot, because if the nodes had not rebooted, we could simply have discarded the timestamps from one of the servers, using the other for all time calculations.

To extract the data in the face of these problems, we will have to perform the following tasks:

1. Calculate the average time between samples, separately for each server.
2. Find the points in time when the node have rebooted, combined for both servers, and calculate the sample numbers for these points.
3. Adjust the sample numbers, so that each data set appears as it would have if the nodes had not rebooted.
4. Merge the datasets.
5. Output the data as a .CSV file.

Instead of simply merging the two datasets, we now have to perform two new steps, before we can merge them.

To calculate the average time between the samples, we only need to perform a minor adjustment to the algorithm from the previous section. If the current sample

Table 9.3 – *The periods the nodes have gathered data, and the coverage*

Node	First Measurement	Last Measurement	Days	Coverage
4D08	2005/02/28 14:05	2005/03/13 20:31	14	44.607%
4D09	2005/02/28 09:59	2005/03/19 22:47	20	71.756%
4D0B	2005/02/28 10:06	2005/03/19 22:48	20	64.567%
4EBC	2005/02/28 10:28	2005/03/19 21:06	20	56.139%
4EBE	2005/03/15 11:07	2005/03/19 21:56	5	61.910%
4EBF	2005/02/28 10:09	2005/03/19 23:02	20	66.717%

number is lower than the previous sample number, we should skip ahead to the next pair. This will leave the reboots out of the calculation.

The next step is to determine the points in time where the node have rebooted. To do this we traverse all the initial packets, and find the ones where the sample number is less than the previous number. For such a packet, we need to figure out what the time was when the node rebooted. This can be done using the same approach as we use to find the $epoch$ in the previous section. After this is done, separately for both servers, we need to combine the reboot times into a single list. We locate any reboot times from the two lists that are less than 2 minutes apart, and delete one of them. The resulting list is then used to calculate the sample numbers the nodes should have had at the time they rebooted.

Using this list of reboots, we can proceed to the next step, and change the numbering of all the pages. For every page that belongs to a check-in that happened after a reboot, we adjust the number by adding the sample number from the reboot list. After this we can merge the two datasets, using the same merge algorithm as described in the previous section.

9.5.3 Anomalies in the Extracted Data

While the algorithm described in the last section should cover the problems with the time synchronization between the servers and the rebooting nodes, a couple of anomalies in the dataset still show up. After adjusting the sample numbers, three of the nodes have decreasing sample numbers, when the pages are sorted according to the receive time. In all cases the receive time is late the 7^{th} of March, or early the 8^{th} of March. Digging further shows that the decreasing sample numbers happens just after a node has rebooted.

This suggests that one of the servers have been running with a wrong time for a period. This would most likely occur in conjunction with a reboot of the server. However we have not performed a manual reboot around that time, and we do not have any indication of an unassisted reboot. The cause of these anomalies remains unknown. To ensure the integrity of the extracted data, we simply remove the pages that exhibit this problem.

Some properties of the extracted data can be seen in Table 9.3. The “First Measurement” column, shows when the nodes were turned on. The node 4D08 was started later than the others, because the power was not connected correctly, the first time it was attached to the sow. This was also the node, which lost its Bluetooth module after the reprogramming, and was replaced with 4EBE. From the “Last Measurement” column, we can see that the nodes stopped checking in late

on the 19th. We did not remove the nodes until Monday the 21st, but the Bluetooth modules on the two servers stopped working before this happened. As the heat-period was over, we did not try correct this. The coverage varies a lot: The best node has a coverage of approximately 72%, while the worst has approximately 45%. Overall we collected approximately 60% of the data. As previously explained in Section 9.2, the lost data seems to be caused by the sows sleeping on the nodes. Because the sows spend much of their time sleeping, the coverage will depend on where in the stables the specific sow is sleeping. This can explain the large difference in coverage.

9.6 Validating the Collected Data

To ensure that the data collected from the nodes is correct, we want to perform two validations. The first validation will assert that the dataset conforms to our expectations, by comparing the two datasets to each other and to the gravitational acceleration.

The second method will seek to establish a correlation between the dataset and the video data. This will verify that our time-stamping and extraction algorithms work correctly.

9.6.1 Verifying the Correctness of the Dataset

To verify the correctness of the dataset, we must first setup some assumptions about what it should contain. The tests in this section have been devised by Cécile Cornou and Peter Sestoft at KVL.

We expect that the largest influence on the measurements, by far, will be the gravitation. So we can convert the raw data from the 3-axis accelerometer to the corresponding g value. When we have the g values for all three axes, they can be used to create a vector, which will point in direction of the acceleration. Most of the time, it will point directly towards the ground, and have a length of one. The average length of this vector should therefore be close to 1 g.

If this test shows that the data from the digital accelerometer is correct we can proceed to verify the data from the analog accelerometer. For this we have two options:

- We can convert the measurements for each axis to g, and compare the result with the corresponding results from the digital accelerometer.
- We can find the slope and offset, needed to convert the output of the analog accelerometer to the g measurements obtained from the digital accelerometer.

We will focus on the first approach.

To perform these verifications, we need to know how the axes on the two accelerometers correlate to each other. We also need to know how to convert the raw values we obtained in the experiment to g.

As described in Section 4.1.7, the X axis of the analog accelerometer should be identical to the X axis of the digital accelerometer with the sign reversed, and that the Y axis should be the same for both accelerometers.

The conversion to g is different for the two accelerometers. The digital accelerometer is the most simple. The output from each axis is a 12 bit signed integer,

Table 9.4 – *The average length of the acceleration vector for the 6 nodes*

Node	Average	Distance from 1 g
4D08	1.037 g	0.037 g
4D09	0.996 g	0.004 g
4D0B	1.019 g	0.019 g
4EBC	1.016 g	0.016 g
4EBE	0.987 g	0.013 g
4EBF	1.054 g	0.054 g

Table 9.5 – *Comparison of the average of the X and Y axis of the digital and analog accelerometer*

Node	Digital X	Digital Y	Analog X	Analog Y
4D08	0.381 g	0.016 g	-2.434 g	-2.408 g
4D09	0.216 g	0.045 g	-2.513 g	-2.223 g
4D0B	0.357 g	-0.126 g	-5.489 g	-5.215 g
4EBC	0.268 g	-0.092 g	-2.556 g	-2.363 g
4EBE	0.467 g	0.256 g	-2.690 g	-2.227 g
4EBF	0.365 g	-0.070 g	-2.498 g	-2.243 g

where `0x000` corresponds to 0 g, `0xFFF` corresponds to -2 g and `0x7FF` corresponds to 2 g. To convert from the raw value, the following formula can be used:

$$acceleration = 2\ g \cdot raw/2048$$

For the analog accelerometer, the output voltage is 1.65 V, when an axis experiences 0 g, and the output voltage changes with 0.174 V for each g. The voltage is measured as a 10 bit unsigned integer, where `0x000` corresponds to 0 V and `0x3FF` corresponds to 3.3 V. So the conversion formula for the analog accelerometer is:

$$acceleration = \frac{1.65\ \text{V} - 3.3\ \text{V} \cdot raw/1024}{0.174\ \text{V}/g}$$

In Table 9.4 the average length of the acceleration vector from the 3-axis digital accelerometer is listed for each of the nodes. As we can see from the table, the average is very close to 1 g in all cases. This suggests that the measurements from the digital accelerometer are valid.

In Table 9.5, the average for the X and Y axis of the digital and analog accelerometers are listed. From this table it is obvious that either the measurements from the analog accelerometer are wrong, or there is an error in the formula that convert from the raw reading to g.

To test if the conversion formula is correct, we placed a node with the accelerometer board attached, flat on a table. In such a position both axes of the accelerometer should measure 0 g. We then raised the board so that it was perpendicular to the table. Depending on which side the node rests on, we should measure -1 g or 1 g on one of the axes, and 0 g on the other. For this test, we reused a test program that was previously constructed to ensure that all accelerometers returned valid measurements. In all cases, we got the expected results, and tries to place the node so that each axis should measure 0.7 g also gave the expected result.

When we performed the same tests with the application used in the field experiment, the results were not correct. Further investigation revealed a bug in the test application, so that the analog accelerometer was never turned off. Fixing this bug, caused the test application to exhibit the same behavior as the application used for the field experiment.

The problem turned out to be a too short turn-on time. Where it should have been 80 ms, as described in Section 5.1.1, it was only 20 ms. This makes it impossible to convert the measurements to g. It might still be possible to use the measurements for heat detection, as KVL have shown that the analog measurements are in fact correlated to the digital measurements.

The measurements from the digital accelerometer seems to correlate with what we expect. In the next section we will try to further ascertain that the time-stamps on these measurements are correct, by correlating them to the ground truth.

9.6.2 Correlating the Acceleration Data to the Video

To ascertain that the time-stamps on the gathered data from the digital accelerometer data is correct, we will have to correlate the extracted data with the gathered video data.

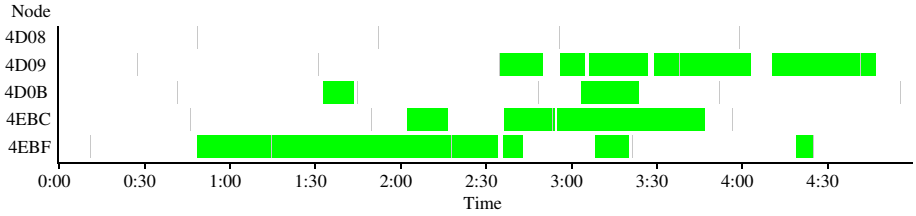
A way to do this, is to manually find periods in the video data where the sow is active. We then need a way to automatically find active and inactive periods from the extracted dataset. Then we can compare the results to see if the two approaches produce the same result.

To find active periods in the dataset, we use the Elastic Burst Detection program, developed at New York University[63]. The program can find bursts of high values in a dataset. To prepare the extracted data for the Elastic Burst detection program, we take the scalar of the difference between two raw measurements from the digital accelerometer. Whenever this scalar is greater than 20, we output a 1, otherwise we output a zero. The burst detection program is then used to find bursts with a sum greater than 600, in a window of 1000 data points. This means that 60% of the values in the window should be ones. These windows will correspond to the sow being active. The window size and threshold are found through experimentation on the data gathered from node 4D09, on the first day of the experiment. We will not use this day for the validation, as this would ruin the objectivity of the validation. Only using a single node for the validation, also means that the method is not calibrated to the other sows. This is on purpose, as it will provide us with a clearer view on how different the data from the sows are.

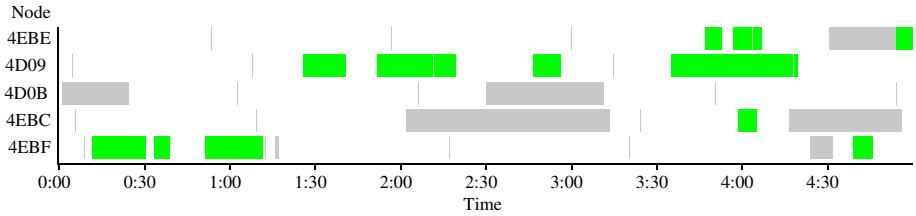
To make sure that all the data can be correlated we have to look at data from late in the experiment, where the chances of the time being wrong are highest. To ensure that our method actually works, we will also look at the beginning of the dataset, where we are sure the time is correct. Therefore we will concentrate our efforts on the 1st of March, and the 18th of March. Both of these days are outside the heat-period for all the sows.

Describing the sows movements from the video data is a time-consuming process. We will therefore only look at a 5 hour window from 00:00 to 05:00. The reasoning behind this choice is that the auto-iris function in the cameras does not work very well, so when the sun is up, the video is over-exposed, making it hard to discern the markings on the back of the sows. The sows are active during the night, so the chosen period should not present a problem.

Figure 9.2 – *Activity plots from 0:00 to 5:00*



(a) The 1st of March



(b) The 18th of March

The green areas are the periods where the sow is active, and the grey areas are the periods where the data is missing.

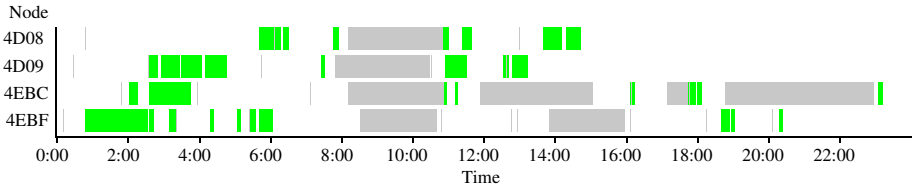
In Figure 9.2, the activity plots from the 1st and 18th of March can be seen. Table B.1 and B.2 in Appendix B lists the activity of the 5 sows in the experiment. A ↑ in the table means that the sow was seen standing. The sow is standing until a ↓ is plotted in the table. The ↓ means that the sow is laying down. The last symbol ⊗, means that the sow walked out of view. We should not expect to be able to find exact matches between the plots and the tables. However we should be able to see if the walking periods marked in the tables, are in the vicinity of the active periods on the plots.

If the 1st of March is compared, we can see that most of the activity periods in the tables matches activity periods in the dataset reasonably close. However, one of the nodes, `4D08`, does not have any active periods at all, neither in the table nor in the plot. This is most likely because the sow is sleeping during this period. If the period from 5:00 to 6:00 is observed on video, which is just outside the period covered by the graph, it can be seen that the sow starts to move around. This matches with the output from the burst detection code.

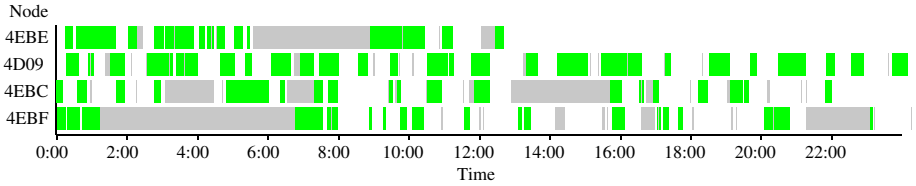
For the 18th of March there is very little activity in the plot. The data for nodes `4D08`, `4D09` and `4EBF` matches the graph. However for `4D0B` there is no active periods, even though there should be some according to the video. The same is true for `4EBC`. This can either be caused the time synchronization not working correctly, or by the fact that the activity detection method is not calibrated correctly, and therefore leaves out periods of activity. As 3 of the 5 nodes match, we conclude that the time synchronization works as expected.

As a cursory look at heat detection, we have included the full day activity plots from the 1st of March, and from a single day in the middle of the heat-period, for each of the sows that experienced heat during the experiment in Figure 9.3. From

Figure 9.3 – *Whole day activity plots for the 4 sows that experienced heat during the experiment*



(a) 1st of March



(b) A day in the middle of the heat-period

these graphs, it looks as if the sows are more active during the heat-period, than on a normal day. However we will not go into more detail on this, as it is beyond the scope of this thesis. The data have been handed to the people at KVL, where they are currently devising a model for heat detection in much more detail than we have done here. However from the activity graphs it seems the creation of a model this should be feasible.

9.7 Node Lifetime

Included in the initial packet sent when a node checks in to a PC, is the current voltage of the battery-pack. This was included so that we would be able to discover if the nodes were running out of power before the end of the experiment.

When the experiment ended, the nodes were taken off the sows, and placed at DIKU, without turning them off. After a few days, we started the check-in handling application on a PC close to the node. We left this application to run, until the nodes would no longer check in to the PC, to get an idea of how long the nodes could have survived. We only included the 4 nodes that were present during the entire experiment.

Figure 9.4 displays the voltage curves from the experiment. The nodes were taken off the sows after 20 days, and after 24 days the nodes started to check-in to the PC at DIKU. Between these two days there are no data points, which is why the line in between them is smooth. There is no noticeable change in the slope of the lines after 11 days, where the nodes were reprogrammed. The decline in voltage matches what we have seen when we measured the capacity of the batteries in Section 7.2.2.

Table 9.6 lists the last check-in that happened before the nodes experienced their first brown-out. The shortest lasting node, `4EBF` died after approximately 30

Figure 9.4 – *The voltage curves for the nodes during the experiment*

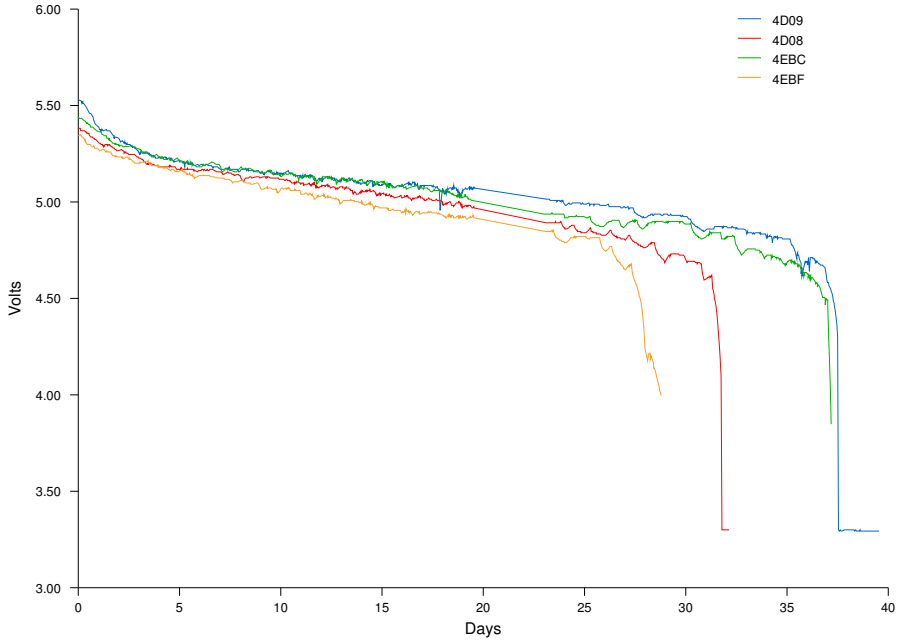


Table 9.6 – *Last post-experiment check-in, where the node have not rebooted due to brown-outs*

Node	Date	Time	Total Days
4D09	08/04	19:28	40
4D0B	01/04	07:09	32
4EBC	06/04	17:53	38
4EBF	29/03	06:12	30

days, and the longest lasting node, 4D09, ran for 40 days. The long tail seen on this node is continually at 3.3 V for the last 3 days, but the node does not reboot until day 40. Exactly why this happens is not clear to us. However a guess is that the node continues to run, even though the input voltage becomes lower than 3.3 V. As the voltage regulator cannot boost the voltage to 3.3 V, the node runs on the same voltage as the batteries provide. This in turn means that the ADC reference voltage is also lowered, and explains why the output from the ADC becomes steady at this point.

The nodes 4EBF and 4D08 both used the Panasonic 2100 mAh cells, while 4EBC and 4D09 used respectively the Ansmann 2300 mAh and 2400 mAh cells. If we assume that we could use $\frac{2}{3}$ of the cell capacity in the experiment, this leaves us with an average current consumption between 1.66 mA and 1.94 mA, which is pretty close to our conservative estimation. However it is far the result we obtained by measuring the current consumption of the node over 3 days. This can be explained by the many connection problems, which was not experienced during the 3 day test.

The node with the shortest lifetime was still able to run for 30 days, 10 days longer than the experiment lasted. So we reached our goal, of making the nodes last through the entire experiment.

9.8 Lessons Learned

While we encountered many problems during the experiment, we have made good use of the lessons from the related work (see Chapter 2). We only described one problem in that chapter, that we were caught by. This is the range problems with the radio communication. We had taken a lot of precautions to avoid this problem, but apparently it was not enough. Our lesson in this case is that we should expect the range to be lowered when the radio is placed in close proximity to an animal. This correlates somewhat with one of the lessons from ZebraNet, where they experience a much lower range when the nodes are deployed on the zebras[62]. If this experiment was to be repeated with the same radio equipment, we should connect at least two Bluetooth modules to each PC, and place them as far away from each other as possible. This would allow a better coverage of the entire pen.

Another lesson we can take with us, is that we should ensure the proper functioning of the sensors using the same application that is going to be deployed. As we have demonstrated, programs which test only a specific functionality can be prone to errors, which does not exist in the final application. Another way to catch such errors would be to extract the data during the experiment. That way we could have discovered the error earlier, and corrected it before the end of the experiment.

This leads us to the problems we had extracting the data. With regard to the time synchronization on the two servers, the only thing we could have done differently was to manually check that the time was synchronized.

With regard to the reboots, we have shown that these can be solved by using welded battery packs. This lesson is important for other projects, where the nodes can move around, as these will be prone to the same problems. However the welded battery packs are more expensive, and leads to more difficult packaging, especially for nodes which include a battery holder, such as the Mica motes.

Another thing that could help alleviate the rebooting problems, is if we had a reboot counter on the node. If we increment a number stored in the flash or

EEPROM each time the node reboots due to power loss we would know how many times the node had rebooted. We could even just zero a bit each time, so we would not have to erase the flash. If we had this information when we extracted the data, we could have used it to treat the data from each reboot as a separate dataset, enabling the use of the simple extraction algorithm.

On the positive side, our packaging of the node worked. When inspecting the boxes after the experiment, there were no signs of water (or manure) entering the boxes. This is even though the neck collar slided down, so that the node was hanging underneath the sow, and therefore was laying directly in the manure.

Furthermore we have shown that even extremely power hungry radios can be used in sensor network deployments, given the correct duty cycle. As the power consumption of Bluetooth modules have gone down, since the introduction of the ROK 101 007 modules, used on the BTnode 2.2, Bluetooth might even be a better choice in some deployments than other radio technologies, due to its high bandwidth.

9.9 Summary

In this chapter we have described the problems encountered during our experiment. We have described how we were able to extract the data, even in the face of the problems we encountered. Furthermore we have verified that the data we extracted is correct, and that we can correlate it with our ground truth. The data from the experiment both as the raw data and in its extracted form is available from `http://hogthrob.42.dk/data`. The application to extract the data is available from the same location.

While we experienced many problems during the experiment, especially connection problems, we still have good coverage of data (approximately 60% overall) when compared to some deployments[55, 57]. Then again other deployments have above 80% of gathered data[33], so there is still room for improvement. The Wired Pigs project[42] deployed two networks, where one delivered at most 30% of the expected data, and the other delivered close to 90%. But we have gotten enough data, that it should be possible to create a model from it, which was the most important goal of the project.

With regard to energy consumption, our conservative estimation was pretty close to the mark, while the estimation obtained by measuring the power consumption for 3 days was very much off. As discussed in Section 9.7, this is most likely caused by the communication problems, which did not occur during our estimation.

Chapter 10
Compression

In the experiment at the farm, the data stored on the node was not compressed. We simply stored 12 bits of data for each axis of the 3-axis digital accelerometer, and 10 bits of data for each axis of the 2-axis analog accelerometer, or in total 60 bits per measurement.

If we had compressed the data we would have been able to store more data in the flash. This could be used to either lengthen the lifespan of the node, or make it more resilient to check-in failures.

We can lengthen the lifespan if we compress all the data, and offload only when the node has almost filled its memory. This will result in a better duty cycle of the radio. If the compression uses less energy than offloading the data does, the lifetime of the node is increased.

However the field experiment has shown that it would be better to make the node more resilient to check-in failures. We can do this by compressing the data, but instead of waiting until the node is filled up, we initiate the offload after an hour (i.e. the same amount of time between offloads as in the field experiment). This way there is less chance that the node has to drop data, as the check-in can fail for a longer period, before this happens.

In this chapter we will look at different compression algorithms, and create a framework for testing them. The focus of this testing will be how the algorithms would have affected the field experiment. We will evaluate two general purpose algorithms, and design one specifically modelled to our dataset. We will evaluate the algorithms with regard to compression ratio, compression speed and current consumption, as all three will affect the algorithms performance in a real deployment.

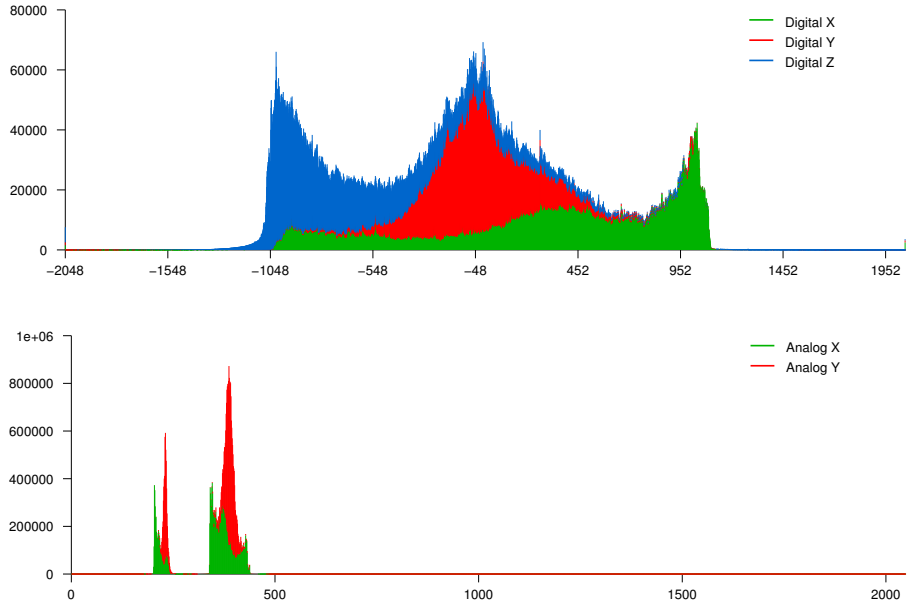
10.1 Overview of the Data

To have a better idea of how the data from the nodes can be compressed, we will take a look at how the 21 million samples obtained during the experiment is distributed.

In Figure 10.1 the number of times each of the possible values in the raw data occurs, is plotted. For the digital accelerometer, we can see that the values we obtain are spread out evenly over a wide range. If we look at the analog accelerometer, we can see that the data is not spread out, but instead lumped together around a few values. The reason behind this, as described in Section 9.6, is that the wrong startup time is was used for the analog accelerometer.

As the data from the digital accelerometer is spread evenly across a wide range, compressing the raw values might not be the best possible solution. If we consider

Figure 10.1 – *Frequency of the raw data from the accelerometers*



that a sow spends most of its day sleeping, we would expect large periods where the measured acceleration does not change much. Then a short period with large changes, and then back to a long period with small changes. The box containing the accelerometer does not necessarily end up oriented in the same direction each time the sow lies down, which affects the raw measurements. However the difference between two consecutive samples should be about the same. To see if this really is the case, we have plotted the difference between two samples. The resulting graph can be seen in Figure 10.2.

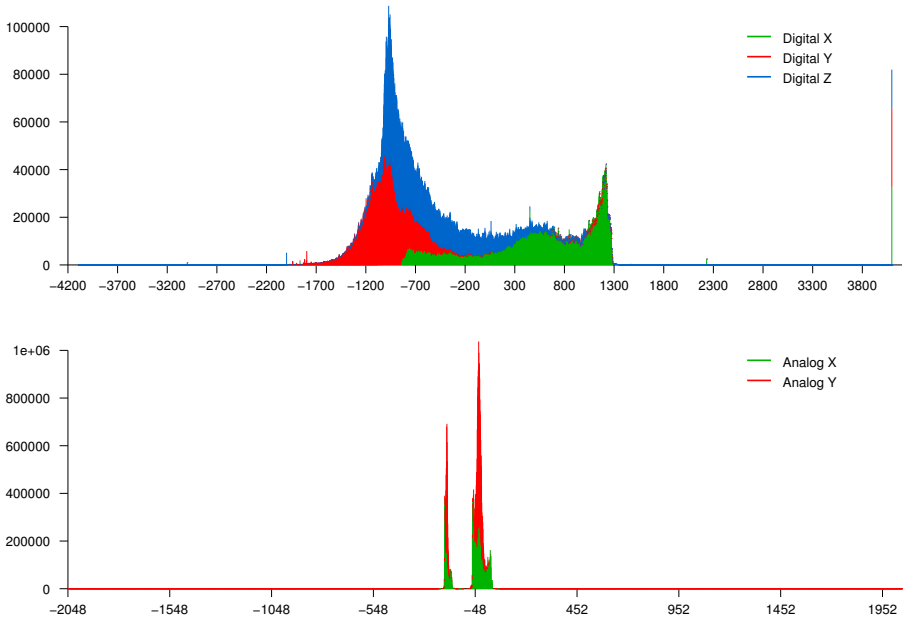
Looking at the graph for the digital accelerometer, we can see that the data becomes less spread out. This indicates that it should be easier to compress the difference, than the raw values. For the analog accelerometer we can see the same trend, although to a lesser extent.

As the data from the analog accelerometer in no way resembles the data we would expect if the accelerometer had been properly initialized, we decided to exclude it from the compression algorithm testing. We suspect that the data would compress much better than correctly measured data, as it spans so small a range of values. Therefore, if we included it in our tests we would get better compression ratios than if the data had been correct.

10.2 Choosing Compression Algorithms

When choosing a compression algorithm for use on the BTnode, there are several limiting factors that influence the choice. First of all, the amount of memory available to the compression algorithm is maximum 1.5 KiB, as we have disabled the external memory, and the application takes up 2.3 KiB of the 4 KiB (see Section

Figure 10.2 – *Frequency of the difference between two measurements*



6.6 for details). This is very limiting as especially algorithms based on a dynamic dictionary, will have trouble producing good results. On the other hand, if a fixed dictionary is needed, this can be stored in the node's program memory, where we can comfortably allocate 16 KiB or more. However a large fixed dictionary will affect the lifespan of the node, as there will be less flash-memory available for storing the sampled data in.

Another limitation is the processing power. The compression of the five samples obtained from the accelerometers must not take longer than approximately 200 ms. If it does, the node will not be able to compress the data at the rate it is obtained. But spending even 100 ms on the compression might be too much, as it will raise the energy consumption, and thus lower the node's lifespan. Lastly it will be an advantage if the algorithm is able to compress the data one sample at a time. Otherwise we will have to collect the samples in a temporary buffer, further raising the memory requirements of the algorithm.

All of these limitations have a severe impact on the available compression algorithms, and on the performance and implementation of these. For example when implementing the Lempel-Ziv-Welch (LZW) compression algorithm, it is customary to use a hash table during the compression. We cannot do that on the node, due to the memory limitations. A further implication of this, is that we cannot simply reuse a previous implementation of an algorithm, as they usually require much more memory in order to work, and often uses dynamic memory allocation which is not available in TinyOS.

In the following sections we will describe three different compression algorithms that are well suited for use on the node.

10.3 Huffman Coding

The Huffman coding[30] is a statistical code, which outputs variable-length codes. The frequencies of the different symbols in the input data is used to assign unique codes to each symbol, while ensuring that the most common symbol receives the shortest code.

A problem for the Huffman coding, is that the compressor must either know the frequencies of different symbols in the data it compresses in advance, or it must do a pass of the data before compressing it, to calculate these frequencies.[1] We do not want to do a pass of the data, as we want to compress it as it arrives. Also, if we did pre-process the data, we would have to include the Huffman codes used to encode the data, in the communication stream.

If we instead use a static set of frequencies, we can compress data as it arrives, and we do not have to transfer the used Huffman codes, as both ends of the communication knows them in advance. And since we already have run the experiment, we have a good idea of what the frequencies of the input data is going to be. The static Huffman codes can be stored in the nodes program memory, where it will not take up any RAM. If the codes are stored in an array, where the symbol to encode is used to lookup the corresponding code, the run-time of encoding a single symbol will be constant.

For the symbol list, we can choose to use either 8-bit symbols, and convert each accelerometer measurement into two symbols, or to use 12-bit symbols for the digital accelerometer, and 10-bit symbols for the analog accelerometer.

10.3.1 Generating the Code Table

To generate the code table, we must first create a Huffman tree of all the symbols that should be handled by our Huffman encoder. From this Huffman tree, we can decide the Huffman codes for each symbol.

To generate the Huffman tree we will use an algorithm that imposes further restrictions on the tree than the original Huffman algorithm does. We go through the input data, and create a frequency table. This table is then used to create a priority queue. In this priority queue there can be two different kinds of elements, Symbols and Nodes. A Symbol represents a symbol in the data, and has the frequency of the symbol, and a depth of zero. A Node has two children, each of which can be either a Symbol or a Node. A Node also has a frequency, which is the sum of the frequency of its children, and a depth which is the maximum of the children's depth plus one.

The priority queue is sorted first according to frequency, and secondly according to depth. In both cases the lowest values goes to the top of the queue. The Huffman tree is then generated using the following algorithm in Algorithm 10.1

To illustrate this, we will generate the Huffman tree for the frequency Table 10.1. The resulting Huffman tree can be seen in Figure 10.3. The first two Symbols we extract from our priority queue are `d` and `a`. These are used to create a new Node with the frequency 10. The next two Symbols are `b` and `c`, as they both have zero depth, while the new `d+a` Node has depth 1. The Symbols `b` and `c` results in a new Node with a frequency of 20. Then `d+a` and `f` are combined. Then `b+c` and `e`, and lastly `d+a+f` and `b+c+e` are combined.

[1] Another option is to use the adaptive Huffman algorithm[51], which performs this frequency calcu-

GENERATING A HUFFMAN TREE (10.1)

```
1:  Input: The priority queue (q)
2:  Output: The root of the Huffman tree
3:  GENERATECODETREE()
4:      while (q.size() != 1) {
5:          left = q.top();
6:          right = q.top();
7:          q.push(new Node(left, right));
8:      }
9:      Return q.top();
```

Table 10.1 – *A frequency table, for Huffman tree generation*

Symbol	Frequency
a	6
b	10
c	10
d	4
e	27
f	11

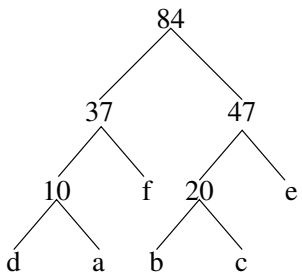


Figure 10.3 – *A wide Huffman tree, as generated by our algorithm*

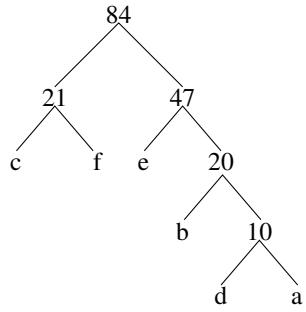


Figure 10.4 – *A deep Huffman tree*

Table 10.2 – *The generated Huffman codes*

Symbol	Wide tree code	Deep tree code
a	001	1111
b	100	110
c	101	00
d	000	1110
e	11	10
f	01	01

Table 10.3 – *Our different versions of the Huffman coding*

Name	8-bit symbols	Difference
`huffman`	✓	
`huffman_diff`	✓	✓
`huffman_whole`		
`huffman_whole_diff`		✓

From the Huffman tree, we can generate the Huffman codes. We assign a bit-string to each symbol in the tree, by adding a 0 to the bit-string each time we descend through left branch, and a 1 each time we descend through a right branch. So the code for `d` in the our tree is `000`, as we need to go left 3 times in the tree to reach it. All the codes for the tree can be seen in Table 10.2.

If we choose not to impose the additional sorting according to depth, we could end with a Huffman tree as seen in Figure 10.4. As can be seen, this tree is one level deeper, than the one created by our algorithm. The codes for this tree is also listed in Table10.2.

It is important to point out, that both the wide and the deep tree are completely valid Huffman trees, and when the data used to generate the frequency table is encoded, the resulting output will have the same length, no matter which tree is used. However we have a preference for the wide tree, as this provides the shortest codes. The length of the codes is important when we wish to store the code table as a big array, as the longest code determines how many bits will be used to store all of the codes. So the length of the longest code, is proportional to the amount of program memory the table will use.

10.3.2 Implementation Notes

We choose to implement 4 different versions of the Huffman coding. The names of these different versions, together with their properties can be seen in Table 10.3. We choose to implement both a version using 8-bit symbols, and one using the whole raw value, and for each of these we have a version encoding the difference, and one encoding the raw values. We did this to see how the compression ratio is affected by these different choices.

We created a program `generate_codes`, which can create the Huffman codes for the 4 different versions. It takes a CSV file containing the data extracted from one node as the input, and outputs a C header file. This C header file contains both the Huffman code table, and the Huffman tree. The codes are needed to compress data, and the tree is needed to decompress the data.

The code table is stored in a large array. For each symbol, the length of the code, and the actual code is stored. The length is needed because we wish to have a fixed length for all the entries, so that we can find the entry for a specific symbol quickly. The length and code are stored bitwise, so if we need 32 bits to represent the longest code, we need 5 bits for the length, resulting in 37 bits per entry.

We could have encoded the table in another way, so that we did not need to store the length. Instead we could have prepended each code with a 1 bit, and then

lation on the fly, using an in-memory sorted table to assign codes to the symbols.

Table 10.4 – *Total size of compressed data, when compressed with the 4 Huffman versions and different code tables*

Algorithm	Total size of the compressed data in MiB					
	Code table					
	4D08	4D09	4D0B	4EBC	4EBE	4EBF
`huffman`	97.28	97.10	**96.78**	96.88	98.92	96.84
`huffman_diff`	103.84	**96.47**	102.83	99.22	103.81	107.95
`huffman_whole`	83.25	83.40	83.11	83.22	85.09	**82.93**
`huffman_whole_diff`	90.63	**82.73**	89.80	89.58	88.43	96.21

added 0 bits in front of this until all codes were the same length. This way we would only have needed 33 bits to store 32 bit long codes. However storing the length in the array was simpler to implement, so we used that encoding. Only the program memory usage should be affected by this, and this will not affect our tests.

A small hurdle in the implementation was that `avr-gcc`, which is used to compile code for the node, is only capable of handling compile-time initialized arrays stored in the program memory if they have a size less than 32 KiB. We needed more space for the `huffman_whole_diff` code table. So the `generate_codes` program can split the code table into two arrays, if necessary.

10.3.3 Choosing the Best Code Tables

We have decided to generate the code tables from a single node's data. We have done this, to make the algorithm perform as good or bad as it would in a real deployment. By not using the entire dataset to generate the code table, we make sure that the code table is not specifically optimized to all the data we compress with it.

Since we have 4 different versions of the Huffman encoding, and 6 different datasets that can be used to generate the Huffman codes, we want to limit the number Huffman encodings we need to test. Therefore we will select the best performing Huffman codes for each of the 4 versions.

To do this, we have generated code tables for all the different combinations of versions and datasets. Then we use these code tables to compress all the data from the experiment. The resulting output sizes can be seen in Table 10.4. From this table we have selected the code tables that gave the best compression for each of the different versions of the Huffman encoding. These are the bold numbers in the table.

10.4 Lempel-Ziv 77

Lempel-Ziv 77[64] (LZ77) is a dictionary based data compression algorithm. It is based on a sliding window over the data to be compressed. In the following we will describe the algorithm using "Data Compression: The Complete Reference[51]" as the reference.

The sliding window is split into two parts. One called the "search buffer" and one called the "look-ahead buffer". The search buffer should be much larger than the look-ahead buffer.

When compressing, the algorithm searches the search buffer after sequences that match the beginning of the data in the look-ahead buffer. If more than one such match is found, the longest one is used. When a match is found, a three part token is outputted. The token contains the offset and length of the prefix match and the first symbol in the look-ahead buffer not matched. If no match is found, the offset and length of the token are both set to zero. After the token have been output, the window is moved to the first element in the input data stream, not matched by the newly written token.

The only major memory requirement of the LZ77 compression algorithm, is the sliding window. So it is easy to adapt LZ77 to the small memory available on the node.

A common improvement to LZ77 is to use two-part tokens. Instead of storing offset, length and the next symbol, only the offset and length is stored. If no match is found the unmatched symbol is written. This requires that all tokens are prefixed with a bit, to tell the difference between plain symbols, and the two part tokens. This changes the algorithm from using constant length codes, to using variable length codes but should improve the compression ratio. We will also include this improvement in our implementation.

10.4.1 Implementation Notes

We choose to implement two different versions of the LZ77 compression algorithm. One where we store the raw accelerometer measurements, and one where we store the difference. We choose to make the compression algorithm work with 8-bit symbols, as this was by far the easiest to implement, and also the how we expected to get the best performance, speed-wise.

10.5 A Simple Data Specific Algorithm

To compete with the two general purpose algorithms, we want to design a very simple algorithm, that is adjusted closely to dataset we need to compress. The algorithm we have chosen is based on the observation that for long periods of times, e.g. when the sow is sleeping, the measurements from the accelerometer should not change much.

When the algorithm receives a measurement, it notes the difference between the last measurement and this measurement. If this difference is small enough to be represented in 4 bits, i.e. if the difference is between -8 and 7, it is written as a 4 bit difference. Otherwise, the raw 12-bit measurement is written. To discern the two cases, a 1 bit is written in front of a short value, and a 0 bit is written in front of a long value.

We have tuned the algorithm on the dataset from the 4D08 node, where a 4 bit length for the short value yields the best compression. We did not use the entire dataset for this tuning, as this would give this algorithm an unfair advantage.

10.6 Compression Framework

To make the implementation and testing of the compression algorithms easier, we designed a simple compression framework. In this framework, we implemented the algorithms in C. The framework is constructed so that we can compile the al-

Figure 10.5 – *The prototypes for the compression framework*

```
void compress_sample(int16_t digi_x, int16_t digi_y, int16_t digi_z,
                     uint16_t ana_x, uint16_t ana_y);
void flush();
void handle_full_buffer(uint8_t *buffer);
```

Figure 10.6 – *The buffer interface*

```
void reset_buffer();
uint8_t *get_buffer();
uint8_t *get_unwritten();
uint16_t bits_left();
void write_bits(uint8_t data, uint8_t len);
```

gorithms both for the PC and for the nodes. Having the compression algorithms running on the PC makes debugging and testing easier.

This also made it easier to verify that the compression algorithms works correctly. We can simply compress all the data from the field experiment, and then decompress the result, while verifying that the decompressed data is the same as the data we compressed.

10.6.1 Compression Algorithm Interface

To work with our compression framework, the compression algorithm must output its data in 256-bytes chunks. The reason for this requirement, is that we would need this on the node if we were to store the compressed data in the flash. So by outputting 256 byte chunks in our tests, we make the results similar to what we would expect if we had compressed the data in the field experiment.

The compression algorithms must provide two functions, the prototypes of which can be seen in Figure 10.5. The `compress_sample` function is called when a new measurement must be compressed, and the `flush` function is called when there is no more data, and the compression algorithm should empty its internal buffers. The last function, `handle_full_buffer` is for the compression algorithm to call, whenever it has filled a buffer, and wants to empty it.

To ease the programming we have also created a very simple interface to handle the 256 bytes buffer, and to write to it bitwise. This interface can be seen in Figure 10.6. The `reset_buffer` function initializes the buffer. This function should be called at the beginning of the compression, and whenever the buffer needs to be reset. `get_buffer` returns a pointer to the start of the buffer, and `get_unwritten` returns a pointer to the parts of the buffer that have not been written to, since the last call to `reset_buffer`. The `bits_left` function returns the number of bits left before the buffer is full. Lastly `write_bits` writes at most 8 bits of data into the buffer. The bits that are written are the least significant bits of the `data` parameter.

10.6.2 Testing the Algorithms on the PC

On the PC we created two programs, called `compress` and `decompress`. Both of these programs take as the first parameter, the compression algorithm, and as the second a CSV file containing samples to either compress or to compare the decom-

pressed data against.

From the first parameter, the programs will construct a path to the compression/decompression algorithm, and dynamically load the algorithm. If the compression algorithm given is called `foo`, then the path will be `foo/foo_comp.so` for the `compress` program, and `foo/foo_decomp.so` for the `decompress` program.

`compress` writes the compressed data to `stdout`, while `decompress` reads compressed data from `stdin`. The `decompress` program cannot currently output the decompressed data, but only compare it to the contents of a CSV file, as this is all we needed for our tests.

10.6.3 Testing the Algorithms on the Node

To test the compression algorithms on the node, we designed a simple application, called `CompressionTest`. This application will read a data stream from the PC through the serial port, and store it in memory. When the memory is filled, the node will compress all the data, optionally sending the compressed data back to the PC. This construction allows us to both verify that the algorithms works correctly on the node, and measure the runtime energy consumption during the compression, without including a serial transfer in this measurement.

To make the measurements more precise, we wish to transfer as much data as possible to the node, before it begins to compress it. Therefore we decided to enable the external memory again for these tests. Some of the nodes actually have 256 KiB external memory, but as the ATMega only handles 64 KiB, the external memory is separated into 4 banks. Two GPIO pins from the ATMega128 have been soldered to the free address-pins on the external memory. Then the application can simply turn these pins on and off, to select the required memory bank. As the node still uses its internal memory for the lowest 4 KiB, we can switch bank without the state of the application disappearing. We will use this to increase the amount of data we can store, before starting the compression.

On the node, only the compression is important for our tests, which is why we did not create a way to test the decompression algorithms on it.

10.7 Testing the Compression Algorithms

When testing the selected compression algorithms the interesting properties of the algorithms are their compression ratio, how long it takes to compress data, and what the node's energy consumption is when compressing data. In the following we will discuss how to measure these properties.

The data we will use for the tests, is the data gathered by the nodes in the field experiment. This way we are ensured that our tests resemble the results we would get, if the compression algorithms were actually used during the field experiment.

10.7.1 Compression Ratio

The compression ratio is defined as follows[51]:

$$Compression\ ratio = \frac{size\ of\ the\ output\ stream}{size\ of\ the\ input\ stream}$$

The compression ratio does not depend on where the compression algorithm is executed, so this test can easily be executed on the PC. To find the compression rate of

the different algorithms, we compress them on the PC, and look at the compressed size. As the size of input stream, we use the size of the `passthru` algorithm. This algorithm simply writes 12 bits per axis, for each measurement from the digital accelerometer, much like we did in the field experiment.

10.7.2 Compression Time and Energy Consumption

The compression time is important, because we need to be able to compress the samples from the accelerometers faster than we obtain them. Compressing a single sample set, must take less than 200 ms, so that we still have time left to perform other tasks, such as offloading the data. The energy consumption is important because it affects the lifetime of the node.

The first thought is that the compression time is going to be a good approximation of the energy consumption. However the energy consumption of the ATMega128 MCU is dependent on the code it executes[37]. A tight loop of `nop` instructions uses 47.5 mW of power, while a tight loop of `add` instructions only uses 30.1 mW, both at 3.3 V. So the energy consumption of the compression algorithm will be dependent on the mix of instructions issued. Therefore we want to measure both the compression time and the energy consumption to get the whole picture.

10.8 Results

In this section we will present the results from our tests of the different compression algorithms. To have a reference to compare the algorithms against, we have included the results we get when we store each measurement as 12-bit per axis, just as in the experiment. We have called this algorithm `passthru`.

10.8.1 Expected Results

We expect that the results from compressing the field experiment data will show that compressing the difference will result in a better compression ratio than compressing the raw values, as already discussed in Section 10.1. Apart from this we do not have any expectations as to which compression algorithm will yield the best compression ratio.

As for the speed, we expect that the two LZ77 variants will be the slowest by far, as they have to search the entire search buffer, for much of the input stream. The other algorithms should be pretty close to each other speed-wise, but the `passthru` should be the fastest, as it does not perform any processing on the data.

With regard to energy consumption we expect the LZ77 algorithm to use most energy, primarily because we expect it to be slower than the others. We do not expect the different algorithms to differ much in their average current consumption, so the algorithm that is fastest, is the one we would expect to have the lowest energy consumption.

10.8.2 Code Size and Memory Usage

To measure the code size and memory usage of the different algorithms, we compiled the `CompressionTest` application with the different compression algorithms. To get comparable results, we annotated the functions in the compression framework (i.e. the functions in Figure 10.5) with `__attribute__((noinline))`, so that

Table 10.5 – *The code size and memory usage for the different compression algorithms*

Algorithm	Code size	Memory usage
`huffman`	1414	8
`huffman_diff`	1500	14
`huffman_whole`	19867	8
`huffman_whole_diff`	43158	15
`lz77`	998	1547
`lz77_diff`	1192	1553
`simple`	602	9
`passthru`	*434*	*3*

code from the compression algorithms did not get inlined into other components.

The results from these compilations can be seen in Table 10.5. For the memory usage column, we have subtracted 256 bytes, as this is the size of the buffer we store the result in during the compression. This buffer is not strictly needed by the compression algorithms, so it seemed most fair to exclude this.

The `passthru` algorithm uses the same bit writing code as the other compression algorithms, but is otherwise as simple as possible. The 3 bytes of memory it uses are all state needed for the bit writing routines. These 3 bytes are also included in all the other algorithms memory consumption, as this is something they would need, in all cases.

All the `_diff` functions use 6 bytes more memory than their non-`_diff` counterparts. This is because the compression algorithms uses 6 bytes to remember the values of the last sample set compressed.

If we look at the `simple` algorithm, we can see that it uses only 9 bytes of memory. 6 of these are used to calculate the difference, and the last three are for the bit writing code.

The `huffman_whole_diff` algorithm uses 42 KiB of program memory. The most significant part of this comes from the code table store in the memory. However it is still much when compared to the 19 KiB of the `huffman_whole` algorithm. The large difference comes from the fact that when we subtract two 12 bit numbers from each other, we need 13 bits to represent the result. So while the `huffman_whole` only has 4096 entries in its code table, the `huffman_whole_diff` has 8192 entries. With more entries in the code table, the codes also become longer, and these two facts explains the size of the code table.

10.8.3 Compression Ratio

To find the compression ratio, we have compressed all the data with the different algorithms. From this we can find the compression ratio, by using the `passthru` algorithm as the input size.

The results from this is available in Table 10.6. From this table it is quite clear that the 8-bit Huffman encodings are worthless when compressing the accelerometer data, as the size of the "compressed" data is larger than the uncompressed dataset. The whole symbol Huffman encoding performs better with a compression ratio of approximately 90%, for both the one that uses the raw value, and the one using the difference.

Table 10.6 – *The compressed sizes of the data collected in the experiment*

Algorithm	Compressed size	Compression ratio
`huffman`	96.78 MiB	105.94%
`huffman_diff`	96.47 MiB	105.61%
`huffman_whole`	82.93 MiB	90.78%
`huffman_whole_diff`	82.73 MiB	90.56%
`lz77`	78.95 MiB	86.42%
`lz77_diff`	64.29 MiB	70.38%
`simple`	53.54 MiB	58.61%
`passthru`	*91.35 MiB*	*100.00%*

The LZ77 algorithm performs better than all the Huffman variants, but the `simple` algorithm actually turns out to be the algorithm that compresses best with a compression ratio of a little less than 60%.

Generally the difference based algorithms perform better than their non-difference counterparts. This was expected, but in the case of the Huffman encoding we expected the difference between the two to be larger.

To compare our algorithms against more common algorithms we have compressed the raw data from the field experiment, stored as 16 bit per axis, using the `gzip` and `bzip2` programs. The compression ratio from this experiment is 61% for the `bzip2` program and 68% for the `gzip` program. Seen from this perspective, the compression ratio of the `lz77_diff` algorithm is quite good, and the `simple` algorithm is even better. However if the input data to the two programs was the difference between samples, instead of the raw values, they would most likely beat the `simple` algorithm, in terms of compression ratio.

10.8.4 Compression Speed and Current Consumption

To measure the compression speed and current consumption, we have attached a node with the `CompressionTest` application installed, to both a PC and to our DAQ. When the node begins to compress data, we measure the current consumption with the DAQ and records both this and the time it took to compress the data. We take care not to include the data transfer in these measurements.

We have compressed all the digital data, obtained from the experiments on the node, while measuring the current consumption, using the program described in Section 10.6.3.

In Table 10.7 the total time to compress all the data and the average time to compress a single sample is listed. While the fastest algorithm compresses all the data in less than one hour, it still takes the most of a day to test it, as the serial transfer speed is the limiting factor. In the case of the slowest algorithm it takes a little more than 6 days to run the experiment.

As expected the `passthru` algorithm is the fastest. The second fastest algorithm is the `simple` algorithm, which only uses 11 minutes more than the `passthru` algorithm to compress all 21 million samples. The two 8-bit Huffman encoding functions are pretty close to each other with regard to compression time. The `huffman_whole` is close to being twice as fast, but as the 8-bit versions need to look up twice as many codes, this makes sense. The `huffman_whole_diff` algorithm is much slower than

Table 10.7 – *The speed of the different compression algorithms*

Algorithm	Total time (hh:mm)	Average time per sample
`huffman`	4:08	0.710 ms
`huffman_diff`	4:09	0.712 ms
`huffman_whole`	2:35	0.443 ms
`huffman_whole_diff`	5:10	0.888 ms
`lz77`	131:19	22.560 ms
`lz77_diff`	121:24	20.857 ms
`simple`	0:48	0.137 ms
`passthru`	0:37	0.105 ms

Table 10.8 – *The current consumption of the different compression algorithms*

Algorithm	Current consumption		
	Average	Total	Per Sample
`huffman`	4.944 mA	20.44 mAh	0.975 nAh
`huffman_diff`	4.940 mA	20.46 mAh	0.977 nAh
`huffman_whole`	4.879 mA	12.57 mAh	0.600 nAh
`huffman_whole_diff`	5.033 mA	26.02 mAh	1.242 nAh
`lz77`	5.540 mA	727.52 mAh	34.720 nAh
`lz77_diff`	5.661 mA	687.20 mAh	32.796 nAh
`simple`	5.185 mA	4.14 mAh	0.198 nAh
`passthru`	4.962 mA	3.02 mAh	0.144 nAh

all the other Huffman algorithms. The only explanation for this is that the extra code needed to use two arrays for the code tables, is the cause of this slow-down.

Looking at `lz77` and `lz77_diff`, these are by far the slowest algorithms, approximately 200 times slower than the `passthru` algorithm. However this matches with our expectations.

As a general note, it does not seem like any of the algorithms would be too slow to use, if we had to redo the field experiment while compressing the data. The slowest algorithm compresses a sample from the digital accelerometer in 22.6 ms (see Table 10.7), which is far from the 200 ms we set as a maximum limit. Of course in the field experiment we would also have to compress the analog data, but even with this included, we would not expect the compression time to become higher than 50 ms, still within our range.

In Table 10.8, we have listed the current consumption of the different compression algorithms, as average, total and per sample. From this table, we can see that the average current consumption changes according to the algorithm used. The Huffman algorithms has the lowest average current consumption, while the LZ77 algorithms has the highest. The difference between the lowest and highest average current consumption is approximately 10%. This is somewhat more than we would have expected, but it still much less than the potential difference in current consumption we refereed to in Section 10.7.2.

If we instead look at the total current consumed in order to compress the data, the `simple` algorithm wins. The closest Huffman algorithm uses 3 times as much

Table 10.9 – *The effect compression would have on the amount of gathered data*

Node	Percent gathered data	
	In experiment	With compression
4D08	44.607%	54.988%
4D09	71.756%	82.176%
4D0B	64.567%	77.345%
4EBC	56.139%	68.656%
4EBE	61.910%	73.286%
4EBF	66.717%	78.773%

current, while the LZ77 algorithms uses approximately 160 times as much.

The approximately 700 mAh needed by the LZ77 algorithms to compress, is quite a lot, when our energy budget is in the 1500 mAh range. The current consumption can probably be lowered, as no optimizations have been performed on the code. Most likely the compression time could be lowered significantly by rewriting the LZ77 algorithms in assembler.

However the `huffman_whole_diff` algorithm, which is the 3rd worst algorithm with regard to current consumption would use only around 26 mAh to compress the data. So all the algorithms we have tested, except for the LZ77 algorithms, could be included on the node without affecting the lifetime very much.

10.9 Would Compression Have Helped?

In Section 6.6, we concluded that we could store 63 minutes of data in the flash, before the there was no program memory left to store the incoming results in. If we assume that we can compress the data with a ratio of 60% — as we have shown that the `simple` algorithm is capable of — we will be able to store 106 minutes of data, i.e. 43 minutes more data.

To estimate how much this compression would have helped us, we use the following approach. We find the missing periods of data in the results from the field experiment. All the periods that are less that 40 minutes long, we simply delete, and the periods that are longer we make 40 minutes shorter. This should give us a rough estimate of how much data we could have gathered.

In Table 10.9, we have listed how many percent of the data we gathered during the experiment, and how many we potentially could have gathered if we used the `simple` compression algorithm. For most of the nodes we would raise the amount of gathered data from two thirds to three quarters. So while it would have given us more data coverage, it would not have alleviated our communication problems entirely.

10.10 Summary

In this chapter we have developed a simple compression framework, for use in determining the important performance characteristics when compression algorithms are used for sensor networks.

We have described and evaluated 3 different compression algorithms, using this framework. The algorithms have been chosen because they are well suited for

the limited environment provided by the BTnode. We have shown that our own simple algorithm is able to beat the two general purpose algorithms, in all the tests.

Some of this is because our simple algorithm is modeled closely to how we expect the data to behave. However this does not explain it all.

If we start by looking at the poor compression ratio of the Huffman algorithms, we can explain this by the fact that many symbols in the input data has frequencies that are close to each other. This results in relatively long codes, which again ensures that the compression ratio will be poor. For the `huffman_whole` algorithm the shortest code is 9 bits long, while it is 10 bits for the `huffman_whole_diff` and 3 bits for the 8-bit algorithms.

The LZ77 algorithms has a much better compression ratio, but it is limited by the fact that they need to find matches in the previous data. The last bits of the output value of the digital accelerometer fluctuates randomly, which makes it harder for the LZ77 algorithms to find long matches.

The `simple` algorithm basically takes advantage of this random fluctuation, and uses as little space as possible to represent it. This is the key to its good performance, and why it also is able to beat much more advanced algorithms such as `gzip` and `bzip2`.

The compression framework, complete with compression algorithms are available from `http://hogthrob.42.dk/compression`.

10.10.1 Further Work

For future work, there are primarily two things that we would like to do:

- Implement more general purpose compression algorithms.
- Evaluate some lossy compression algorithms.

We would like to implement more general purpose algorithms, as the ones we have chosen are very simple, and we therefore cannot expect good performance from them. However the challenge in implementing more advanced algorithms will be in making them perform well with the limited memory available on the nodes.

For this thesis we have avoided lossy compression techniques on purpose. However there might be a lot to gain from this. If we can detect the activity periods of the sows, using e.g. only the 8 most significant bits of data from the digital accelerometer, we already have compressed the data by 66%. Others have been able to compress temperature readings well, while keeping the loss of precision within the error-margin of the temperature sensor used[52], so a similar approach might give good results for us also.

Chapter 11

Conclusion

We have designed a sensor board for the BTnode, that allows us to monitor the activity of sows. We have implemented a data collection application, and have deployed it for a period of 20 days on 5 sows. In this deployment we have collected a huge amount of activity data, and using this we have shown that our application works as expected.

During the deployment we discovered several problems. The most prominent was that data was being dropped, because of connectivity problems between the PC's and the nodes. This was in spite of the pre-deployment tests performed in the same pen over twice the distance. However we succeeded in collecting just above 60% of the expected data in total.

Our most conservative lifetime estimation ended up being very close to the actual lifetime of the deployed nodes, while the estimation based on measuring the application for 3 days was somewhat off. This can be explained by the connectivity problems, which led to the Bluetooth module being turned on for a larger period than expected.

Our compression framework have allowed us to evaluate how well different compression algorithms would perform, if they had been used during the deployment. We have developed an algorithm specifically designed for the gathered data, and have shown that it allows better a better compression ratio than common general purpose algorithms. From the energy consumption measurements we have shown that our custom compression algorithm would not affect the node's lifetime much, where the LZ77 compression algorithm would have used almost half of the energy available on the node, just to compress the gathered data.

11.1 Future Work

From this point the focus of the Hogthrob project, will be to develop the heat-detection model, and implement it. For this purpose a FPGA based platform have been developed[37], which allows prototyping hardware accelerators that might be needed for an energy efficient implementation of the heat-detection model.

Apart from this, there is the task of incorporating our improvements to the TinyBT stack, into the base TinyOS tree. As described in Section 5.5.1, a port to the BTnode 3 is also in in order, so that TinyOS applications can use the Bluetooth radio on this platform.

The compression framework should also be incorporated into TinyOS, and the interface should be made more generic to allow easier integration into new applications. Furthermore new specific and general purpose algorithms should be imple-

mented, to allow users of TinyOS easy access to a variety of compression algorithms.

11.2 Future Work on Similar Applications

During the development of our application, we have been contacted by several people, who have expressed an interest in using a similar system for health care. Two application areas have been proposed to us:

- Using the system to detect whether fever in children is serious.
- Using the system to detect if senior citizens are prone to falling.

For the first application, the problem is that when children are hospitalized with a fever, it is difficult to measure if the fever is serious. However, doctors and nurses — over time — learns to tell this by observing the children. They are not able to explain exactly which signs they look for, but the assumption is that the activity level of the child plays a part in this. Therefore they would like to attach accelerometers to the children, and gather activity information. These experiments would have to happen at a hospital, so it would be a requirement that the node can collect this activity information in its memory, as using a radio could interfere with other equipment at the hospital. Also the node would have to be small enough to not limit the children's movements.

The system to detect if senior citizens are prone falling, would become part in a larger Bluetooth based system. The system's goal is to monitor the health of senior citizens, and it already includes a scale, and a device for measuring the blood-pressure. The devices offloads the obtained measurements over Bluetooth through a mobile phone to an off-site server, where the physicians can examine the data.

For both systems, a data collecting experiment is needed as a first phase to establish a detection model. A single node with a modified version of our sow monitoring application have already been deployed in an initial feasibility study for the senior citizen project.

Appendix A

Bibliography

[1] Analog Devices. *16-Lead Lead Frame Chip Scale Package*.

[2] Analog Devices. *ADXL202E: Low-Cost ± 2 g Dual-Axis Accelerometer with Duty Cycle Output*, 2000.

[3] Analog Devices. *ADXL210E: Low-Cost ± 2 g Dual-Axis Accelerometer with Duty Cycle Output*, 2002.

[4] Analog Devices. *ADXL320: Small and Thin± 5 g Accelerometer*, 2004.

[5] Ansmann Energy. *Datasheet for Ansmann 2300*.

[6] Ansmann Energy. *Datasheet for Ansmann 2400*.

[7] Atmel. *AVR105: Power Efficient High Endurance Parameter Storage in Flash Memory*, 2003. Application Note.

[8] Atmel. *8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash*, 2004.

[9] *AVR Libc*. Available from: `http://www.nongnu.org/avr-libc/user-manual`.

[10] Jan Beutel, Philipp Blum, Matthias Dyer, and Clemens Moser. *BTnode Programming - An Introduction to BTnut Applications*, 1.0 edition, May 2004.

[11] Bluetooth SIG. *Specification of the Bluetooth System - Core*, 2 2001.

[12] C. V. Bouten, K. T. Koekkoek, M. Verduin, R. Kodde, and J. D. Janssen. A triaxial accelerometer and portable data processing unit for the assessment of daily physical activity. *IEEE Trans Biomed Eng*, 44(3):136–147, March 1997.

[13] Jennifer Bray and Charles F Sturman. *Bluetooth 1.1: Connect without cables*. Prentice-Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[14] Isidor Buchmann. *Batteries in a Portable World*. Cadex Electronics Inc., 2001.

[15] Cécile Cornou and Teresia Heiskanen. Hogthrob - Eksperimentel protocol, 2004.

[16] Crossbow Technology. *Product Feature Reference*.

[17] Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. Compressing historical information in sensor networks. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 527–538, New York, NY, USA, 2004. ACM Press.

[18] Analog Devices. Accelerometer Products Sales Primer, 2000.

[19] Energizer. *Nickel-Metal Hydride Application Manual*, 2001. Available from: http://data.energizer.com/PDFs/nickelmetalhydride_appman.pdf.

[20] Ericsson. *Ericsson ROK 107 001: Bluetooth™Multi Chip Module*, 2001.

[21] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[22] Freescale Semiconductors. *MMA6231Q: ± 10 g Dual Axis Micromachined Accelerometer*, 2004.

[23] Freescale Semiconductors. *MMA6260Q: ± 1.5 g Dual Axis Micromachined Accelerometer*, 10 2004.

[24] David Gay. Matchbox: A simple filing system for motes, 2003.

[25] David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI*, pages 1–11, 2003.

[26] Rony Geers, Steven Janssens, Jan Spoorenberg, Vic Goedseels, Jos Noordhuizen, Hilde Ville, and Jan Jourquin. Automated Oestrus Detection of Sows With Sensors for Body Temperature and Physical Activity. In *Proceedings of ARBIP95*, 1995.

[27] John L. Hennessy and David A. Patterson. *Computer organization and design (2nd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[28] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104, New York, NY, USA, 2000. ACM Press. Available from: http://www.tinyos.net/papers/tos.pdf.

[29] Jason L. Hill and David E. Culler. Mica: A Wireless Platform for Deeply Embedded Networks. *IEEE Micro*, 22(6):12–24, 2002.

[30] David A. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, Sep 1952.

[31] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *ASPLOS, San Jose, CA*, October 2002.

[32] Oliver Kasten and Marc Langheinrich. First Experiences with Bluetooth in the Smart-Its Distributed Sensor Network. Workshop on Ubiqitous Computing and Communication, Conference on Parallel Architectures and Compilation Techniques (PACT) 2001, October 2001.

[33] Lakshman Krishnamurthy, Robert Adler, Phil Buonadonna, Jasmeet Chhabra, Mick Flanigan, Nandakishore Kushalnagar, Lama Nachman, and Mark Yarvis. Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 64–75, New York, NY, USA, 2005. ACM Press.

[34] Kent Krøyer. Bluetooth indtager kostalden. Article in the Danish newspaper Ingenøren the 8th of April 2005, 2005.

[35] Martin Leopold. Evaluation of Bluetooth Communication: Simulation and Experiments. Technical Report 02-03, Dept. of Computer Science, University of Copenhagen, 2002.

[36] Martin Leopold. Tiny Bluetooth Stack for TinyOS, 2003. Available from: http://www.diku.dk/~leopold/work/tinybt.pdf.

[37] Martin Leopold. Power Estimation using the Hogthrob Prototype Platform. Master's thesis, Dept. of Computer Science, University of Copenhagen, 12 2004.

[38] Martin Leopold, Mads Dydensborg, and Philippe Bonnet. Bluetooth and Sensor Networks: A Reality Check. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, pages 103–113, November 2003.

[39] David Linden and Thomas B. Reddy. *Handbook of batteries*. McGraw-Hill, 3rd edition, 2002.

[40] Klaus Skelbæk Madsen. The Case for Buffer Allocation in TinyOS. Hogthrob Technical Note 1, Dept. of Computer Science, University of Copenhagen, May 2005.

[41] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, September 2002.

[42] Ian McCauley, Brett Matthews, Liz Nugent, Andrew Mather, and Julie Simons. Wired Pigs: Ad-Hoc Wireless Sensor Networks in Studies of Animal Welfare. In *EmNetS-II: The Second IEEE Workshop on Embedded Networked Sensors*, pages 29 – 36, 2005.

[43] Brent A. Miller and Chatschik Bisdikian. *Bluetooth Revealed*. Prentice-Hall PTR, Upper Saddle River, NJ, USA, 2002.

[44] Lama Nachman, Ralph Kling, Robert Adler, Jonathan Huang, and Vincent Hummel. The Intel Mote platform: a bluetooth-based sensor network for industrial monitoring. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks, IPSN 2005, April 25-27, 2005, UCLA, Los Angeles, California, USA*, pages 437–442. IEEE, 2005.

[45] Khalil Najafi, Junseok Chae, Haluk Hulah, and Guohong He. Micromachined Silicon Accelerometers and Gyroscopes. In *IROS 2003: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2353–2358, 2003.

[46] National Semiconductor. *LP2987/LP2988 Micropower, 200 mA Ultra Low-Dropout Voltage Regulator with Programmable Power-On Reset Delay*, 2000.

[47] Panasonic. *Datasheet for Panasonic HHR210A*.

[48] Dragan Petrovic, Rahul C. Shah, Kannan Ramchandran, and Jan Rabaey. Data Funneling: Routing with Aggregation and Compression for Sensor Networks. In *Proc. 1st IEEE Intl. Workshop on Sensor Network Protocols and Applications (SNPA)*, Anchorage, AK, May 2003.

[49] Joseph Polastre. Design and Implementation of Wireless Sensor Networks for Habitat Monitoring. Master's thesis, University of California at Berkeley, 2003.

[50] Joseph Polastre, Robert Szewczyk, and David E. Culler. Telos: enabling ultra-low power wireless research. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks, IPSN 2005, April 25-27, 2005, UCLA, Los Angeles, California, USA*, pages 364–369. IEEE, 2005.

[51] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., second edition, 2004.

[52] Tom Schoellhammer, Eric Osterweil, Ben Greenstein, Mike Wimbrow, and Deborah Estrin. Lightweight Temporal Compression of Microclimate Datasets. In *29th Annual IEEE Conference on Local Computer Networks (LCN 2004), 16-18 November 2004, Tampa, FL, USA, Proceedings*, pages 516–524. IEEE Computer Society, 2004.

[53] STMicroelectronics. *LIS3L02AS4: Inertial Sensor: 3 Axis - 2g/6g Linear Accelerometer*, 11 2004.

[54] STMicroelectronics. *LIS3L02DS: Inertial Sensor: 3 Axis - 2g/6g Digital Output Linear Accelerometer*, 2 2004.

[55] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a tlarge scale habitat monitoring application. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 214–226, New York, NY, USA, 2004. ACM Press.

[56] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons From A Sensor Network Expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)*, January 2004.

[57] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 51–63, New York, NY, USA, 2005. ACM Press.

[58] B. Warneke, M. Last, B. Liebowitz, and K. S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. *Computer*, 34(1):44–51, 2001.

[59] P. Wouters, R. Puers, R. Geers, and V. Goedseels. Implantable Biotelemetry Devices for Animal Monitoring and Identification. In *Engineering in Medicine and Biology Society, 1992. Vol.14. Proceedings of the Annual International Conference of the IEEE*, 1992.

[60] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A wireless sensor network For structural monitoring. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 13–24, New York, NY, USA, 2004. ACM Press.

[61] Hugh D. Young and Roger A. Freedman. *University Physics*. Addison-Wesley, Reading, Massachusetts, ninth edition, 1996.

[62] Pei Zhang, Christopher M. Sadler, Stephen A. Lyon, and Margaret Martonosi. Hardware design experiences in ZebraNet. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 227–238, New York, NY, USA, 2004. ACM Press.

[63] Yunyue Zhu and Dennis Shasha. Efficient elastic burst detection in data streams. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 336–345, New York, NY, USA, 2003. ACM Press.

[64] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Appendix B

Sow Movements

Legend for M column	
↑	The sow is standing up
↓	The sow is laying down
⊗	The sow is out of view

Table B.1 – *Sow movement form 0:00 to 5:00, the 1th of March 2005*

Time	Sow 1 4EBC		Sow 2 4D08		Sow 3 4EBF		Sow 4 4D08		Sow 5 4D09	
	M	Cam	M	Cam	M	Cam	M	Cam	M	Cam
0:00	⊗		⊗		↑	4	↓	2	⊗	
:05					⊗	4				
1:05					↑	4				
:48							↑	2		
:52							↓	2		
2:36	↑	2								
:43					⊗	4				
:48	↓	2								
:50									↑	4
:53	↑	2								
:54									↓	4
:57									↑	4
3:01									↓	4
:11									↑	4
:16									↓	4
:17	⊗	2					↑	2		
:20									↑	4
:22									⊗	4
:26									↑	4
:33							↓	2		
:35									⊗	4
:43									↑	4
4:07									⊗	4
:32					↑	4				
:34					↓	1				

Table B.2 – *Sow movements form 0:00 to 5:00, the 1th of March 2005*

Time	Sow 1 4EBC		Sow 2 4D08		Sow 3 4EBF		Sow 4 4EBE		Sow 5 4D09	
	M	Cam	M	Cam	M	Cam	M	Cam	M	Cam
0:00	⊗		⊗		⊗		⊗		⊗	
:19					↑	4				
:20	↑	4								
:28					⊗	4				
:55	↓	3								
:56					↑	4				
:57	↑	3								
1:10					⊗	3				
:15	↓	2								
:20	↑	2								
:25	↓	3								
2:02	↑	2								
:03	↓	2								
:56									↑	4
3:10									⊗	4
:41							↑	2		
:42							↓	2		
:45			↑	2						
:46			↓	2					↑	4
:48									⊗	4
:53							↑	2		
:54			↑	2						
:55							↓	2		
:59			⊗	2						
4:31									↑	4
:36							↑	2		
:39			↑	2						
:40			↓	2						
:41							↓	4		
:43			↑	2						
:46							↑	4		
:48							↓	4		
:50	↑	2								
:51	↓	2								