Paper submitted within the scope of the Master's Thesis
Master of Industrial Sciences
GROUP T - Leuven Engineering College - 2012-2013

Design of a Wireless Sensor Networking Test-Bed

Bjorn Deraeve*, Roel Storms*

*Master student EA-ICT focus Internet Computing, GROUP T - Leuven Engineering College, Andreas Vesaliusstraat 13, 3000 Leuven

Supervisor: Dr. Luc Vandeurzen
Unit Information, GROUP T - Leuven Engineering College, Andreas Vesaliusstraat 13, 3000 Leuven, luc.vandeurzen@groept.be

ABSTRACT

Wireless Sensor Networking (WSN) is an emerging technology that is shaping the world around us. However, as WSNs proliferate, connecting these nodes becomes more and more challenging. In this work we designed, implemented and configured a ZigBee WSN test-bed and a middleware communication layer to extract the sensor data from the network. This paper details the challenges we faced and the solutions we developed for them. These challenges are network stability, latency, energy consumption, quality of service and both event and query-based sensor acquisition. All these challenges have been implemented and tested with Libelium's first generation Waspmote. A stable monitoring network consisting of at least five nodes, including one weather station, was deployed at Group T, campus Vesalius. This network responds to several commands sent through a web interface. Developing this network has convinced us that ZigBee is well suited for networks that share our objectives. We conclude that ZigBee is a promising protocol with many possible applications. On the other hand, developing the network also leads us to conclude that Libelium's platform still faces practical limitations when extended with ZigBee.

Keywords

Internet of Things, Libelium Waspmote, Wireless Sensor Network, ZigBee

1 Introduction to Wireless Sensor Networks

1.1 Introduction

The ENIAC[1], the first electronic computer designed by American scientists J. Presper Eckert and John W. Mauchly, Turing-complete, pioneering in 1946 and designed to calculate artillery firing tables [Flamm, 1988]. A wireless sensor network node, over 50 years later, 10 million times smaller than the ENIAC, still Turing-complete, and still military roots. The origins of the research in Wireless Sensor Networks (WSNs) can be traced back to DARPA[2],which sponsored a workshop at Carnegie Mellon University in 1978, identifying the technology components for a Distributed Sensor Network (DSN) [Balansingham and Wang, 2010]. But at the time the technology was not quite ready. Sensors could take up the size of a shoe box and up, limiting the number of potential applications. The earliest DSNs were also not very tightly associated with wireless communication. In 1998 a new wave of research in WSNs started. Again DARPA acted as a pioneer by launching the initiative research program 'SensIT', which added new capabilities to the current sensor networks such as ad hoc networking, dynamic querying and tasking, reprogramming and multi-tasking. At the same time the IEEE[3] noticed the high capabilities and low expenses of WSNs and defined the IEEE 802.15.4 standard for low power consumption, low data rate wireless PANs[4] for 868MHz, 915MHz and 2.4GHz radios. Finally, in 2002, the ZigBee Alliance was established and published the ZigBee standard, based on IEEE 802.15.4. The standard adds a suite of high level communication protocols for WSNs such as device coordination, network topologies and interoperability with other wireless products.

What if:

Ubiquitous Computing & Networking

+

Sensing & Control ?

Currently, WSNs are seen as one of the most prospective technologies of the 21st century [21 ideas for the 21st century, 1999]. China for example, has involved WSNs in their national strategic research programs (Program 973)[Zennaro, 2008]. That project follows an application-driven methodology and aims to issues identified with the real-world critical problems facing Chinese society. Over the years the program has funded areas such as agriculture, health, resources, energy, population and materials and brought significant benefits to China's sustainable economic and social development.

[1] Electronic Numerical Integrator And Computer (ENIAC)

[2] Defense Advanced Research Projects Agency (DARPA), is an American agency responsible for developing military technologies. DARPA has fund the research in many technologies which have had major effect on the world, including ARPANET, the first wide-area packet switching network (ancestor of the Internet) and Douglas Engelbart's precursors to the graphical user interface (GUI)[Wikipedia, 2013c].

[3] Institute of Electrical and Electronics Engineers (IEEE)

[4] Personal Area Network (PAN)

1.2 Wireless is everywhere

1.2.1 Definition of Wireless Sensor Network

A Wireless Sensor Network (WSN) consists of spatially distributed autonomous sensors connected via a (wireless) communications infrastructure to cooperatively monitor, record and store physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants [Buttrich, 2010]

1.2.2 Internet of Things

The Internet of Things (IoT) is a term that was first used by Kevin Ashton[5] in 1999 and answers the question introducing this chapter. It refers to uniquely identifiable objects (things) and their virtual representation in an Internet-like structure. It is a vision of a network of Internet-enabled objects, combined with web services which interact with these objects. If all objects in the world would be equipped with minuscule identifying devices this could transform our daily lives. By embedding computational capabilities in all kinds of objects, including living beings, it will be possible to provide a qualitative and quantitative shift in several sectors: logistics, domotics, healthcare, entertainment, and so on.

WSNs provide a virtual layer where the information about the physical world can be accessed by any computational system. As a result, WSNs are one of the most important elements for realizing the vision of the Internet of Things paradigm [Alcaraz et al., 2010]. On May 2nd, 2012, Libelium[6] published a list of 50 cutting edge *Internet of Things* applications. According to Libelium, by 2020, more then 50 billion devices will be connected to the Internet [Libelium, 2012].

The IoT can also be considered as the third wireless wave, following cellular technology and WiFi. Today wireless technology also includes the world of sense and control technology bridging the gap between the virtual electronic world and the human physical world [deGategno and Garret, 2010].

1.3 Characteristics of a WSN

1.3.1 Node architecture

A typical sensor node (mote) has the following components:

- Microcontroller (+ memory)
- Transceiver
- Sensors (+ ADCs)
- Power supply

[5] Kevin Ashton co-founded the Auto-ID Center at the Massachusetts Institute of Technology (MIT). The Center is a research group in networked RFID and newly emerging sensing technologies. The main goal was the development of the Electronic Product Code (EPC), a global RFID-based item identification system intended to replace the UPC bar code [Wikipedia, 2013a]

[6] The manufacturer of the Hardware we used. See chapter 3 for more information.

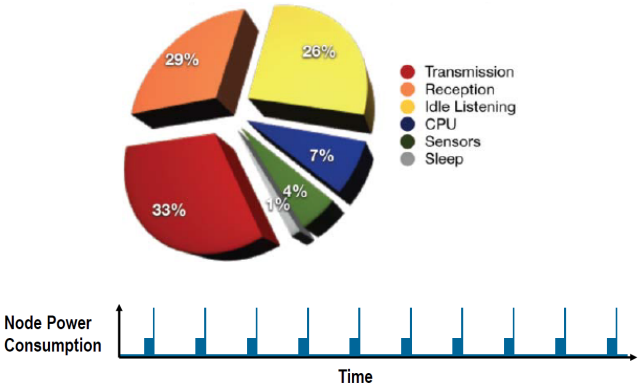


Figure 1: The typical power consumption of a node

The main controller options are Microcontrollers, DSPs[7], FPGAs[8] or ASICs[9]. A microcontroller is the best option for a WSN node. They are general purpose and are optimized for embedded applications, so they use little power. DSPs are optimized for signal processing tasks and not suitable for WSNs. FPGAs can be good for testing purposes and ASICs are good if peak performance is needed without flexibility. The Libelium Waspmotes used for the test-bed have an 8-bit Atmel controller (see 4.4.2).

1.3.2 Fundamental Challenges in WSNs

The biggest challenge WSNs encounter is without a doubt power consumption. Power! Power! Power! The next section will briefly introduce the general concepts and in section 3.3 the power consumption of the WSN we developed will be analyzed thoroughly. Other challenges often fall back to this limited amount of available energy, such as for example security.
There are dozens of other basic challenges for WSNs. Unattended operation and environmental influence makes a mote prone to failure. Mobility can cause topology changes. WSNs can use more than 100 nodes, this leads to scalability and synchronization issues. There must also be a form of synchronization with sleeping nodes, network responsiveness and robustness, not to forget making sense out of sensors.

1.3.3 Power considerations

Figure 1 shows a typical power usage division in a WSN node [Berger, 2009]. Clearly the main power cost is due to Transmission / Networking. As a result, to obtain an acceptable battery life nodes must sleep most of the time. Effective use of network transmission (section 3.3.4), effective dynamic power management (section 3.3.1) and optimal duty cycles (section 3.3.2) are the key to conserving power.

[7]Digital Signal Processor (DSP)
[8]Field-Programmable Gate Array (FPGA)
[9]Application-Specific Integrated Circuit (ASIC)

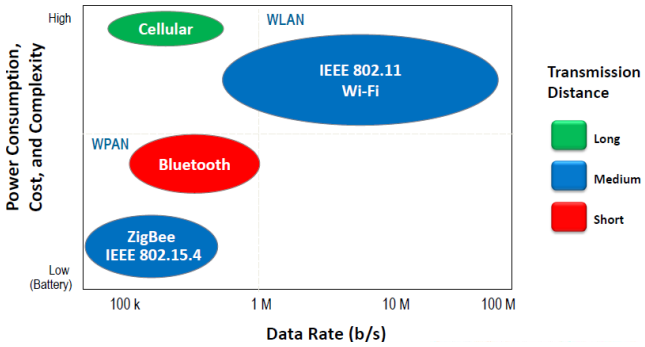


Figure 2: Comparing the ZigBee standard with Cellular, Bluetooth and WiFi

1.3.4 Benefits of Wireless Measurements

Wireless Sensor Networks provide benefits in roughly three categories: installation and maintenance costs can be reduced, measurements can be optimized thus increasing efficiency, and finally infrastructure limitations can be overcome.

1.4 Wireless Standards and Technology

1.4.1 Standards enable growth: ZigBee

Not only do standards allow devices from different vendors to interoperate, they also provide OEMs[10] and integrators the flexibility of second sourcing. The ZigBee Alliance is an independent standardization organization which is driven by a large group of OEM companies and has definitely had a large impact on the rapid development of WSNs. Figure 2 indicates the most critical properties of the ZigBee standard. Some rules-of-thumb are:

- The higher the frequency, the higher the data rate
- The lower the frequency, the further the reach
- All radio waves show strong absorption in water and metal

Table 1 shows the typical power consumption, throughput, range and application examples of each technology [Farahani, 2008].

1.4.2 ZigBee standard

At the moment ZigBee is the leading protocol to implement low-cost low-data-rate, short-range WSNs. It provides extra functionality regarding advanced routing capabilities and network stability. A common concept used to simplify and to make digital communication more flexible, is the use of networking layers. Figure 17 in appendix A shows how this is organized in the ZigBee protocol stack. The bottom two layers are defined by the IEEE 802.15.4 standard and define the specifications for PHY and MAC layers. ZigBee only defines the networking, application and security layers on top of IEEE 802.15.4.

[10]Original Equipment Manufacturer (OEM)

	Battery Life	Data Rate	Range	Application Examples
ZigBee	1-4 years	20 to 250Kbps	100 m	Wireless Sensor Networks
Bluetooth	1-2 weeks	1 to 3 Mbps	10 m	Wireless Headset
IEEE 802.11g	1-2 days	6 to 54Mbps	30 m	Wireless Internet Connection

Table 1: Comparing the ZigBee standard with Bluetooth and WiFi

2 Design of a Wireless Sensor Networking Test-Bed

2.1 Thesis subject context

This thesis is actually part of a bigger project with as goal the creation of a WSN test-bed, the storage of sensor data to a cloud database and the development of a graphical web-application to visualize, configure and control the WSN test-bed.

2.2 WSN Test-Bed and gateway to extract the data

The purpose of this thesis is to create a WSN test-bed, covering part of the Group T, campus Vesalius, building. So the main focus of this thesis subject, is to sample the data via the nodes and to extract that information out of the network, indicated in green on figure 3. This test-bed consists of at least five nodes, including one weather station. The weather station will be deployed on the roof of Group T, campus Vesalius. The other nodes will be installed amongst the classrooms of the campus[11] and the gateway will be installed at module 14. We will then, for example, monitor the temperature, humidity and CO_2 values at those rooms. This data acquisition can happen at intervals between one and several minutes, depending on the user requirements entered through the web interface and on the intended battery life time of the node.
In the end, this test-bed can be used as a lab environment for educational-related projects or can form the base for future research projects, for example by extending the network with actuator nodes. The test-bed will thus represent as a stable network of sensing nodes and it will be very easy to integrate new nodes to the network.
The gateway developed to extract the data from the WSN test-bed is a middleware communication layer supporting sensor discovery, sensor state tracking and event or query-based data acquisition. Since privileged web-interface users have the possibility to communicate directly to the WSN, the gateway also runs a web service that can be contacted by clients via a web browser.
ZigBee defines three logical device roles, differentiated in the network layer (table 18 from [Dynamic C: An Introduction to ZigBee User Manual, 2008] in appendix A compares the device roles in detail). Each network must have one and only one *coordinator*, responsible for establishing the network. In the test-bed this is the XBee radio installed at the gateway. A *router* node has the second most complex device role and can also allow other nodes to join the network by becoming its parent. The third, simplest and cheapest device role is the *end device*. End devices don't have the parenting functionalities and thus cannot allow other nodes to join the network via them. In a typical ZigBee network end devices are also the only nodes which are allowed to sleep. Coordinator and routers are expected to be net-powered and end devices can be battery powered.
ZigBee extends the star and basic peer-to-peer topologies supported by IEEE 802.15.4 with the mesh and cluster tree topology. The test-bed uses a mesh topology, in which coordinator and routers are responsible for route discovery, so there are no static routes. Figure 4 displays a basic star, tree and mesh topology.

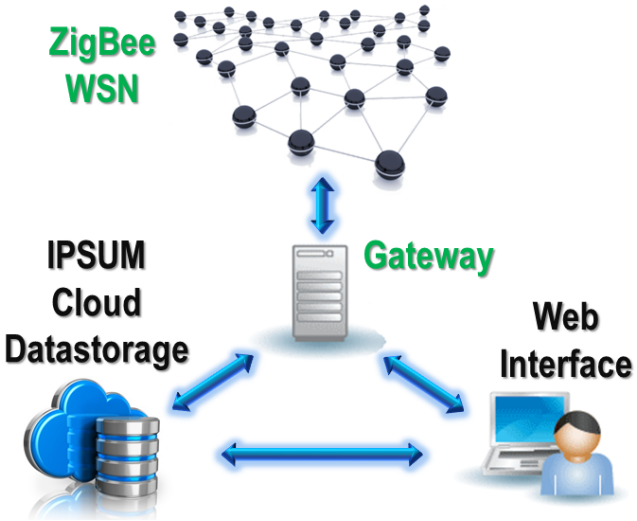


Figure 3: The context of the WSN Test-Bed

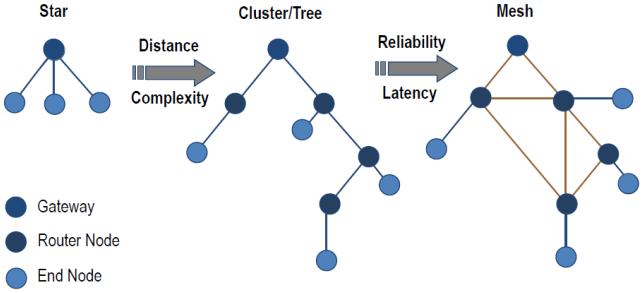


Figure 4: The basic network topologies

[11] Most likely, this will be in rooms V200, V6.01, V10.01 and V11.01.

2.3 Data storage

For reliable sensor data storage the gateway will forward the data to Ipsum. This is a cloud storage system developed at Group T as a master thesis of Ruben Taqc. Ipsum is already in use to log data coming from other sensors and has received a few updates to better support the WSN test-bed.
For a WSN it is crucial that data storage happens correctly at all times. No data can be lost. So if Ipsum is unreachable, the gateway will temporally store the incoming data.

2.4 Web-Interface

The web interface of the WSN test-bed is the master thesis of Matthias Verhelst and provides the user with 2D/3D models indicating the location of the sensor nodes. By selecting a sensor, the user will), depending on his privileges, will have access to the current state of the node, will be able to change the node's configuration and will be able to send messages (requests) to the node. For non-privileged users the web interface will provide data via Ipsum. The web application also provides the service to add or delete sensors and to define their service capabilities, e.g. the sensor layout of a certain node.

2.5 Requirements

In a first stage, this work aimed to design, implement and configure the ZigBee WSN test-bed and the communication layer to extract the sensor data. The most important aspect of this part were network stability, network topologies and network coverage. The network must monitor sensors at default intervals by all means and this data is not to be lost. Nodes must be able to self-recover when they were temporarily unable to join the network.
The network must also support additional services such as sensor discovery, sensor state tracking and event-based sensor acquisition. A privileged user of the web interface must for example be able to set different time settings. This can be for all sensors of a certain type or for all sensors of a certain node. This means a node may, for example, have to sample different sensors at different intervals. These settings must be handled by the gateway and stored into the nodes. To assure overall network stability, network security aspects must be taken into account as well.

3 A WSN with Waspmotes: Background information

3.1 Introduction

Waspmote is more than just hardware. It is an open source platform for wireless sensors, that especially focuses on low power consumption and autonomy. Battery lifetime greatly depends on the duty cycle and the used radio.
But it didn't just start with Waspmote and it will definitely not end with it. In 2007 developers from Libelium collaborated with the Arduino Team creating the first open hardware shield for Arduino, the "Arduino XBee Shield" [Arduino Team, 2012]. The shield allowed an Arduino board to communicate wirelessly via ZigBee. Libelium used the shield to develop their first sensor device, the SquidBee, intended for creating sensor networks. Although the SquidBee is self-powered and implements wireless communications, it is more a sensor device than a wireless sensor device. It has a constant consumption of 50mA, discharging the battery within hours. The SquidBee was created for teaching and educational purposes only [Libelium, 2009]. Since the platform was not radio certified the motes could not be deployed in real scenarios like cities, factories or even houses, so it did not fit Libelium's corporate customer requirements at all. However, the tone of an open hardware and open source wireless sensor device was definitely set.
In 2009 the Waspmote (used for the WSN test-bed) was born, conform to all the above requirements and meeting three radio certification requirements[12]. In addition the Waspmote was built with a complete modular philosophy. The idea behind this design is to integrate only the needed modules in each device, optimizing costs. This is why all modules are connected to the Waspmote via sockets [Cooking Hacks, 2013].
Since its introduction, more than 2000 developers used Waspmote (v1.1) and the platform has received many suggestions and possible improvements. Libelium carefully listened to all these proposals and decided to bring out a new version with the name of Waspmote PRO (v1.2) in February 2013 [WSN Research Group, 2013]. This new version comes with upgraded hardware and an improved API[13], which is unfortunately not compatible with the older API. The most important improvements of the hardware is that the code can be uploaded much quicker and that the XBee radio must no longer be removed to do so. A new interrupt enables the XBee to wake the Waspmote PRO when XBee sleep modes are used [Waspmote (v1.1) vs Waspmote PRO (v1.2), 2013]. To do this on the first version one must solder pin 13 of the XBee to the MUX_RX pin of the Waspmote. This way interruptions caused by the XBee module can be captured, but all other interruption options (RTC[14], Accelerometer, etc.) will be masked by pin 13's output and will thus be lost. The new version no longer has jumpers and there is no need for a coin battery. Regarding the API, Libelium claims it is much more robust and easier to use than the previous one. One big advantage of the new API is the support to send AT commands to the XBee.

3.2 Hardware

3.2.1 Modular Architecture

As mentioned in the introduction, Waspmote is based on a modular architecture. Doing so optimizes costs and allows it to change according to the specific user's requirements. The available modules are split up into five categories: ZigBee, GSM - 3G/GPRS, GPS, Sensor Boards and Storage. Figure 6 indicates the Waspmote's main components.

[12] CE for Europe, FCC for the US and IC for Canada
[13] Application Programming Interface (API)
[14] Real Time Clock (RTC)

Figure 5: An Arduino XBee Shield (left), A Libelium SquidBee (middle) and a Libelium Waspmote V1.1 (right)

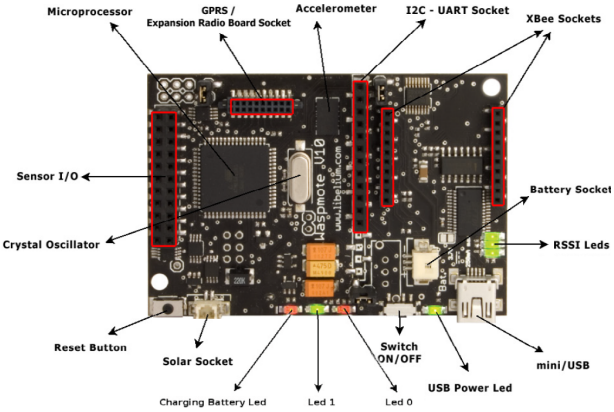


Figure 6: The main Waspmote components

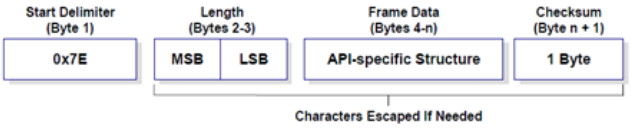


Figure 7: The UART Data Frame Structure

API Frame Name	API ID
AT Command	0x08
ZigBee Transmit Request	0x10
AT Command Response	0x88
ZigBee Receive Packet	0x90

Table 2: The API Frame Names and Values

3.2.2 XBee radio

The WSN nodes and the gateway make use of a XBee-ZB PRO radio. XBee is the brand name from Digi International for a family radio modules, based on the IEEE 802.15.4 standard [Digi International Inc, 2012]. The PRO radio is a higher power, longer range version of the XBee-ZB radio.
To communicate with the XBee radio there are two modes: AT (transparent mode) and API (application programming interface). AT mode means that what you send to the XBee radio using RS232, the XBee radio will send to its default destination. Unless you send '+++', wait for the XBee to reply with OK, and then send an AT command. AT commands are used to change the configuration of the XBee radio. For instance the AT command OP requests the operating PAN ID. An AT command can also be used to change the default destination.
AT mode is fairly limited and only good for point to point communication since you can't really specify the destination unless you change the default destination all the time. So that is why the sensor network operates in API mode. This means that everything sent to the XBee radio, using serial communication, is now packetized.
API defines a number of different packets as can be found in chapter 9 of [XBee/XBee-Pro ZB RF Modules User Manual, 2012]. An API packet is shown in figure 7. It starts with `0x7E` as a start delimiter and is followed by the length of the data, excluding the checksum. Then an API-specific structure follows, which depends on the type of the packet.
A reduced list of possible packets can be found in table 2 (for the full list please see appendix F). As mentioned, an AT command is used to alter the configurations of the ZigBee radio. This can of course also be done in API mode using the AT command packet. The details about all the packets can be found in the datasheet. The only packet types used in this project are: 'ZigBee transmit request, `0x10`' and ZigBee receive packet, `0x90`'. The reason only these packets are used is that Libelium puts its own structure inside the data of these packets. To send sample data not a 'ZigBee IO Data Sample Rx indicator' packet is sent but just a 'ZigBee transmit request, `0x10`' packet, with the sample data as data. It's not the XBee radio that takes samples. It's the Waspmote's ATmega processor that samples the sensors and does some processing on them to go from a voltage to a more familiar value. Then this data has to be sent to the gateway and the only way to achieve this is by sending a data packet.
A ZigBee transmit request is shown in figure 42. This packet is used to send data from this ZigBee radio to a remote one. The remote ZigBee address is all that has to be known. These types of packets are constructed by the gateway to send out data to the Libelium nodes but also by Libelium nodes to send data to other Libelium nodes or to the gateway. Libelium has its own specific format for the RF Data as explained in section 4.2.3 and as shown in figure 44. To reach the gateway, the address of the coordinator can be used, since the coordinator and gateway in our case are the same. This

is convenient since the coordinator can always be addressed with `0x0000000000000000`. The reason we chose the gateway as coordinator to be the same is that the coordinator receives a lot of traffic due to its role and the same goes for the gateway. So these two devices should be in the center of the network for efficiency reasons. Assigning one device for these two roles and trying to position this device as central as possible will ask for the least amount of routing overhead. However in this phase the position of the gateway is not optimal since it has to be stationed at module 14 for convenience. Module 14 is at the top of the building and packets coming from the ground floor will have to pass about two or three routers.
When data is received by a ZigBee radio, this radio will send out a ZigBee receive packet via its serial communication. An example of this packet can be found in figure 43. Again the received data has an additional format as specified in the Libelium Application header.

3.2.3 Microcontroller and memory

Because of the modular design of the Waspmote, the block diagram (see figure 32) is very simple. Waspmote integrates an 8-bit ATmega 1281 microcontroller with 128KB programmable flash, 8KB SRAM runtime memory and 4KB EEPROM memory. Since SRAM is built with cleverly combined MOSFETs it must not be periodically refreshed, but it is still volatile memory. The main advantage compared to DRAM is that, when moderately clocked like in the Waspmote, it consumes very little power.
The AVR[15] microcontroller was developed in 1996 by Atmel. It is a modified Harvard architecture 8-bit RISC[16] microcontroller and was one of the first microcontroller families to implement flash memory for program storage (opposed to other microcontrollers at the time that were using 1-time programmable ROM, EPROM or EEPROM[17]).
A Harvard computer architecture has separate storage and signal pathways for instructions and date. It can fetch instructions and data at the same time and can thus be faster than pure von Neuman architecture. Since most AVR instructions are 16 bit wide, the flash memory of the ATmega1281 with 128KB is organized as 64K x 16, so the Program Counter is 16 bits wide [Atmel ATmega 1281 Datasheet, 2012].
The architecture of the ATmega microcontroller modifies the Harvard architecture but the separate address space nature of a Harvard machine is preserved. In contrast to systems that add CPU cache, where data and instructions are unified, implementing the von Neumann model, the adaptation is much more subtle. The ATmega has an Extended Load Program (ELP) instruction which can allocate constant tables within the program memory address space (see figure 33). So the contents of the instruction memory can be accessed as data, saving scarce runtime memory. This however creates certain difficulties in programming, since the C Language was not designed for Harvard architectures (see section 4.4.2).

3.2.4 Timers

The Waspmote's system clock is an 8MHz quartz oscillator. This means that every 125 ns a machine language instruction is executed by the microcontroller. Keep in mind that one C++ instruction is several instructions in machine language.
To generate interrupts the Waspmote has an internal watchdog timer (WDT) and a Real Time Clock (RTC). The WDT is used to awake the Waspmote from *Sleep* mode, because of its high precision. Thus, *Sleep* mode allows small intervals, going from 16 milliseconds to a maximum of 8 seconds (see table 4). To store an absolute time base the RTC can be used. Alarms programmed in the RTC specify days, hours, minutes and seconds. This clock is used to wake the Waspmote from *Deep Sleep* or *Hibernate*[18], with intervals ranging from 8 seconds to even days.

3.3 Power considerations

3.3.1 Waspmote power modes

The Waspmote has 4 operational modes: ON, Sleep, Deep Sleep and Hibernate. They differ in which type of interruptions they can be woken up and duration interval. Table 4 summarizes the Waspmotes operational modes. The main difference between the sleep modes and the hibernate mode is that, in a sleep mode the program is paused. Whereas with hibernate mode, all the Wasmpote's modules, including the microcontroller, are disconnected from the main battery. Only the RTC is powered through the auxiliary battery and can wake the node. This means the CPU is also switched of and does not remember any values from variables. When waking up the Waspmote reinitializes, the microprocessor is reset and the program restarts from the beginning. Both **setup** and **loop** routines are executed as if the main switch would be activated. By placing the `ifHibernate()` function in setup the program can determine if it came from a hardware reset or from a hibernate reset.

3.3.2 Sampling sensors

To measure the sensors, we originally took 10 samples with 100 milliseconds recommended delay between the measurements. Since we want to make the energy consumption as low as possible we now do the iterations without delay. Appendix B.4 contains 60 test samples per sensor and indicates that there is no significant deviation on the average by removing this delay.

[15] It is commonly accepted that AVR stands for Alf (Egil Bogen) and Vegard (Wollan)'s RISC processor, after its developers [Wikipedia, 2013b].
[16] Reduced Instruction Set Computing (RISC)
[17] (Electrically) (Erasable) (Programmable) Read-Only Memory (EEP)ROM
[18] It is important to notice that in *Hibernate*, the RTC is no longer powerd through the main battery but through the auxiliary (button) battery. So when problems occur when using hibernate, it is recommended to measure the button battery's voltage and possibly it must be replaced.

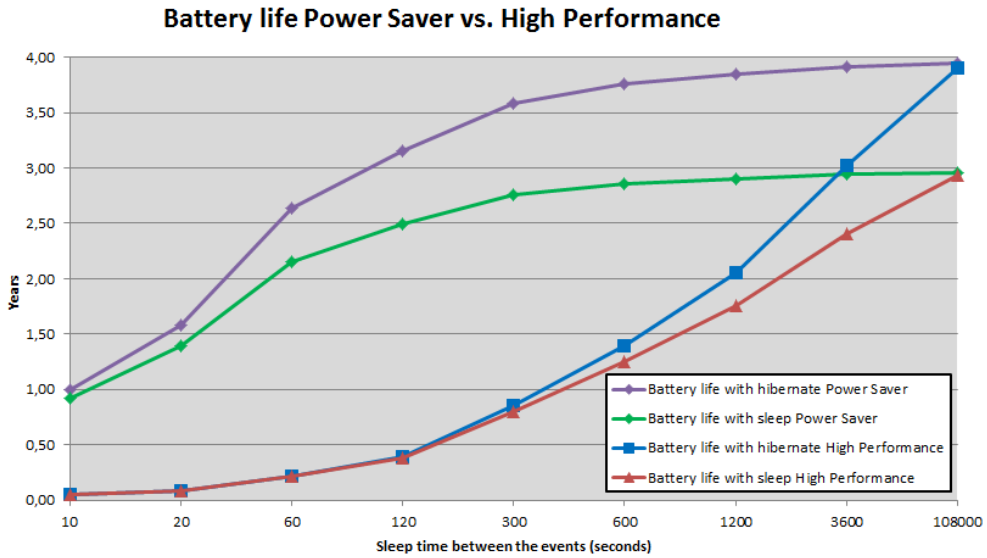


Figure 8: Battery life High Performance vs. Power Saver

3.3.3 Battery life estimation

In order to be able to provide recommended sensor measuring intervals this section will analyse the estimated battery life of the Waspmotes. Table 3 enumerates the typical consumption of a node's common components. The batteries included with our Waspmotes are rechargeable Lithium-ion batteries with a capacity of 6600mAH. Lithium-ion has a self-discharge rate of typcially 1 to 2 percent per month [Buchmann, 2013] and since the batteries are new we expect high battery efficiency.

Action	Average Current
Waspmote, ON	9mA
Waspmote, sleep	62μA
Waspmote, hibernate	0,007μA
XBee, ON	45mA
XBee, sending,	105mA
Temperature	6μA
Humidity	380μA
CO_2	50mA

Table 3: A Waspmote's typical power consumptions

The application scenario for this battery test is as follows: the Waspmote will be turned on as short as possible and 4 sensors, namely temperature, humidity, pressure and battery level will be sampled. The node will take 10 samples for each sensor and calculate the average. Those values are put into one ZigBee packet and are sent to the gateway. By adapting the sleep time between the event we came to the rather disappointing results shown in figure 34. Please see appendix D for more information.
Since the Waspmotes use this much energy when operating this way we will refer to it as the High Performance mode from now on. The next section calculates an alternative approach, referred to as the Power Saver mode. This nomenclature is continued in the program:

```
typedef enum{HIGHPERFORMANCE, POWERSAVER}
        PowerPlan;
```

3.3.4 Battery life optimizations

For end devices it is obviously recommended to turn on the XBee as little as possible, within a user defined limit.
Figure 8 shows the same results as the application scenario discussed in section 3.3.3 and adds the results for the Power Saver mode.
When the Power Saver mode is enabled, the Waspmote will awaken each time it has to sample data, but it will only send that data to the gateway once there are enough samples collected to send a maximum payload packet. Depending on whether one or two bytes are required per sample, 30 or 60 measurements can be taken before the Waspmote has to activate the XBee. In case the user requires a minimum responsiveness of the network there will also be an upper limit on the time a node may wait to send its collected samples.
As visible on figure 8, *Hibernate* has more influence in Power Saver mode, already extending battery life significantly at a 20 seconds interval compared to a 10 minutes interval in High Performance mode. Also the energy breakdown graph in figure 38 shows that the interval times must be increased much less before the dominant factor becomes self-discharge and sleep mode energy consumption, compared to figure 35.
Finally, although Over the Air Programming (OTAP) is supported on Waspmotes, it should be limited to a minimum in order to obtain reasonable battery durations. Writing to FLASH takes about 83nAH per byte, whilst reading only takes about 1.1nAH per byte [Fischione, 2011].

4 A WSN with Waspmotes: Implementation aspects

4.1 Introduction

In this section the program running on the WSN nodes will be discussed. To start programming, Libelium offers its customers a customized IDE[19] and a fairly extensive API. The IDE uses the same compiler and core libraries as the Arduino IDE and is ideal to upload small examples and test programs to the nodes. However, to facilitate programming and to obtain more C/C++ support, expanding the API is required. This will be discussed in more detail in section 4.5.2. After experimenting with the ZigBee sleep modes it became clear that the instability caused by delayed, pending messages for end devices could not be avoided. The program does support ZigBee sleep modes but this section will only discuss the *stable operation modes without ZigBee sleep*. From now on, we will no longer differentiate between ZigBee routers and ZigBee end devices, but the sleep options will be controlled via Waspmote sleep modes, as recommended by Libelium [Libelium-dev, 2013]. So a ZigBee router forced to sleep as well as a ZigBee end device will be considered an 'End Device'.

4.2 Overall Program Structure

When ZigBee sleep modes are ignored, basically three different programs suffice to program the entire network. There is one program to support the gateway (which is also the ZigBee coordinator) which analyses the data received from the other nodes, which is running on a Linux machine instead of a Waspmote. This will be discussed in section 5. All the nodes that collect sensor data are either running a 'Router' program or an 'End Device' program, which is designed to collect and send the sensor samples to the gateway. The only difference between these last two programs is that a 'Router' program does not implement sleep modes. The next sections will discuss the 'End Device' operation. A flowchart is shown in figure 9.
In a default program cycle each node has to associate with the network, measure some sensors, send the measured samples and possibly enter a sleep mode before repeating this cycle, depending on whether it has a 'Router' program or an 'End Device' program running.

4.2.1 Initialization

4.2.1.1 Device start-up: Full initialization When the program is executed for the first time, a full XBee setup will be executed with the parameters retrieved from the program in the Libelium IDE. To ensure stability it takes about 8 seconds to write the settings to the XBee, to reset it afterwards (turn the power off and back on) and finally to perform the joining process. From this moment on the node will send its battery level to the coordinator and wait to receive an 'ADD_NODE_REQUEST' packet, containing its physical sensor layout settings, from the web interface. Until such a packet is received the node's sleeping time will gradually be increased.

4.2.1.2 Next cycles: Reduced initialization By not resetting the PAN ID but simply fetching it from the XBee's memory, the joining process only takes about two seconds[20] (Please see appendix B.1 for more measurement results). Unfortunately a disadvantage of this shortened setup is that the XBee is no longer able to detect if the coordinator or his parent is actually available. The result of the executed association check is not always correct and the program will only notice this for the first time when it is trying to send a message. If this function results in a send error the program will do a full setup routine and resend the message. If the node then fails again to send the message we can conclude that the coordinator is really off-line or that there are no joinable nodes within range. In that case the message content will be saved and tried to be sent during the next cycles. In order not to lose the results, the gateway or web interface can keep track of the number of consecutive cycles a node has failed to report and notify a system administrator to avoid losing the saved values.

4.2.2 Measuring sensors

To sample a sensor from the expansion board, the Libelium API functions can be used. The functions return a float value corresponding to the sensor's measuring unit. Until an 'ADD_NODE_REQUEST' is received each node will measure its battery level and send it to the gateway. Via this packet the node gets to know its physical sensor layout and can decide if a future request of a sensor value is allowed.
To shorten messages (the ZigBee maximum payload is 74 Bytes [Waspmote ZigBee Networking Guide, 2012]) and to reduce transmission times, all values are sent in hexadecimal representation. This way all sensors fit within with 1 or 2 bytes. An example conversion is given for a temperature sample with two decimals:

```
float value_temp = 0;
byte temp[2];

// TEMPERATURE SENSOR RANGE: -40 -> +125
value_temp = readValue(SENS_TEMPERATURE);

value_temp += TEMPERATURE_OFFSET; //+= 40
value_temp *= 100; // 2 DECIMALS ACCURACY

temp[0] = MSByte(value_temp);
temp[1] = LSByte(value_temp);
```

[19] Integrated Development Environment (IDE)

[20] By enabling ZigBee sleep, the node does not lose the association with its parent and the re-joining process only takes about 10ms. However, the time needed to receive messages pending in router devices varies between 5 and 15 seconds and is never compensated by the faster joining process. There is also no guarantee that pending messages will be received or in which order they will be received, so network stability can no longer be guaranteed.

Mode	Consumption	CPU	Cycle	Accepted Interruptions
ON	9mA	ON	-	Synch and Asynch
Sleep	62μA	ON	31ms - 8s	Synch (WDT) and Asynch
Deep Sleep	62μA	ON	8s - min/hours/days	Synch (RTC) and Asynch
Hibernate	0.7μA	OFF	8s - min/hours/days	Synch (RTC)

Table 4: The operational modes of Libelium Waspmote V1.1

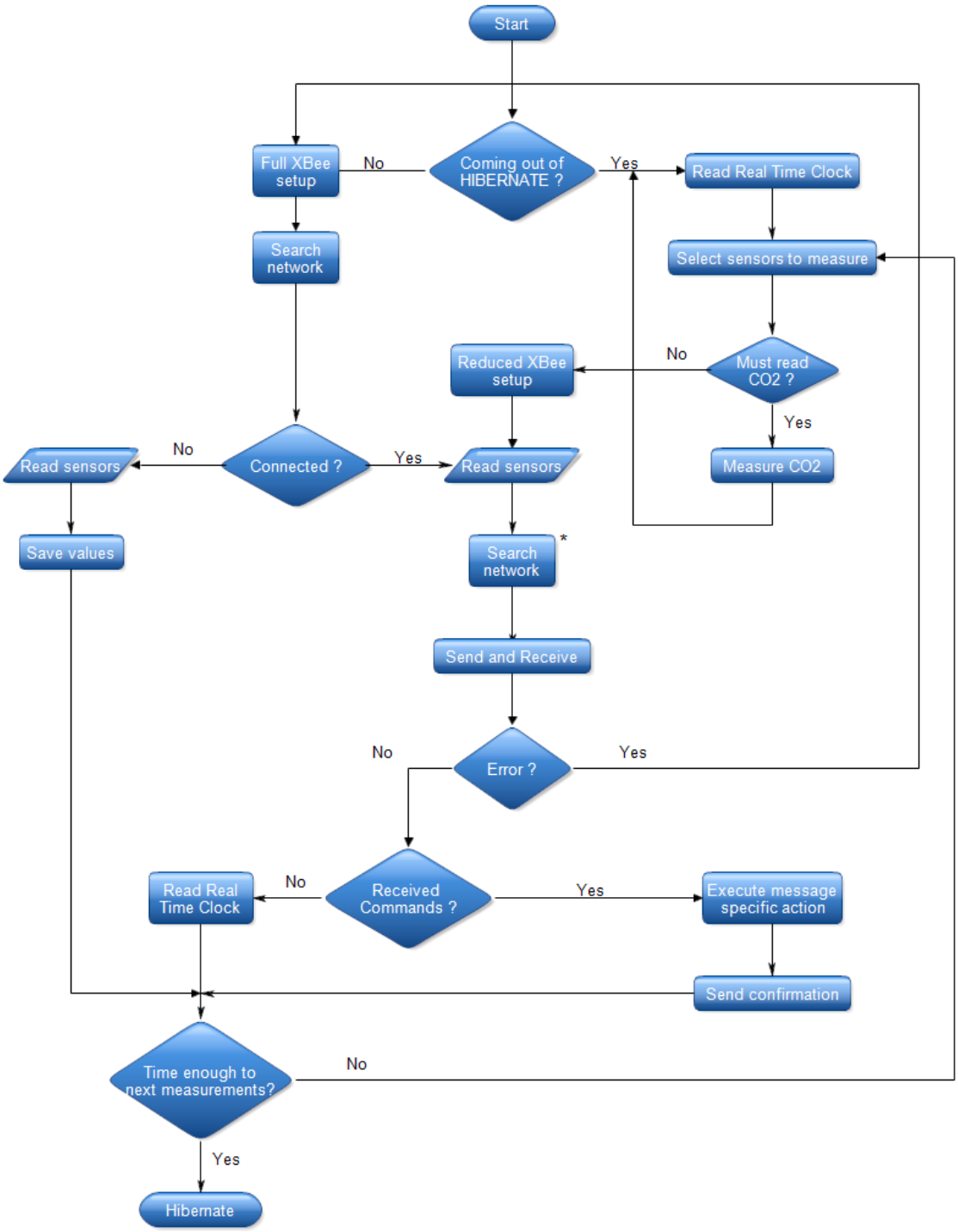


Figure 9: A flow chart of the Waspmote program for end devices

4.2.3 Sending samples

The API of Waspmote V1.1 introduces an 'Application Header' (see figure 10) to send and receive data. This header takes care of packet fragmentation if packets exceed the maximum payload limit and can also be used by the receiver to treat the packet or fragment. The header itself is sent inside the RF Data field of the API Frame Structure, which was discussed in more detail in section 3.2.2 (see figure 44).

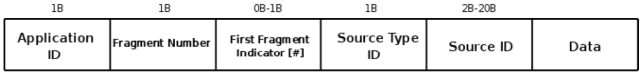


Figure 10: The application Header

The Waspmote and gateway programs use the 'Application ID' field in this header to distinguish different data packets. This extra layer also serves as an acknowledgement in the communication protocol. Depending on the 'Application ID' the layer contains a sensor mask (a 16 bit-flag) and data. With uniform timing settings, when a node awakens, it will measure the sensors found in its 'activeSensorMask' and send them to the gateway via an 'IO_DATA' packet. This packet will be indicated as 'IO_DATA' via the 'Application ID' field and will contain the mask of the measured sensors followed by their corresponding values.
To easily process this mask all sensor types are stored as follows:

```
typedef enum {TEMPERATURE = 0x0001,
    HUMIDITY = 0x0002, PRESSURE = 0x0004,
    BATTERY = 0x0008, CO2 = 0x0010, ... }
    SensorType;
```

The sensor mask of a packet as well as the active mask stored in a certain node is a 16-bit value in which each bit indicates if a sensor must be taken into account or not.

4.2.3.1 Escaping zeros Before sending the hexadecimal values all zeros must be removed (zeros are interpreted as end of string characters). Zeros will be replaced by `0xFFFE`, so an extra byte is necessary. `0xFF` is the chosen escape character and `0xFE` is added to distinguish between an escaped zero and the value `0xFF` itself. The escaped character itself is of course also escaped.

4.2.3.2 Constructing packets To easily introduce the sensors into the packet to be sent, the program uses static function pointers. Part of the code to insert the measured sensors into an 'IO_DATA' packet is given below. All Waspmote functions with error checking start with `uint8_t error = 2`. If the returned error equals 2, the function has not been executed. Whereas 1 means something went wrong and 0 indicates a success. Each function pointer knows the size of the sample and updates the `pos` variable, which determines the location of the sample in the 'IO_DATA' packet. This size and thus position depends on which sensor is included into the packet. The first two bytes of this packet are reserved for the sensor mask, indicating which sensor values are following. For example, an 'IO_DATA' packet with as mask `0x09` indicates to the gateway that the packet contains temperature and battery data.

```
uint8_t error = 2;
uint8_t pos = 2;
uint16_t indicator = 1;
for(int i=0; i<NUM_SENSORS; i++)
{
  if(indicator & sensor_mask)
  {
        (*Inserter[i])(&pos, packetData);
        error = 0;
  }
  indicator <<= 1;
}
```

4.2.4 Wait for received messages

To assure overall stability ZigBee sleep (see appendix A.3) is not used, meaning nodes can only receive messages when they are turned on. To coordinate this, the end devices will check all messages for a fixed amount of time immediately after sensor data has been sent. It allows the gateway to send all its pending messages for this node. This means the Waspmote will only wake up when sensors have to be measured.
To select the right function when a request packet is received, the program uses function pointers in the same way it adds data to packets. There are a few additions however. Firstly a check is done if the received Packet ID is a valid index in the array of function pointers. This way, when intruders send an invalid Packet ID, the program will not crash because it jumped to a wrong place in program memory. Secondly the node will check if the sending node is authorized by comparing the origin's MAC address in the API frame to its authorized nodes. By default, only the gateway XBee radio is authorized. For example, when an 'ADD_NODE_REQ' is received, some settings will be done and the node will send either an 'ADD_NODE_RES' or an 'SEND_ERROR' packet back to the gateway. Appendix F.5 shows a complete overview off all the Application IDs stored in the following enum:

```
typedef enum {ADD_NODE_REQ, ADD_NODE_RES,
    CH_SENS_FREQ_REQ, CH_SENS_FREQ_RES,
    IO_REQUEST, IO_DATA, ...}
    ApplicationIDs;
```

4.2.5 Enter low power mode

The user can set the device role ('Router' or 'End device'[21]) during setup or even later on by sending a command to the Waspmote, which makes it easy to change a mote's device role. The device role is stored in the xbeeZB device type

[21] This affects the Waspmote sleep mode. Setting 'Coordinator' role or changing the ZigBee device role is not possible because the XBee radio must be reflashed for this (The XBee's memory is to small to contain all profiles at the same time)

identifier (see [Waspmote ZigBee Networking Guide, 2012] and '**WaspXBeeZBNode.h**').
Several Waspmote sleep functions have been added to the Libelium API, the most advanced one combines *Sleep* and *Hibernate* depending on the duration of the next time to sleep. By doing this the number of EEPROM writes is minimized. In one of the sleep modes, the program is paused and the next time to wake can be fetched from SRAM memory. With hibernate however, the node is completely disconnected from the main battery and all program variables are lost, so they must either be stored in the onboard EEPROM memory or on the optional micro-SD card. To save the scarce free memory and to optimize power consumption the EEPROM option is chosen and the objects needed to control the SD-card have been removed from the API.
As indicated in table 4, only *Deepsleep* and *Hibernate* make use of the Waspmote's RTC, which is perfect to set the time to wake up and to select the sensors to measure. Unfortunately it is not possible to combine RTC usage for both *Hibernate* and *Deepsleep*, from the moment *Hibernate* is implemented the RTC can no longer be used for other functionalities. So a tweak has been implemented to use the Watchdog instead, making it possible to combine *Hibernate* and *Sleep*.
To determine the next time to sleep several algorithms can be used, depending on the used Waspmote sleep mode. Since we are working with embedded systems with limited possibilities, one should also consider to limit the users options in order to facilitate the calculations. When the program detects inefficient measuring intervals, for example, 1 minute and 2 minutes 10 seconds, this can be notified to the installer or even be refused during setup.Another one of those limitations is a maximum of 10 seconds time resolution, meaning each value in the algorithm must be multiplied by 10. Suppose we want to measure one sensor each 30 seconds, another one each 40 seconds and a last one every 100 seconds. This means the node has to wake up at each multiple of those measuring intervals. The absolute times to wake up will be at `0 3 4 6 9 10 12 15 18 20 21 24 27 28`.
Each time the Waspmote wakes up, it will compare its current RTC value with the stored times. The biggest stored time that is an integer multiple of the RTC value is the current position in the array. With this time we can determine which sensors to measure and what the next time to sleep will be. The shortest algorithm to calculate these values is given below. From the moment the calculated times reach the least common divider (LCM) of the measuring intervals the algorithm can stop. Recursion is nice, however, without compiler optimization this is not recommended for embedded devices. To execute the code literary we would need extra instructions and memory for each function call[22], which can easily lead to stack overflows in case of embedded devices. Since this algorithm is tail recursive the recursive calls can simply be replaced by a loop, eliminating the overhead[23].

[22]Each call requires a stack frame, containing the function parameters, return address and possibly local variables.

[23]GCC with O3 optimization recognizes tail recursion and will do this for us

```
uint16_t pos = 0;
for(uint16_t i=1; i<LCM; i++)
{
  if( isTime2Wake(i) )
    times2wake[pos++] = i;
}

bool isTime2Wake(uint16_t & value)
{
  static uint8_t i = 0;
  if( !value % sortedIntervals[i] )
    {i = 0;  return true;}
  else if( ++i == nrActiveSensors )
    return false;
  isTime2Wake(value);
}
```

4.2.6 Router program

The 'Router program' is based on the same concepts discussed in the previous subsections, which were describing 'End Device' operation. The main difference is that a router node will not implement sleep possibilities and attention will be divided differently.
A 'Router program' will continuously check for new requests and will be interrupted by the RTC when it has to measure and send sensors. Initialization, measurements and sending samples is the same as for end devices.

4.3 Weather Station Program

Although the previous section stated there are only two different program versions running on the mobile nodes, this section will briefly discuss one more variant. The program running on the weather station, installed on the roof of Group T Campus Vesalius (see figure 11), is actually a specialized version of an 'End Device' program.
Firstly, because the weather station is equipped with a solar panel for energy harvesting, enabling the hibernate interruptions to save power consumption is no longer required. This means the RTC interruptions are fully available to determine the sampling intervals.
The weather station Waspmote is also equipped with an 'Agriculture Sensor Board' instead of a 'Gasses Sensor Board'. This means, for example, to sample the temperature sensor, different API functions must be used. Unfortunately, when the program creates both those objects (see section 4.5.2.1), none of the sensor readings are correct any more. To solve this and to save program memory at the same time, conditional pre-processor directives are used. To compile for the weather station node, `#define WEATHER_STATION` must be uncommented in 'BjornClasses.h'.
The weather station program also has an extra interrupt routine attached. This interrupt is triggered each time the pluvio meter becomes full, incrementing the pluvio counter. When this happens for the first time within one hour, a started raining notification will be sent to the gateway. Afterwards, the pluvio meter counter can be read to know the millimetres of rainfall since the first interrupt occurred.

Figure 11: The weather station deployed on the roof of Group T, campus Vesalius

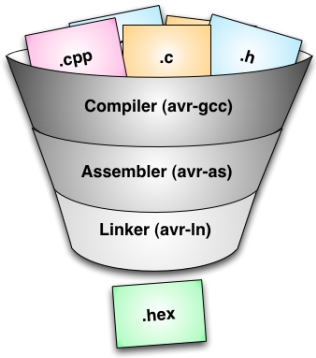


Figure 12: An overview of the AVR toolchain

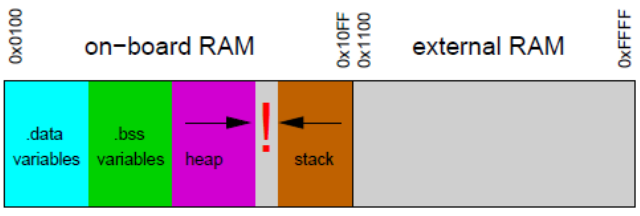


Figure 13: The AVR / ATmega1281 standard RAM layout

4.4 AVR compiler

4.4.1 Toolchain Overview

To develop software for an AVR microcontroller several tools are working together. This group of tools produces the final executable and is commonly called a toolchain and is shown in figure 12.

4.4.1.1 GCC: AVR uses the open source GNU Compiler Collection (GCC) with AVR microcontroller as target system. This version of GCC is known as 'AVR GCC' [AVR-Libc User Manual, 2008]. GCC differs from other compilers, it only focuses on translating high level language to target assembly. For AVR GCC there are 3 language options: C, C++ and Ada.

4.4.1.2 GNU Binutils: The next step is done by another open source project called GNU Binutils. This contains the GNU assembler and GNU linker.

4.4.1.3 AVR-libc: GCC and Binutils provide the tools to make the machine code but one critical component they do not provide is the Standard C Library. The open source AVR toolchain therefore comes with its own open source C Library project which contains many of the same functions found in the regular Standard C Library. It also adds many additional library functions that are specific to AVR microcontrollers.

4.4.1.4 GNU Make: Finally all pieces must be tied together. This is done by Make, which interprets and executes the Makefile of the project.

4.4.2 Memory Sections

The available non-volatile memory sections are the **.text** section (FLASH), which contains the actual machine instructions and the **.eeprom** section. Many AVR devices have a minimal amount of RAM. This limited amount of runtime memory needs to be shared between the following memory sections:

1. All initialized variables and static data such as `char message[] = "An error message"` or `USB.println("Another message")` are stored in **.data variables**
2. Uninitialized global or static variables: **.bss variables**
3. Dynamic memory: **heap**
4. Area used for calling subroutines and storing local variables: **stack**

The standard RAM layout is shown in figure 13. Since there is no hardware supported memory management, separate regions can overwrite each other. Heap and stack can collide if either of them require large memory space or even when the allocations aren't high at all but because heap allocations get fragmented over time and new request don't fit in freed areas.
As discussed in section 4.4.2 the ATmega1281 is uses a modified Harvard architecture, meaning that data can also

be stored in program memory space. This is useful when you have constant data and you're running out of room to store it. Remember that many AVRs have limit amount of RAM to store data, but may have available FLASH space left. For the compiler this is however a challenge, which is exacerbated by the fact that the C Language was designed for Von Neumann architectures. So the AVR compiler has to use other means to operate with these separate address spaces (cf. pointer usage). The AVR toolset used the GCC `__attribute__` keyword, which is used to attach different attributes to function declarations and variables. AVR GCC provides a special attribute called `progmem` for data declarations and tells the compiler to store the data in Program Memory. To increase the convenience to the end user AVR-libc provides a simple macro `PROGMEM` which can be found in 'avr/pgmspace.h'. To read the data another macro is provided, which generates the correct address to retrieve the data from Program Memory. Storing data in Program Space incurs extra overhead in terms of instructions and execution time, but usually this is minimal compared to the space savings.

4.4.3 Memory problems

Libelium's Programming Style Guide warns its users about the amount of memory `USB.print("Test message!")` requires. The program memory increases due to the instructions and arguments (the chars) needed to print the string. Since also the message itself is stored in the **.data** section [AVR-Libc User Manual, 2008], precious memory is wasted. As a fix, Libelium recommends to do the following:

```
#define Message "Test Message"!
USB.println(Message);
```

This however still uses both program and data memory and is only useful if one wants to print the same message in different parts of the program, so this is not really a solution. The only way to save RAM memory while printing messages is to hard code them on the stack by doing the following:

```
char str[5] = {'t','e','s','t','\0'};
USB.println(str);
```

A less cumbersome way would be to give the message and an address where to store the message as argument of a recursive macro which does this operation for us. However, standard C macros cannot simply split a string into characters. So the only ways to save RAM is to hard code the string as data or to store the string in Program Space and use the `strcpy_P` to copy the string to stack when it is needed.

4.4.4 Compiler bugs

When combining all pieces of software discussed in section 4.2, the Waspmote failed after a few cycles. It appeared as if Libelium's `sendXBee(packetXBee *)` function had a constant memory leak of 302 bytes. However, after examining all the related functions it became clear there were no implementation errors. After putting the necessary `USB.print(freeMemory())` statements before and after `free()` instructions, we concluded the `free()` instruction at line 5252 of 'WaspXBeeCore.cpp' failed[24]. This was avoided by introducing a stack variable instead of a heap variable.
Another bug we discovered is with the assignment operator. For example assigning a `uint16_t` stack variable, that was needed for some evaluation, to a global `uint16_t` variable also fails sometimes for no clear reasons. Instead of using the temporarily stack variable, we re-assigned the evaluation to the global variable.

4.5 Libelium IDE and API

4.5.1 Waspmote-IDE

The Libelium IDE offers some advantages compared to using other IDE's. For example by using Eclipse it is possible to update programs that are to big thereby over-writing the bootloader. Then they must be sent back to Libelium to restore them. The Waspmote IDE does not allow this accident, so Libelium does not support using other IDE's in an official way so that there is no valid warranty if you've erased the bootloader.

4.5.2 Waspmote-API

To facilitate the programming Libelium offers a quite big API and after some exploring you are quickly started with it. The structure of the API is very simple, there is little to no inheritance[25]. In this section, the most important classes are indicated with a red box and are recommended to explore before starting future programming.

4.5.2.1 Original structure Each module or concept has its own class, for example all RTC functions are in 'WaspRTC.h' and 'WaspRTC.cpp'. To be able to use those functions in the IDE an object from the class 'WaspRTC' must be created. This object is created by default by the library and it is public to all libraries. All types to be run on the API can be found in the 'WaspClasses.h' file and each file also includes this file so it is aware of all available types. For our application WaspXBeeZB is one of the most important classes. It inherits from WaspXBee Core and this way it is also related to the WaspXBee class. Figure 14 displays the relationship with the AVR-libc libraries and the Waspmote hardware.
During the development of our program the Waspmote showed several strange effects that could only be explained by bad stack management or heap and stack conflicts. Because of this lack of free memory (SRAM, 8KB) we discovered that the V1.1 API wastes a lot of memory by always

[24] Issue 857: avr-libc 1.6.4 dynamic memory (malloc, free) bug, available at `https://code.google.com/p/arduino/issues/detail?id=857`, confirms this conclusion

[25] Have a look at the graphs available at `http://www.libelium.com/v11-files/api/waspmote/db/d09/WaspClasses_8h.html` to get a complete overview of the API structure

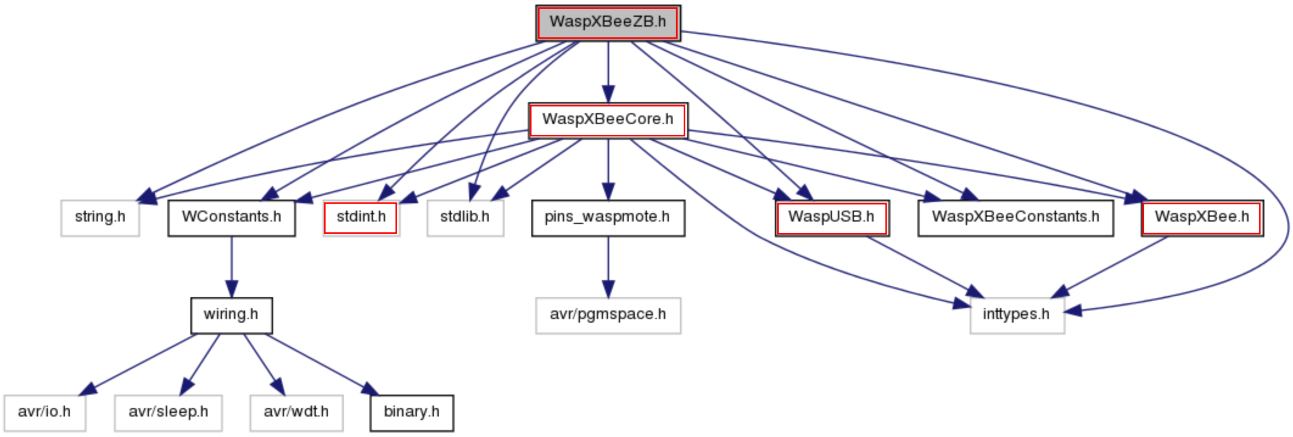


Figure 14: Reduced dependency of Waspmote core libraries

including all libraries, despite not using them. As a fix Libelium recommends via their forum to remove all classes you do not use, including there fields that are used in other classes, 'just' going through all the compiler errors one by one. After doing this our program had enough free memory and showed normal behaviour again.

4.5.2.2 Added functionality

To facilitate programming extra functionality has been added to the Libelium API. They are inside files containing the original name with the 'Utils' addition, for example 'WaspRTCUtils.h' and can be found in 'BjornClasses.h'.

5 Extracting the data

5.1 Gateway requirements

The gateway consists of a XBee radio connected to a computer via RS232. This computer runs a separate gateway program which will be explained in this section. The computer will evolve into an embedded Linux device such as the Raspberry Pi [The Raspberry Pi Foundation, 2011].
It is important that it runs on Linux since some libraries are necessary and the serial communication is based on system calls to the Linux kernel. For instance, xerces 3.1.1 [Xerces developers, 2010] library for XML parsing or boost 1.49 [Boost developers, 2013] for multi-threading and some other small features.
The library for the web server is Mongoose and the one for the SQL database is SQLite [QSLite developers, 2013]. Both libraries are written in C and consist out of a limited amount of files and are both compiled into the final program. So unlike Xerces and Boost the OS will not need to install these libraries since they are not dynamically linked.
RS232 is a simple serial protocol that is used for low data rates. In Linux an RS232 connection is easily set up by opening a file descriptor with the necessary options to configure baud rate, parity, stop bits, etc[Sweet, 1999]. After setting up the connection read and write functions can be used to receive and to send packets to and from the XBee ZigBee radio.

5.2 Programming language choice

Before starting to program, a few decisions had to been made: which programming language and which operating system? One of the requirements was that the program could run on an embedded device. Linux has more options in the embedded domain than windows. For instance Raspbian, which is optimized for Raspberry Pi [Raspbian developers, 2013]. The Raspberry Pi is a cheap embedded device with an Ethernet connection, 2 USB ports, a 700mhz ARM processor and which can run different Linux distributions.
The gateway program has also been tested on a Raspberry Pi running Raspbian. Going from a normal PC to a Raspberry Pi is really easy and was done without any major issues.
As programming language we chose for C++ since we have some experience in this programming language and it has some libraries which are not available in C. C++ also facilitates developing for example because of the built in types like String. The objects and classes allow for easier structuring of the program.

5.3 Gateway program

5.3.1 Overall Program Structure

For the gateway it took some time to come up with an optimal program structure[26]. Since a gateway generally has to wait a lot for incoming messages on different connections, it is logical to construct threads which can wait while other threads keep on running.
One option would be to use the *pipeline pattern* where one thread 'generates' packets. This would be the ZigBee receiver or the web service. Then a few threads would act as filters on these packets, doing the necessary processing. The last thread would be Ipsum to send data out. However, one simple pipe would not work since there are more threads generating packets. The web service as well as the ZigBee sender are generating packets. There is not really one flow of information going from one place to another. Information is coming in at different places and is also leaving the pro-

[26]The entire UML-diagram and code of this program can be found on Github at: `https://github.com/Silverslide/Qt_ZigbeeWSN`

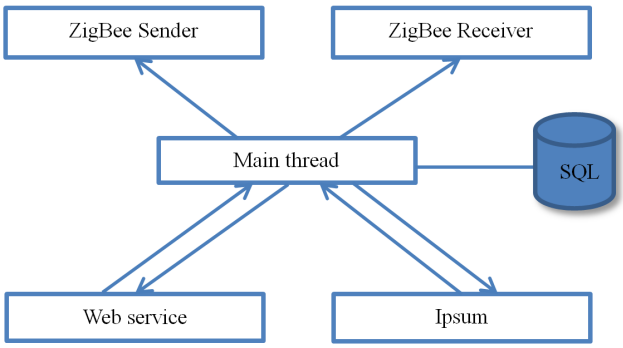


Figure 15: Gateway thread pool model

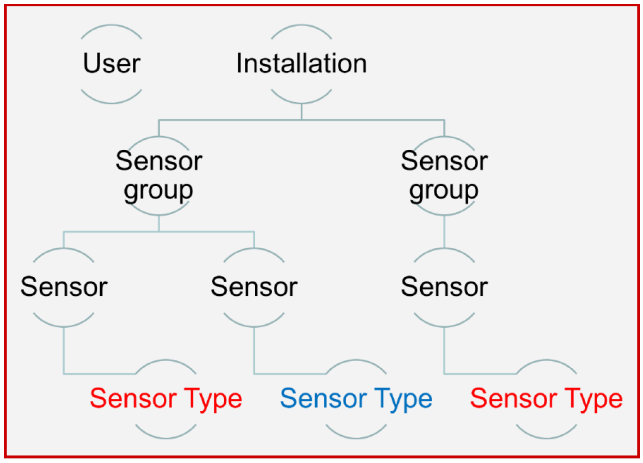


Figure 16: The hierarchical design of Ipsum

gram at different places. The web service is not only generating packets, it also needs to be able to receive replies from the main thread. Ipsum on the other hand will most often receive packets but there are scenarios in which the Ipsum thread sends some packets to the main thread.
A more flexible pattern is the *thread pool pattern*. This pattern is often used for web servers where each thread handles an incoming connection. This is not entirely the same for the gateway program. The gateway has a few threads, one for each type of connection. These threads are all different, whereas on a web server these threads are mostly copies of the same code. The gateway consists of a main thread that receives packets from different threads and sends out other packets. It sends these packets to different threads which handle the input and output of the gateway. Figure 15 displays this structure. The transfer of packets between threads happens via queues which are protected by mutexes. These queues contain objects of a child class of Packet. All the different packets can be found in the UML on github. Since this is a gateway for ZigBee and ZigBee itself is not supposed to deal with large amounts of data, no special priorities have been set to different threads nor have the queues been limited in size or have any possible bottlenecks(they shouldn't exist) been analysed. The only optimization for multithreading that has been done, is using local queues to store packets before they are processed. So the pointers to the packets from the shared queues are copied to the local queues. Then processing is done on the packets found in these local queues. Now the shared queues are only being read while the packet pointers are being copied and not while the packets are being processed. So other threads will have to wait less for these shared queues.
The IO-threads in charge of dealing with correct processing of incoming and outgoing information, are: Ipsum, ZBSender, ZBReceiver and Webservice. They will be discussed in the next subsections.

5.3.2 IO-threads

5.3.2.1 ZigBee On the ZigBee side there are 2 threads, one for sending packets and one for receiving packets. Received packets are pushed onto a queue that is read out in the main thread. Received packets can be sensor data, errors and responses from sent packets. Again all different packets can be found in the UML on github. When the main thread needs something to be done in the ZigBee network it will push a packet onto the queue going to the ZigBee sender thread. For instance: change sensor sample frequencies, request IO data, activate a node.

5.3.2.2 Ipsum Ipsum is the name of a storage system developed by Group T. In the gateway the Ipsum thread is thus responsible for sending data to this storage system. To do so it uses a HTTP connection and POST or GET messages where the data consist of XML [Raspbian developers, 2013]. Ipsum differs from other storage systems. It has virtual sensors to which data can be uploaded. Sensors can be grouped into sensor groups and those in can turn be grouped into installations. Figure 16 shows Ipsum's internal layout. For a WSN this is exactly the structure needed. A node has different sensors so a node can be mapped on a sensor group in Ipsum. For each 'floor' in Group T a different installation is used. All these structures in Ipsum are generated by the web application. A sensor also has an 'InUse' and a 'Frequency' field by default. 'InUse' will be set to true when the ZigBee network confirmed that a sensor has been added successfully. The 'Frequency' field is actually the interval in seconds between measurements. Also a sensor group has an 'InUse' field this is also enabled when a node is successfully added.
Since Ipsum is a remote storage system it must be taken into account that the connection to Ipsum or Ipsum itself, can fail. Therefore local caching in the Ipsum thread is done. When Ipsum is available again all cached packets are sent out.
An improvement can be to check the available memory. In case memory is almost full the packets should be sent back to the main thread and stored there in the SQL database. Or a separate SQL database can be used so that the Ipsum thread can do this storage itself.

5.3.2.3 Web service The web service is a simple REST[27] service which makes use of the Mongoose library. Mongoose sets up some threads (by default 20) to handle incom-

[27] Representational State Transfer (REST)

ing connections [Valenok, 2013]. These threads wait for incoming HTTP requests. The type of command can be found in the URL and the necessary data can be found in the POST data and is formatted in XML. The XML is read in one of the web service threads and if nothing goes wrong a HTTP 202:Accepted, is sent back to the client. If the command does not exist or the XML is invalid an error is replied. Also a key has to be added to the URL to allow for some sort of authentication. Since the key shouldn't be read by anyone or its use would be futile, the messages are encrypted using SSL. So in fact the web service uses HTTPS instead of HTTP.
Possible commands are: add node[28], add sensor, request IO and change frequency. For a complete list and a detailed format of these requests please see appendix F.6.

5.3.3 Main thread

All intelligence can be found in the main thread. This is where the decisions are made. What to do with incoming packets? Most often this means creating new packets and putting them in the appropriate queues.
Since it is possible that packets going into the ZigBee network are not received nor responded to correctly, a list of sent packets is kept in the main thread. This is only done for packets we expect a reply from. For the 'request IO data' packet this is not the case. This packet is sent out and it is not replied to. Only the next time data is sampled for that node, also sensor data for sensors specified in the 'request IO data' packet is provided. But if an 'add node' or 'change frequency' packet does not receive a reply, it will be resent. The time used to expire and amount of resends done, can be set on startup. These replies are generated by the Waspmotes but also ZigBee can send back acknowledgements. If this acknowledgement is not successful then the remote radio will not have received the packet and also a resend can be done.
But delivering the packet to the remote radio is not sufficient. It's not because the remote radio received the packet that it has accepted it correctly. So therefore also a reply from the Waspmote is expected before the delivery can be assumed successful. A failed acknowledgement is usually faster than the timeout of a sent packet. But an acknowledgement alone is not sufficient. So this is why both factors are taken into account to resend packets.
The matching between an acknowledgement and the original sent packet is done by frame ID. This is an unsigned char given to each packet and when it differs from zero an acknowledgement will be sent. This frame ID can overflow, however this is not a problem unless so many packets are sent that there are now two packets with the same frame ID waiting for a response. Then the strategy is to take the last sent packet since it is most likely that this packet received a reply. If the number of resends and the expiration time of sent packets is low enough this scenario will almost never occur. Only admins can send 'add node' and 'change frequency' packets. So these admins should send more than 255 packets in the time equal to expiration time X number of resends. Normal values for expiration time are 10 or 15 seconds. The number of resends is usually below three. So this problem only occurs if more than 255 packets are sent in less than one minute.
The main thread also has access to local storage in the form of an SQLite database. Node information has to be stored locally in order to link ZigBee addresses to the correct Ipsum IDs. When a sensor data packet is received, the main needs to look up to what Ipsum sensors it has to upload this data. The relation between Ipsum ID and ZigBee address is made when the web service receives add node and add sensor requests.
Also errors could be logged in the database. For now errors are printed to `stderr` (standard error stream) which is coupled to a file.

[28] This is the 'ADD_NODE_REQUEST' packet mentioned in section 4.2.1.

6 Discussion

WSNs can change everyone's life drastically and can be applied to many different areas. WSNs can help reduce CO_2 emissions for example. In a city where each parking space logs its occupation, a car can be guided to the nearest free spot. We have no doubt this technology has a bright future.

7 Future Work

The WSN Test-bed can be extended with the newer version of the Waspmote. Since the new Waspmotes provide an interrupt which enables the XBee to wake the Waspmote PRO, these ones should be used as end devices and the current versions should be used as routers.
In the preparation phase of this thesis we also created a simple XBee ZigBee module which runs on two AA batteries and is able to monitor a few sensors (see appendix A.6). Since these nodes are much cheaper than Libelium's hardware it would be nice if these nodes could also be integrated into the network. Unfortunately, because developing with the first generation Waspmotes took longer then expected, we could not test this any more. To realize this, one should extend the Libelium API with a function that is also able to read default API packets (without needing Libelium's extra application header).
At this moment, the test-bed will only accept request by authorized nodes (by default this is only the gateway). It would be a nice challenge to send the data encrypted as extra security. One must take in mind that this takes extra processor power and thus power consumption.
The gateway has mostly been tested on a computer with plenty of memory and processing power. Optimizing this program for an embedded device is certainly possible. For instance, packets that could not be sent to Libelium are cached in memory. On embedded devices memory is limited so when memory is getting full these packets should be stored in a database until Ipsum is back online.
On the web service part it would be good to send a more detailed reply back to the client. Now only XML and URL are checked but if a node doesn't exist this is only detected

in the main thread and the client is not notified of this error. The problem is that there are 20 threads to handle incoming requests. If the main detects a problem with a request it is hard to tell to which thread it should reply this problem since all the web service threads put packets in only one queue.
There is no way to communicate back to the client when something happens in the ZigBee network. If on Ipsum a sort of notification system could be installed where the gateway can put announcements or errors and the client would poll these announcements some more information about the status of the network could be delivered. For instance the pluvio sensor, which detects rain, works on interrupt basis. When this interrupt occurs for the first time in a longer period of time, we can assume it started raining. This could then be communicated back to the client trough an announcement. Or if the client requested an add node or add sensor but the ZigBee failed to reply in a certain amount of time it could be announced that this packet failed to deliver.
The program also has an SQL database. When this database fills up there is no strategy as what to when we run out of storage space. There are other and better database structures to deal with this.

8 Conclusion

During this thesis we successfully developed the WSN Testbed, the gateway and the web client. To further expand the sensing capabilities of the WSN, we added a waterproof weather station mounted on a custom fitted stainless steel construction to guarantee long term operation.
For the ZigBee network the main challenges were power consumption and the responsiveness towards the users. To achieve low power consumption, long sleep periods are necessary, however this deteriorates the response time of the network. ZigBee has several good sleep options and supports wake on interrupt thus combining low power consumption with a high responsiveness. The first generation Waspmote does not support this, but the second generation does. By integrating the second generation Wasmpote PRO into the network, end devices can be made more responsive, increasing the overall network performance. ZigBee as a low power, low throughput networking protocol lives up to its expectations.
Also while developing the gateway different challenges were faced. Different services had to be connected to one another. Establishing the program structure required an iterative approach. Once this design was known, a decision had to be made on how to transfer information between the threads. We developed a lightweight solution using queues to achieve this. The gateway also provides certain features to avoid data loss. Since the gateway had to be multithreaded concurrency issues are avoided by using the necessary tools from the Boost library. Since the gateway is exposed to the outside world so that the client can access it, HTTPS is used as a safe means of communication. To verify the requirement that the gateway software can run on an embedded device, we successfully ported the program to a Raspberry Pi. This is a cheap solution in anticipation of a more permanent server.
The overall result is a controllable WSN test-bed in which new nodes and sensors can easily be integrated. Also the sensor data can be consulted in different formats via the web interface. This is the result of a good cooperation between all parties involved.

9 Acknowledgements

We would like to thank many people who have helped us with the completion of this dissertation. First of all we want to thank our supervisor, Luc Vandeurzen, for his guidance on technical as well as non-technical matters.
We are also thankful to Koen Pelsmaekers, who helped putting requirements together for the entire WSN, Ipsum and the web application project. To Ruben Tacq, who developed and helped us on our way with Ipsum.
We are beyond grateful to Matthias Verhelst, who worked on the development of the web interface for the WSN test-bed and for the cooperation in combining the different projects.
To our parents for all there support, and especially Albert Deraeve, for transforming the weather station model into an exquisite construction.
Furthermore we would like to thank the people at Libelium for their advice on the Waspmote sketches.
Finally we want to thank Group T, for offering us the Libelium University Lab Kit and infrastructure to test and deploy our results.

References

21 ideas for the 21st century. *Business Week*, Aug. 30:77–167, 1999.

AVR-libc, 1.6.4 edition, 2008.

Cristina Alcaraz, Pablo Najera, Javier Lopez, and Rodrigo Roman. *Wireless Sensor Networks and The Internet of Things: Do We Need a Complete Integration?* Computer Science Department University of Malaga, Spain, 2010.

8-bit Atmel Microcontroller with 64K/128K/256K Bytes In-System Programmable Flash. Atmel, 2549p edition, 2012.

Ilangko Balansingham and Qinqhua Wang. *Wireless Sensor Networks: Application - Centric Design*. Yen Kheng Tan, 2010. Available from: `http://www.intechopen.com/books/wireless-sensor-networks-application-centric-d wireless-sensor-networks-an-introduction`.

Robert Berger. *Introduction to Wireless Sensor Networks*. NI Technical Symposium, 2009.

Isodor Buchmann. Elevating self-discharge. *Battery University*, 2013. `http://batteryuniversity.com/learn/article/elevating_self_discharge`.

Sebastian Buttrich. Wireless sensor networks, 2010. Course Lecture SPVC2010.

Patrick deGategno and Banning Garret. The second wave of wireless communications: A game changer for global development? *Atlantic CouncilNational Intelligence Council*, October 2010.

Boost Developers. Boost c++ libraries. 2010a. Available at http://www.boost.org.

Raspbian developers. Welcome to raspbian. 2013. Available at: http://www.raspbian.org/.

SQLite Developers. Sqlite. 2013. Available at http://www.sqlite.org/.

Xerces Developers. Xerces-c++ xml parser. 2010b. Available at http://xerces.apache.org/xerces-c/.

Dynamic C: An Introduction to ZigBee. Digi International Inc., 2008. Available on http://www.rabbit.com.

XBee XBee-PRO ZB RF Modules. Digi International Inc., 90000976l edition, April 2012.

Sahan Farahani. *ZigBee Wirlesss Networks and Transceivers*. Elsevier, 2008.

Carlo Fischione. *An Introduction to Wireless Sensor Networks*. KTH Royal Institute of Technology, November 2011.

Kenneth Flamm. *Creating the Computer*. The Brookings Institution, 1988.

The Raspberry Pi Foundation. Raspberry pi. 2011. Available at: http://www.raspberrypi.org/.

WSN Research Group. Libelium launches new generation of waspmote sensor nodes. April 2013. http://www.sensor-networks.org/index.php?page=1309909904.

Cooking Hacks. Wireless sensor networks open source platform. February 2013. http://www.cooking-hacks.com/index.php/documentation/tutorials/waspmote/.

Libelium. 50 sensor applications for a smarter world. get inspired! May 2012. http://www.libelium.com/50_sensor_applications/.

Waspmote ZigBee Networking Guide. Libelium Comunicaciones Distribuidas S.L., 1.0 edition, November 2012.

Waspmote (v1.1) vs Waspmote PRO (v1.2). Libelium Comunicaciones Distribuidas S.L., 4.0 edition, February 2013.

in response to Bjorn Deraeve Libelium-dev. Sleep options and buffers. 2013. Available at http://www.libelium.com/forum/viewtopic.php?f=17&t=10432.

Libelium Comuniaciones Distribuidas S.L. Squidbee main page. http://www.libelium.com/squidbee/index.php?title=Main_Page.

Michael R. Sweet. *Serial Programming Guide for POSIX Operating Systems*, 1999. Available at http://www.cmrr.umn.edu/~strupp/serial.html.

Arduino Team. Xbee shield, February 2012. http://arduino.cc/en/Main/ArduinoXbeeShield/.

Valenok. Mongoose - easy to use web server. 2013. Available at https://code.google.com/p/mongoose/.

Wikipedia. Auto-id labs. March 2013a.

Wikipedia. Darpa. May 2013b.

Wikipedia. Atmel avr. April 2013c.

Marco Zennaro. Wsn in china. *Wireless Sensor Networks Blog*, January 2008.

A ZigBee Additions

A.1 ZigBee Stack

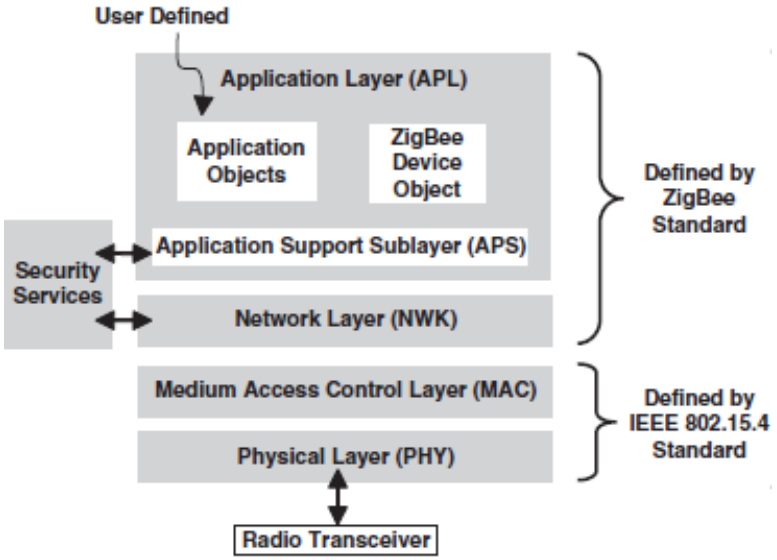


Figure 17: ZigBee Protocol Layers (↻ 1.4.2)

A.2 Logical Device Types

ZigBee Network Layer Function	Coordinator	Router	End Device
Establish a ZigBee network	•		
Permit other devices to join or leave the network	•	•	
Assign 16-bit network addresses	•	•	
Discover and record paths for efficient message delivery	•	•	
Discover and record list of one-hop neighbors	•	•	
Route network packets	•	•	
Receive or send network packets	•	•	•
Join or leave the network	•	•	•
Enter sleep mode			•

Figure 18: Comparison of ZigBee Devices at the Network Layer (↻ 2.2)

A.3 ZigBee Sleep Mode

A.3.1 Waspmote V1.1

As mentioned in section 4.1, the current Waspmote program has the possibility to enable ZigBee / XBee sleep via the following definitions, accessible in 'PowerUtils.h' and the Libelium IDE, but will not use ZigBee sleep in default configurations.

```
typedef enum{END_DEVICE, ROUTER, COORDINATOR}
        DeviceRole;

typedef enum {HIGHPERFORMANCE, POWERSAVER}
        PowerPlan;

typedef enum {SLEEP, DEEPSLEEP, HIBERNATE, NONE}
        SleepMode;

typedef enum {XBEE_SLEEP_ENABLED, XBEE_SLEEP_DISABLED}
        XBeeSleepMode;
```

This decision is based on the next conclusions related to the `XBeeSleepMode`:

1. **Waspmote V1.1 has no support for wakening from ZigBee cyclic sleep**: When the Waspmote and the XBee are in sleep, the XBee should be able to wake the Waspmote when it detects his parent has a message pending for it. This is possible via a new interrupt routine added to Waspmote PRO. To do this on the first version one must solder pin 13 of the XBee to the MUX_RX pin of the Waspmote. This way interruptions caused by the XBee module can be captured, but all other interruption options will be masked by pin 13's output and will thus be lost.
2. Enabling ZigBee sleep removes the association delay (see section B.3) but **requires 5 - 20 extra seconds to retrieve messages which are pending in the parent's buffer**. This has a very negative impact on battery life since all this time the XBee is turned on and receiving.
3. There is no guarantee that pending messages will be found when the Waspmote checks for them or in which order they will be received, so **network stability can no longer be guaranteed**. For example, when a privileged web interface user changes a sensor's measuring interval at minute 0 and another user re-changes the interval at minute 1, it is possible the request of the first user will be received after the request of the second user, so the node will not have the correct settings. This could be solved by adding timestamps to the requests or comparing the nodes responses with the web interfaces values.
4. **Adding timestamps can help, however it is still possible that requests will go lost**. This happens from sleeping times of 15 seconds and more. This means that firstly the node will be much more active than sleeping. Secondly, the goal of using ZigBee sleep was to speed up the network response time and since packets can go lost due to enabling ZigBee sleep the network's response is worse than without using ZigBee sleep.

By checking for received messages immediately after sending sensor data, we can:

1. Guarantee network stability and disable ZigBee sleep
2. Save battery life by:
   - Shorter duty cycles because we know when a message can be received
   - No XBee sleep current

A.3.2 Waspmote PRO

A sleeping Waspmote PRO can be interrupted when its sleeping XBee notices its parent has RF data available. The next paragraphs will briefly discuss the ZigBee protocol and settings..

A.3.2.1 ZigBee Sleep: Managing End Devices ZigBee end devices are intended to be battery powered and are capable of sleeping for extended periods of time. Because of this, routers and coordinators use packet buffers and transmission timeouts to ensure reliable data delivery to end devices.
When an end device joins the network, a parent-child relationship is formed with a router or the coordinator. From then on, if the end device is awake, it will send poll request messages (by default every 100 ms) to its parent to determine if the parent has any data buffered for it, independent of the sleep mode. Routers buffer this data only up to 28 - 30 seconds, so if we want to ensure reliable communication, this is the maximum sleep time. The child poll timeout can however be set up to a couple of months, so an end device can sleep longer than 30 seconds and still be considered to be in the network. This includes the node is associated within a few milliseconds, compared to the 2.5 seconds mentioned in section D.1. End devices can choose between two sleep modes, discussed in the next sections.

A.3.2.2 Pin sleep In this mode an external microcontroller controls when the XBee should sleep and when it should wake by controlling pin 13. The module will not respond to serial or RF data when it is sleeping.

+ lowest power consumption
+ external controller can take samples without powering up the radio
- ZigBee protocol has less control
- external controller's timer is not accurate enough to synchronize the network
- Need fully awake parent

A.3.2.3 Cycle sleep Allows the XBee to determine when to wake up. The module can sleep for a specified time and wake for a short time to poll its parent for buffered data. If the parent has date the device will remain awake for a time, otherwise it will re-enter sleep mode immediately.

- + suitable for DigiMesh, where the sleep clocks is accurate enough to get all nodes awake at the same time
- + with DigiMesh, fully awake routers are not required, so they can be battery powered
- - more power consumption due to accurate clock
- - external controller must also wake when the XBee wakes to treat potentially received messages, even if there is no need to sample data

A.3.2.4 XBee sleep parameters Router/Coordinator Configuration:

- RF Packet Buffering Timeout:
  - Sleep Period (SP) parameter. Max 30 seconds.
  - For cyclic sleep devices: SP should be set the same on routers and coordinators as it is on cyclic sleep end devices
  - For pin sleep devices: SP should be set equal to the time the end device can sleep, up to 30 seconds. If an end device sleeps longer than 30 seconds, parent and possibly non-parent devices must know when the device is awake. Therefore and devices that sleep longer than 30 seconds should transmit some kind of data (API frame) to alert the other devices that they can send data to the end device.
- Child Poll Timeout: Sleep Number (SN) parameter: The number of Sleep Periods (SP) used to calculated end device poll timeout.

A.4 Sleeping Mesh

What is *really* low power about ZigBee? The answer is already deducible from the previous sections, namely End Devices! A sleeping mesh tries to establish a multi-hop mesh network and low power routing functionality in one system, also using battery-powered routers. The system has two big requirements however:

- very low bandwidth, high latency
- static network

For example, delivery of one data packet from every node every 12-24 hours, with a wake period of about 15 seconds. In a sleeping mesh all nodes wake up simultaneously and periodically to exchange and or route data. Afterwards, all nodes except the coordinator go back to sleep, a state which they are in 99% of the time. In such a set-up battery lifes of 10 years and more are easy to reach.

A.5 Range Test

This procedure describes the easiest way to perform a range test using X-CTU. The test requires one coordinator node and one router node configured in AT mode.

1. Set the target module address in both radios, as indicated in figure 19.
   - item Module A will have the address of B in its destination field.
   - Module B will have the address of A in its destination field.

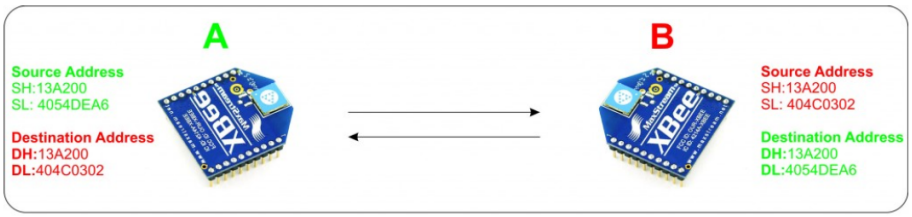


Figure 19: XBee Address Exchange

2. Make sure the modules have the same PAN ID and CHANNEL settings and write the values to the XBees.
3. To do the range test, a data packet will be send and the module expects the same packet to be received:
   - Connect the sending module to your laptop, e.g. module A.
   - Module B must not be connected to any device but its Rx and Tx pin (XBee pin 2 and 3) must be connected to each other in order to create the loopback connection. To do this you can use a small breadboard. See figures 20 and 21.

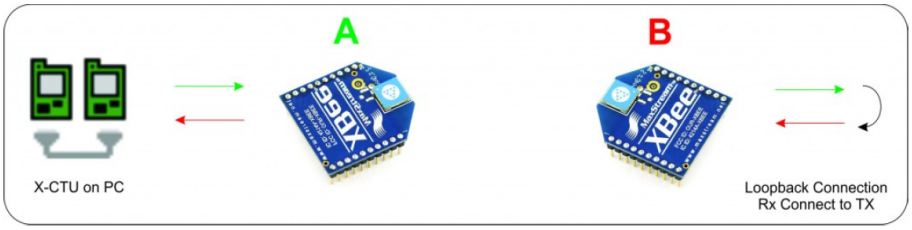


Figure 20: XBee Loopback Test

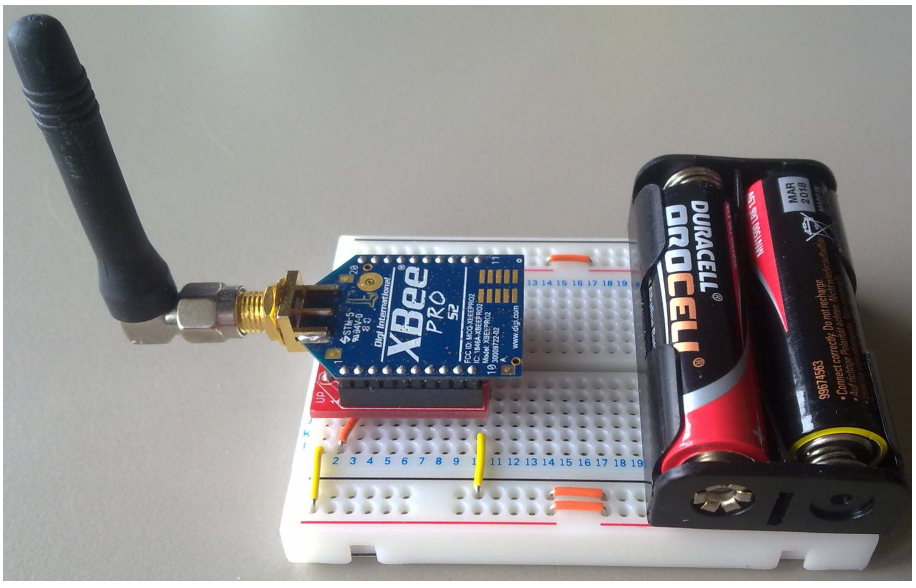


Figure 21: Module B Loopback Connection

4. Go to the *Range Test* tab in X-CTU, **select the checkbox under RSSI** and click *Start* to run the test. Leave module B at a fixed position and walk around with your laptop to immediately see the RSSI value. Your screen should look as indicated in figure 22.

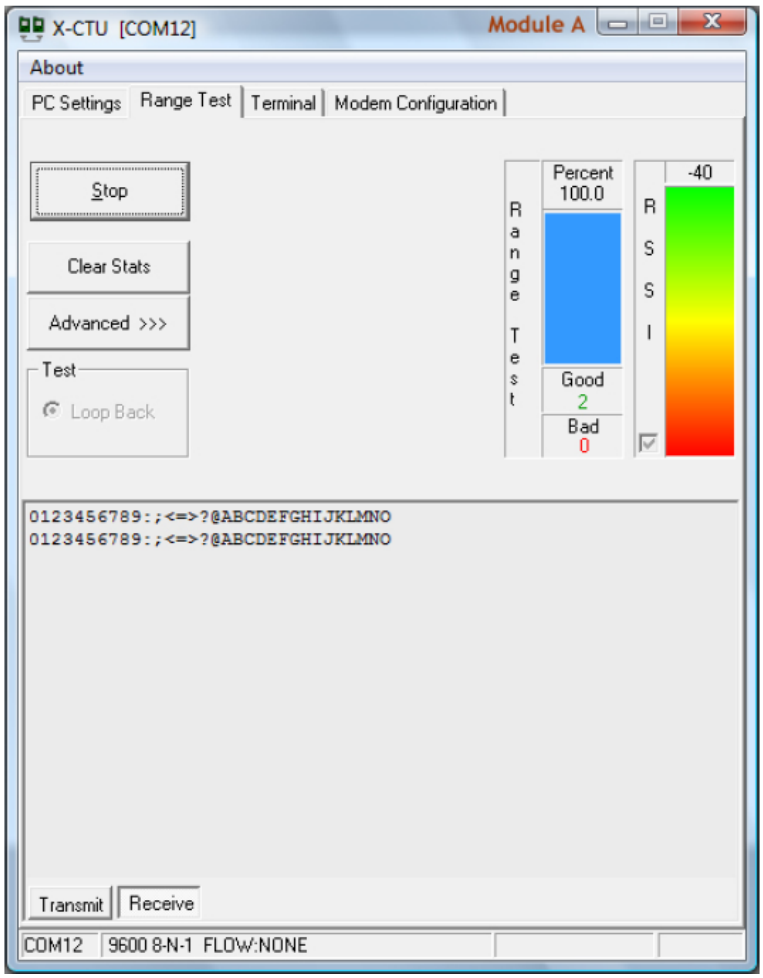


Figure 22: X-CTU Range Test

A.6 Self-made mobile XBee ZigBee node

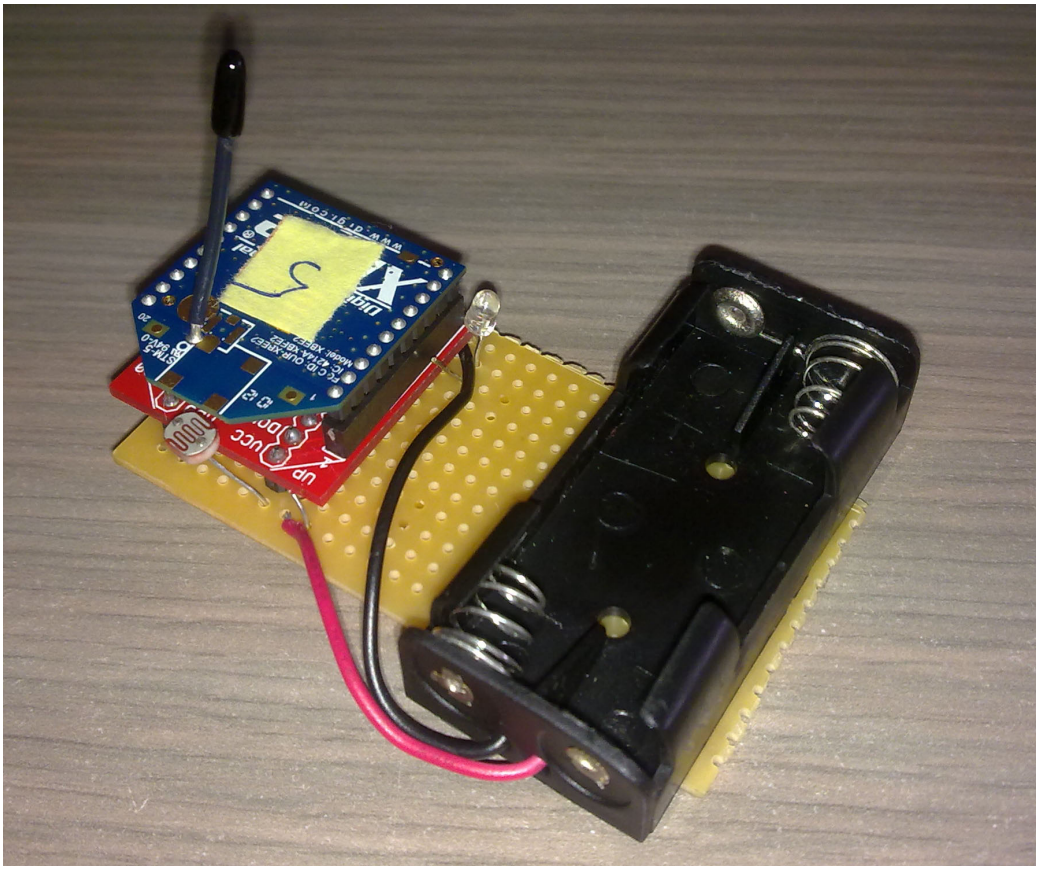


Figure 23: A simple mobile node

B Network Stability Experiments

B.1 Distance relation test

This table contains the results of two different test programs (blue and green values). The program scenario is as follows:

1. Turn on Waspmote and XBee from *Hibernate*
2. Wait until XBee has joined (Reduced setup mode)
3. Measure battery level, temperature, humidity and pressure (each with 100 milliseconds delay between samples)
4. Send the values and turn off Waspmote (enable *Hibernate*)

We can conclude that the join time remains constant but the send time and energy consumption increase when more obstacles must be overcome.

DISTANCE	ON	JOINED	MEASURED	SENT	JOIN TIME	SENDING TIME	
1 FLOOR	26	3082	7325	7461	3056	136	
	26	3082	7325	7481	3056	156	
	26	3080	7323	7760	3054	437	
	26	3082	7325	7467	3056	142	
	26	3082	7325	7467	3056	142	
	26	2462	6705	6738	2436	33	
	26	2462	6705	6742	2436	37	
	26	2460	6703	6864	2434	161	
	26	2462	6705	7677	2436	972	
	26	2462	6705	7279	2436	574	AVG SEND
	26	2462	6705	6866	2436	161	268

Figure 24: Measurements of join and send time at different distances.

2 FLOORS	26	3082	7325	8154	3056	829	
	26	3082	7325	7351	3056	26	
	26	3082	7325	7527	3056	202	
	26	3082	7325	7911	3056	586	
	26	3082	7325	7355	3056	30	
	26	3082	7325	7357	3056	32	
	26	2462	6705	6738	2436	33	
	26	2462	6705	6878	2436	173	
	26	2462	6705	8313	2436	1608	
	26	2462	6705	6738	2436	33	
	26	2462	6705	7256	2436	551	**AVG SEND**
	26	2460	6703	6889	2434	186	357
3 FLOORS	26	3082	7325	7485	3056	160	
	26	3082	7325	8933	3056	1608	
	26	3082	7325	8156	3056	831	
	26	3082	7325	7488	3056	163	
	26	3082	7325	8935	3056	1610	
	26	3082	7325	7467	3056	142	
	26	3080	7323	7469	3054	146	
	26	3082	7325	7471	3056	146	
	26	3082	7325	7461	3056	136	
	26	3082	7325	7463	3056	138	
	26	3082	7325	7467	3056	142	
	26	2462	6705	6738	2436	33	
	26	2460	6703	6878	2434	175	
	26	2460	6703	8313	2434	1610	
	26	2462	6705	6738	2436	33	
	26	2462	6705	7256	2436	551	
	26	2462	6705	6889	2436	184	
	26	2462	6705	6872	2436	167	**AVG SEND**
	26	2462	6705	7917	2436	1212	484
4 FLOORS	26	3082	7325	8419	3056	1094	
	26	3082	7325	8933	3056	1608	
	26	3082	7325	7461	3056	136	
	26	3080	7323	7461	3054	138	
	26	3082	7325	7475	3056	150	
	26	3082	7325	7758	3056	433	
	26	3082	7325	7469	3056	144	
	26	3082	7325	8943	3056	1618	
	26	3082	7325	7471	3056	146	
	26	3080	7323	7463	3054	140	
	26	3082	7325	8322	3056	997	
	26	3082	7325	7461	3056	136	
	26	2462	6705	6848	2436	143	
	26	2460	6703	6868	2434	165	
	26	2462	6705	6868	2436	163	
	26	2462	6705	6845	2436	140	
	26	2462	6705	8724	2436	2019	
	26	2460	6703	7902	2434	1199	**AVG SEND**
	26	2462	6705	6738	2436	33	558
5 FLOORS	26	3082	7325	ERROR	3056	ERROR	
	26	2462	6705	ERROR	2436	ERROR	

Figure 25: Measurements of join and send time at different distances

B.2 First time reduction

The next results are obtained via the following scenario:

1. Turn on Waspmote and XBee
2. Measure battery level, temperature, humidity and pressure (each with 100 milliseconds delay between samples)
3. Check XBee association (Reduced setup mode)
4. Send the values and turn off Waspmote

It appears that when the XBee has been on for a sufficient amount of time, the time to request the node's association state is constant at about 450ms.
A second conclusion is that without obstacles and with a node that already is joined a while before trying to send leads to a constant sending time.

ON	MEASURED	JOINED	SENT / HIBER	JOIN TIME	SENDING TIME	
26	4742	5194	5309	452	115	
26	4742	5194	5307	452	113	
26	4742	5192	5305	450	113	
26	4742	5194	5307	452	113	
26	4742	5194	5307	452	113	
26	4742	5194	5307	452	113	
26	4742	5194	5309	452	115	
26	4742	5194	5303	452	109	
26	4742	5194	5305	452	111	
26	4742	5194	5266	452	72	
26	4742	5194	5309	452	115	
26	4742	5194	5677	452	483	
26	4742	5194	5788	452	594	
26	4742	5194	5437	452	243	
26	4742	5194	5307	452	113	
26	4742	5194	5307	452	113	
26	4742	5194	5307	452	113	
26	4742	5194	5266	452	72	
26	4742	5194	5309	452	115	AVG SEND
26	4742	5194	5309	452	115	158

Figure 26: XBee join and send times: first time reduction

B.3 Second time reduction

The next results are obtained via the following scenario:

1. Turn on Waspmote and XBee
2. Measure battery level, temperature, humidity and pressure (without delay between samples)
3. Check XBee association (Reduced setup mode)
4. Send the values and turn off Waspmote

The sensors are measured after 708ms. However it takes about 2.5 seconds for the XBee to join the network. The total average time the Waspmote is turned on is 3 seconds.

ON	MEASURED	JOINED	SENT / HIBERNATE / ON TIME	SENDING TIME
26	708	2462	2702	240
26	708	2462	2702	240
26	708	2460	2570	110
26	708	2462	2576	114
26	708	2462	2995	533
26	708	2460	4147	1687
26	708	2460	2698	238
26	708	2462	4149	1687
26	708	2462	2574	112
26	708	2462	2702	240
26	708	2462	4147	1685
26	708	2462	2570	108
26	708	2462	2704	242
26	708	2462	4149	1687
26	708	2462	2574	112
26	708	2462	2574	112
26	708	2462	2570	108
26	708	2462	3668	1206
26	708	2462	2531	69
26	708	2462	4151	1689
		AVG:	3073	611

Figure 27: XBee join and send times: second time reduction

B.4 Sensor measuring experiments

B.4.1 Temperature measurements with and without delay between the readings

Typical consumption	6 μA
Time needed with delay	1050ms
Time needed without delay	47ms

Experiment 1	
100 ms delay	**no delay**
25,4838676452	25,4838676452
26,1290340423	25,4838676452
25,1612911224	25,4838676452
25,1612911224	25,1612911224
25,1612911224	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,1612911224	25,4838676452
26,1290340423	25,4838676452
25,4838676452	25,4838676452
Range:	**Range:**
0,9677429199	0,3225765228
Average:	**Average:**
25,4838703155	25,4516099929
Std. Dev	**Std. Dev**
0,3533697514	0,0967729568

Experiment 2	
100 ms delay	**no delay**
25,4838676452	25,4838676452
25,1612911224	25,1612911224
25,1612911224	25,1612911224
25,4838676452	25,1612911224
25,1612911224	25,4838676452
25,1612911224	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,4838676452
25,1612911224	25,1612911224
Range:	**Range:**
0,3225765228	0,3225765228
Average:	**Average:**
25,2258064270	25,2580640792
Std. Dev	**Std. Dev**
0,1290306091	0,1478231333

Experiment 3	
100 ms delay	**no delay**
25,4838676452	25,4838676452
25,1612911224	25,4838676452
25,4838676452	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,4838676452
25,4838676452	25,8064517974
25,4838676452	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,4838676452
Range:	**Range:**
0,3225765228	0,6451606750
Average:	**Average:**
25,2903217315	25,3548377990
Std. Dev	**Std. Dev**
0,1580295768	0,2139746687

Experiment 4	
100 ms delay	**no delay**
25,4838676452	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,4838676452
25,4838676452	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,1612911224
25,1612911224	25,1612911224
Range:	**Range:**
0,3225765228	0,3225765228
Average:	**Average:**
25,2258064270	25,1935487747
Std. Dev	**Std. Dev**
0,1290306091	0,0967729568

Experiment 5	
100 ms delay	**no delay**
26,1290340423	25,4838676452
25,1612911224	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
Range:	**Range:**
0,9677429199	0,0000000000
Average:	**Average:**
25,5161266326	25,4838676452
Std. Dev	**Std. Dev**
0,2258071899	0,0000000000

Experiment 6	
100 ms delay	**no delay**
25,8064517974	26,4516105651
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,4838676452	25,4838676452
25,8064517974	25,4838676452
Range:	**Range:**
0,3225841522	0,9677429199
Average:	**Average:**
25,5483844756	25,5806419372
Std. Dev	**Std. Dev**
0,1290336609	0,2903228760

Figure 28: Temperature measurements with and without delay between the readings (↺ 3.3.2)

B.4.2 Humidity measurements with and without delay between the readings

Typical consumption	0.38mA
Time needed with delay	1049ms
Time needed without delay	48ms

Experiment 1	
100 ms delay	**no delay**
23,4477977752	23,4477977752
23,7946548461	23,7946548461
23,2743625640	23,6212272644
23,2743625640	23,4477977752
23,6212272644	23,1009368896
23,1009368896	23,4477977752
23,6212272644	23,6212272644
23,4477977752	23,6212272644
23,7946548461	23,4477977752
23,6212272644	23,2743625640
Range:	**Range:**
0,6937179565	0,6937179565
Average:	**Average:**
23,4998249053	23,4824827194
Std. Dev	**Std. Dev**
0,2200583592	0,1867899847

Experiment 2	
100 ms delay	**no delay**
22,9275016784	23,2743625640
23,4477977752	23,1009368896
23,1009368896	22,9275016784
23,6212272644	23,2743625640
23,1009368896	23,6212272644
23,1009368896	23,4477977752
23,2743625640	23,2743625640
23,1009368896	23,1009368896
23,1009368896	23,1009368896
23,1009368896	23,4477977752
Range:	**Range:**
0,6937255860	0,6937255860
Average:	**Average:**
23,1876510620	23,2570222854
Std. Dev	**Std. Dev**
0,1939013430	0,1969796132

Experiment 3	
100 ms delay	**no delay**
23,2743625640	23,2743625640
23,6212272644	23,1009368896
23,2743625640	23,2743625640
23,7946548461	23,6212272644
23,6212272644	23,4477977752
23,4477977752	23,2743625640
23,2743625640	23,2743625640
23,4477977752	23,1009368896
23,6212272644	23,4477977752
23,1009368896	23,6212272644
Range:	**Range:**
0,6937179565	0,5202903748
Average:	**Average:**
23,4477956771	23,3437374114
Std. Dev	**Std. Dev**
0,2052059897	0,1768654913

Experiment 4	
100 ms delay	**no delay**
23,2743625640	23,4477977752
23,4477977752	23,2743625640
23,4477977752	23,1009368896
23,4477977752	23,4477977752
23,2743625640	23,7946548461
23,6212272644	23,4477977752
23,4477977752	23,4477977752
23,6212272644	23,1009368896
23,6212272644	23,4477977752
23,6212272644	23,6212272644
Range:	**Range:**
0,3468647004	0,6937179565
Average:	**Average:**
23,4824825286	23,4131107330
Std. Dev	**Std. Dev**
0,1297845810	0,2022524232

Experiment 5	
100 ms delay	**no delay**
23,7946548461	23,2743625640
23,2743625640	23,6212272644
23,4477977752	23,7946548461
23,2743625640	23,6212272644
23,6212272644	23,2743625640
23,4477977752	23,1009368896
23,7946548461	23,4477977752
23,7946548461	22,7540760040
23,6212272644	23,6212272644
23,7946548461	23,4477977752
Range:	**Range:**
0,5202922821	1,0405788421
Average:	**Average:**
23,5865394592	23,3957670211
Std. Dev	**Std. Dev**
0,2022527830	0,2907220514

Experiment 6	
100 ms delay	**no delay**
22,4072113037	22,2337837219
22,4072113037	22,4072113037
22,5806446075	22,5806446075
22,4072113037	22,7540760040
22,7540760040	22,5806446075
22,2337837219	22,4072113037
22,4072113037	22,2337837219
22,7540760040	22,5806446075
22,5806446075	22,7540760040
22,7540760040	22,5806446075
Range:	**Range:**
0,5202922821	0,5202922821
Average:	**Average:**
22,5286146164	22,5112720489
Std. Dev	**Std. Dev**
0,1742966051	0,1768656409

Figure 29: Humidity measurements with and without delay between the readings (↺ 3.3.2)

B.4.3 Pressure measurements with and without delay between the readings

Typical consumption	0.7mA
Time needed with delay	1053ms
Time needed without delay	51ms

Experiment 1	
100 ms delay	**no delay**
102,0491104125	100,3384628295
100,4606475830	100,7050247192
100,4606475830	100,5828552246
100,8272171020	100,5828552246
100,3384628295	100,4606475830
100,4606475830	100,2162704467
100,5828552246	100,8272171020
100,8272171020	100,7050247192
100,3384628295	100,5828552246
100,4606475830	100,4606475830
Range:	**Range:**
1,7106475830	0,6109466553
Average:	**Average:**
100,6805915832	100,5461860656
Std. Dev	**Std. Dev**
0,4850773019	0,1732335658

Experiment 2	
100 ms delay	**no delay**
101,3159790039	100,5828552246
100,4606475830	100,4606475830
100,7050247192	100,4606475830
100,5828552246	100,3384628295
100,7050247192	100,7050247192
100,3384628295	100,5828552246
100,4606475830	100,5828552246
100,4606475830	100,4606475830
100,5828552246	100,3384628295
101,0715942382	100,8272171020
Range:	**Range:**
0,9775161744	0,4887542725
Average:	**Average:**
100,6683738708	100,5339675903
Std. Dev	**Std. Dev**
0,2894105125	0,1466279350

Experiment 3	
100 ms delay	**no delay**
101,8047485351	100,5828552246
100,9494094848	100,5828552246
101,8047485351	100,4606475830
100,3384628295	100,3384628295
100,3384628295	100,8272171020
100,9494094848	100,7050247192
101,3159790039	100,5828552246
100,8272171020	100,4606475830
100,4606475830	100,3384628295
100,4606475830	100,8272171020
Range:	**Range:**
1,4662857056	0,4887542725
Average:	**Average:**
100,9249732971	100,5706245422
Std. Dev	**Std. Dev**
0,5320555163	0,1679820361

Experiment 4	
100 ms delay	**no delay**
101,3159790039	100,5828552246
100,3384628295	100,5828552246
100,4606475830	100,4606475830
100,5828552246	100,3384628295
100,3384628295	100,8272171020
100,4606475830	100,5828552246
100,7050247192	100,5828552246
100,3384628295	100,4606475830
100,4606475830	100,4606475830
100,5828552246	100,3384628295
Range:	**Range:**
0,9775161744	0,4887542725
Average:	**Average:**
100,5584045410	100,5217506408
Std. Dev	**Std. Dev**
0,2775616244	0,1366142138

Experiment 5	
100 ms delay	**no delay**
101,0715942382	100,3384628295
100,5828552246	100,3384628295
100,5828552246	100,3384628295
100,4606475830	100,3384628295
100,3384628295	100,4606475830
100,4606475830	100,4606475830
100,4606475830	100,5828552246
100,3384628295	100,7050247192
100,3384628295	100,8272171020
100,2162704467	100,0940780639
Range:	**Range:**
0,8553237915	0,7331390381
Average:	**Average:**
100,4850906372	100,4484321594
Std. Dev	**Std. Dev**
0,2239772597	0,2004059897

Experiment 6	
100 ms delay	**no delay**
101,3159790039	100,2162704467
100,4606475830	100,2162704467
100,0940780639	100,3384628295
100,3384628295	100,3384628295
100,5828552246	100,4606475830
100,4606475830	100,4606475830
100,5828552246	100,5828552246
100,2162704467	100,7050247192
100,4606475830	100,2162704467
100,8272171020	100,2162704467
Range:	**Range:**
1,2219009400	0,4887542725
Average:	**Average:**
100,5339660644	100,3751182556
Std. Dev	**Std. Dev**
0,3242068755	0,1643901459

Figure 30: Pressure measurements with and without delay between the readings (↻ 3.3.2)

B.4.4 Pressure measurements with and without delay between the readings

Time needed with delay	1014ms
Time needed without delay	11ms

Experiment 1	
100 ms delay	**no delay**
97	96
96	97
97	96
96	97
97	93
96	96
97	97
96	96
96	97
97	96
Range:	**Range:**
1	4
Average:	**Average:**
97	96
Std. Dev	**Std. Dev**
0,50	1,14

Experiment 2	
100 ms delay	**no delay**
97	96
97	97
97	94
96	96
97	96
97	87
96	96
96	97
92	96
98	96
Range:	**Range:**
6	10
Average:	**Average:**
96	95
Std. Dev	**Std. Dev**
1,55	2,81

Experiment 3	
100 ms delay	**no delay**
97	97
96	96
97	96
97	97
96	96
96	97
96	97
96	89
96	96
97	97
Range:	**Range:**
1	8
Average:	**Average:**
96	96
Std. Dev	**Std. Dev**
0,49	2,32

Experiment 4	
100 ms delay	**no delay**
97	97
96	97
97	96
97	96
97	97
96	95
96	96
97	97
97	88
96	96
Range:	**Range:**
1	9
Average:	**Average:**
97	96
Std. Dev	**Std. Dev**
0,49	2,58

Experiment 5	
100 ms delay	**no delay**
97	97
97	97
96	96
96	97
96	97
97	90
96	96
97	97
97	97
97	96
Range:	**Range:**
1	7
Average:	**Average:**
97	96
Std. Dev	**Std. Dev**
0,49	2,05

Experiment 6	
100 ms delay	**no delay**
97	96
97	97
96	97
97	97
97	96
97	97
95	97
97	97
97	89
91	97
Range:	**Range:**
6	8
Average:	**Average:**
96	96
Std. Dev	**Std. Dev**
1,81	2,37

Figure 31: Battery measurements with and without delay between the readings (↺ 3.3.2)

C Waspmote Additions

C.1 Waspmote Architectural Overview

C.1.1 Waspmote Block Diagram

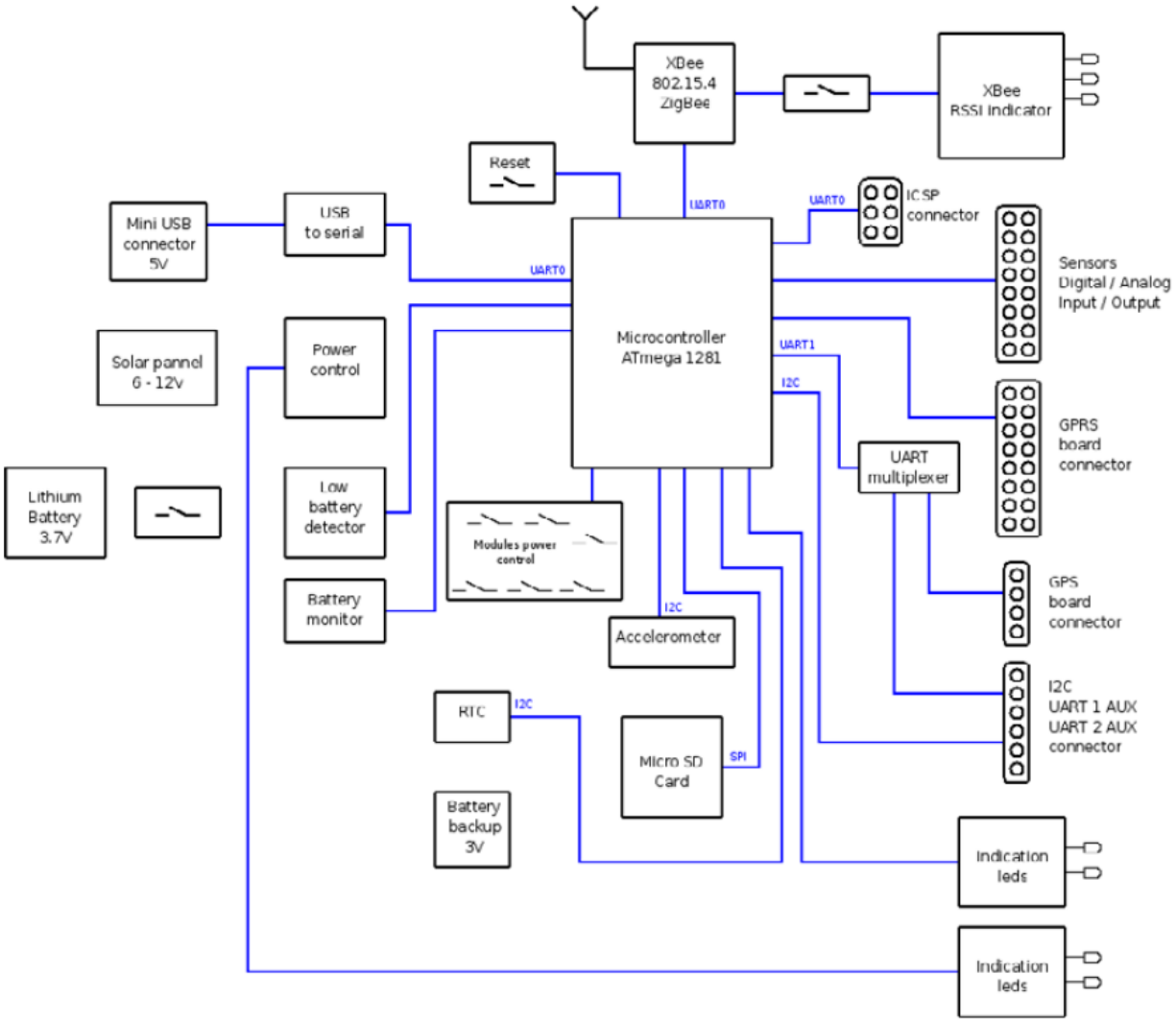


Figure 32: Waspmote block diagram (↻ 3.2.3)

C.1.2 AVR CPU Core Block Diagram

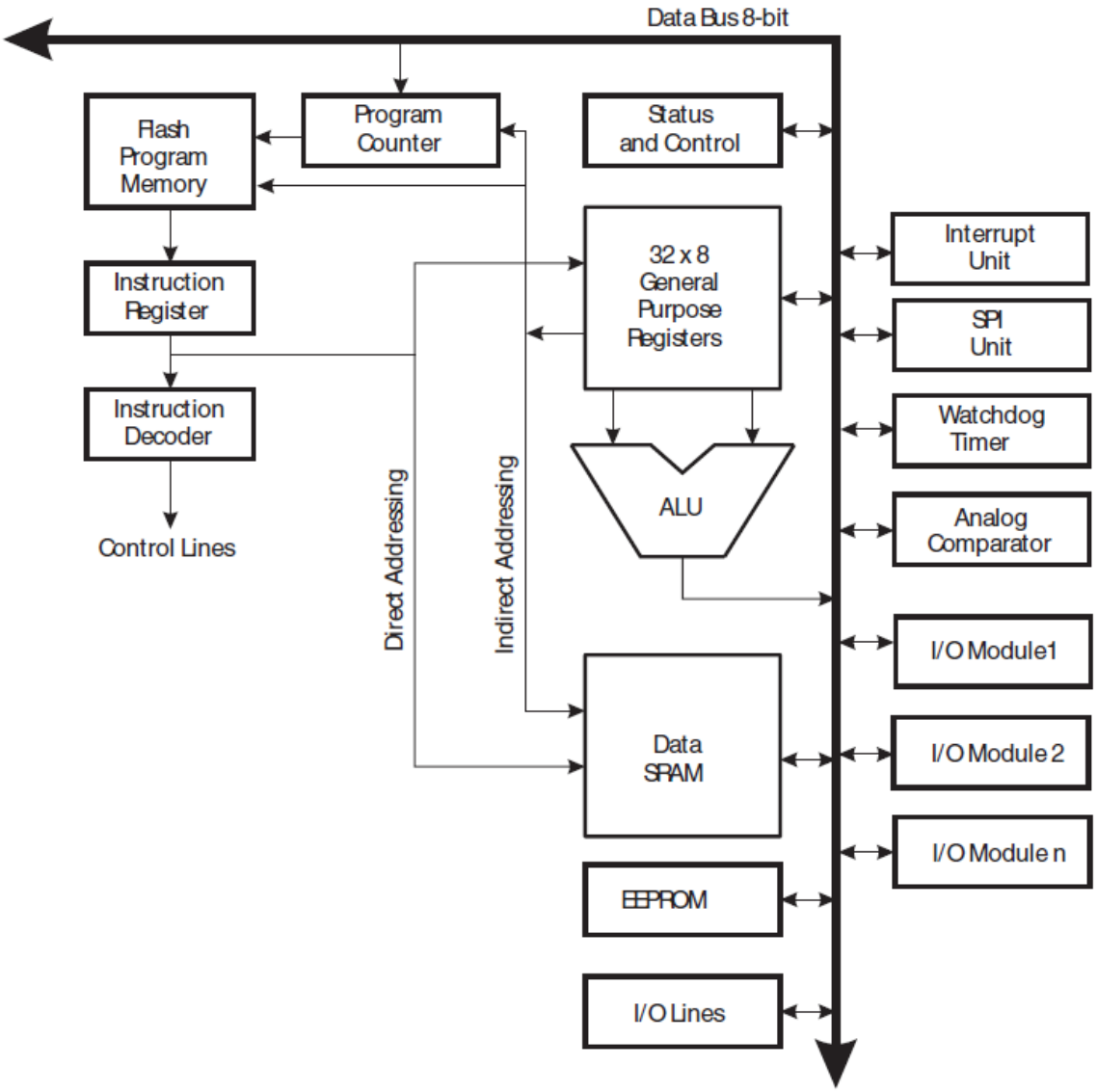


Figure 33: AVR CPU block diagram (↻ 3.2.3)

C.2 Waspmote Variable Sleep Algorithms

To support this algorithm also a copy of the original time will be saved and each time the node wakes up it will look for the smallest next time to sleep. This number will be subtracted from the other sleep times in the array. When a value becomes zero it will be restored with its original value and the cycle continues.
This process is fast and simple. However, the main disadvantage is that the node has to write to EEPROM each time it wakes up. According to the Atmel datasheet, the EEPROM of the ATmega1281 has an endurance of at least 100,000 write / erase cycles. The following equation indicates the problem for an interval of 10 seconds:

$$\frac{100000\text{writes} \cdot 10\text{s}}{60\text{s} \cdot 60\text{min} \cdot 24\text{h}} = 11,57 \text{ days} \tag{1}$$

But the processors has 4Kbytes EEPROM on board so we don't have to write to the same place every time. Since EEPROM is written on a 'per cell' basis this can extend the lifetime. Our sensor mask can contain up to 16 values of 2 bytes. This leads to the next result:

$$\frac{100000\text{writes} \cdot 10\text{s} \cdot 4\text{KB}}{60\text{s} \cdot 60\text{min} \cdot 24\text{h} \cdot 365\text{days} \cdot 32\text{B}} = 3,96 \text{ years} \tag{2}$$

We still must store where the data is stored but this won't cause big problems since we only have to rewrite this cell 125 times:

$$\frac{4\text{KB}}{32\text{B}} = 125 \text{ locations} \tag{3}$$

This supposes however that the EEPROM will not be used for anything else. Below this process is demonstrated.

Sensor[4]	**Sensor[3]**	**Sensor[2]**	**Sensor[1]**	**Sensor[0]**
100	50	35	10	20

Table 5: Individual Sensor Sleep Times in seconds

Cycle	**Sensor[4]**	**Sensor[3]**	**Sensor[2]**	**Sensor[1]**	**Sensor[0]**	**Sleep time**
0	100	50	35	**10**	20	10
1	90	40	25	**10**	**10**	10
2	80	30	15	**10**	20	10
3	70	20	**5**	10	10	5
4	65	15	25	**5**	**5**	5
5	60	**10**	20	**10**	20	10
6	50	50	**10**	**10**	**10**	10

Table 6: Example of sleep algorithm 1

C.3 Practical limitations of Waspmote V1.1

C.3.1 IDE and API

The IDE Libelium offers is very limited, some issues we've experienced are:

- Opening a second or more instance of the IDE sometimes re-opens the previously active files, making it confusing to detect which one you were working in so you end up with two unsaved versions of the same code.
- Once you start compiling (which takes a lot of time) there's no way to stop it, the stop button does not work.
- Uploading immediately after compiling will first re-compile it anyway.
- There is no complete C/C++ support. For example using simple enums is not possible. A workaround is to place the code in additional .h or .cpp files.
- Auto-completion for the Libelium API functionality would be a great addition.

C.3.2 Hardware aspects

Also the Waspmote's (V1.1) hardware slows down the programming process:

- Uploading the code takes a lot of time: 1.5 - 2 minutes.
- The uploading process fails if:
  - The XBee is present
  - The hibernate jumper is not present when the mote is in hibernate
  - The little power switch has been turned off

Often you will want to turn off the power switch temporarily to analyse the content of the serial monitor. Especially in pair programming there is often one requirement you forget and the Waspmote does not check for this on beforehand. It will first compile and do as if it is uploading your code, disappointing you at the end of the process.
When debugging bigger program these actions come even more annoying. Suppose you are testing a program which measures sensors, sends the values and hibernates. Then you must:

1. Remove the sensor board
2. Place the hibernate jumper
3. Remove the XBee
4. Upload
5. Place the XBee
6. Remove the hibernate jumper
7. Re-mount the sensor board

And this is not the end of the list. Removing the hibernate jumper causes the Waspmote to crash one out of two times. Resetting the mote has no effect in this case, just keep inserting and removing the little jumper until it agrees with what you want.

D Waspmote Battery Life Analysis

D.1 Introduction: XBee and Waspmote start-up times

For the XBee node to join an existing network there are two power related possibilities. Either the Waspmote has been turned on already sufficiently long and the XBee had more than enough time to join the network, or either the XBee wasn't joined yet and the program needs to wait on this. From the experiments done at our apartment we came to following conclusions:

1. It takes about 2.5 seconds to join a network after powered on.
2. If the XBee is joined, the program still needs to confirm this. This takes 452 milliseconds.
3. The sending time is constant, about 158 ms, if the XBee had more than 2.5 seconds to join. However in case the XBee must send immediately after it is joined, the sending time is not constant and takes on average 611 milliseconds.
4. The sending time increases if there are more obstructions between the antennas.

Table 7 sums up the results of the distance-relation test.

Distance	Average sending time (ms)
Air	158
1 Floor	268
2 Floors	357
3 Floors	484
4 Floors	558
5 Floors	unreachable

Table 7: Distance consequence on send times

To save power the Waspmote can store the values for a user determined time. Taking samples and save them to EEPROM in case of hibernate mode takes only 6 - 7% of the time to measure and send. Table 9 confirms this.

D.2 High Performance Mode, without ZigBee sleep

For this calculations it is supposed the batteries are in good condition and can be used with optimal conditions. The time values are taken from the third test scenario discussed in section B.3, meaning that the Waspmote uses *Hibernate* or *Deepsleep* and the XBee is completely disconnected from the network. The results are shown in figures 34, 35 and table 36.

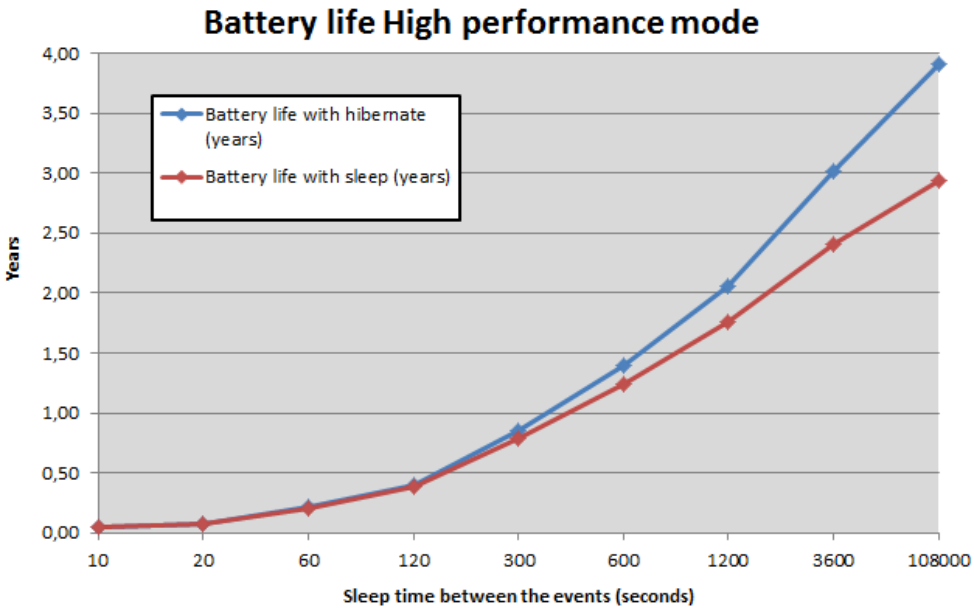


Figure 34: Battery life in High performance mode (↻ 3.3.3)

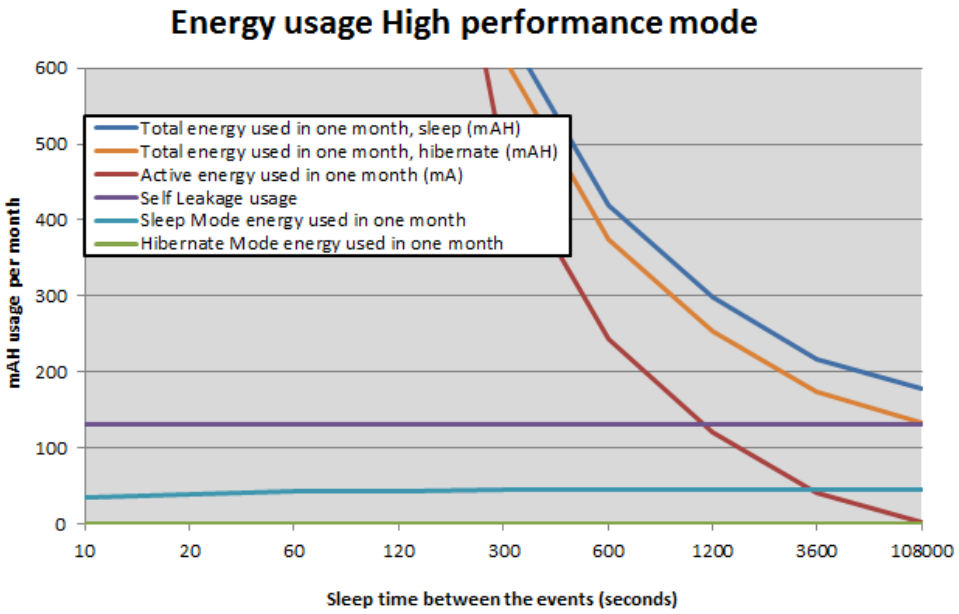


Figure 35: Energy usage in High performance mode (↻ 3.3.3)

The graph in figure 35 breaks down the total energy consumption to five categories. It shows the monthly energy consumption as a function of the time between the events. For small intervals the active energy usage is huge. Only from 20 minutes sleep time, the self-discharge becomes dominant and from 3 hours on the sleep mode current also becomes dominant.
The implementation of this mode will depend on the nodes sleep settings. For *Deep Sleep* the values can simply be stored on the heap, but for *Hibernate* the values must be written to EEPROM.
Because of the size limit of a ZigBee packet we can store maximum 30 values and send them in one packet. However, if the sensor measuring interval is small the user can opt to store more values and send two or more packets after each other. The values for Power Saver in table 8 are of an example scenario that takes 60 measurements and then sends them in two packets to the gateway. It are also those results which are put in function of time in figure 37.
By reducing the sensor measurement accuracy battery life can be extended with modest 3 - 4%, best case scenario.
In case the measuring intervals are small it is recommended to use *Deep Sleep* instead of *Hibernate*, since in hibernate the values are written to EEPROM. Equation 1 shows this can be very destructive for the Waspmote. Depending on how much freedom the user is given, the program can make the decision to switch to *Deep Sleep* on itself, or the installation's administrator can control this.

Battery Characteristics

Battery capacity (nominal)	6600	mAH
Battery efficiency (for the event below)	95	%
Battery capacity (actual)	6600	mAH
Average battery self discharge per month	2	%

Sleep time limits

Minimum sleep time between the even	10	Seconds
Maximum sleep time between the ever	608.800	(7 days)

Application Scenario: no delays between measurements, measuring battery, temperature, humidity and pressure

Step	Action	Duration (ms)	Average Current (uA)	Energy (mAH)	Energy (nAH)
1	Waspmote is in hibernate	Varies	0,7	Varies	Varies
2	Waspmote is in sleep	Varies	62	Varies	Varies
3	Waspmote is ON	3073	9.000	7,68E-03	7682,50
4	Xbee ZigBee PRO is ON	2436	45.560	3,08E-02	30828,93
4	Xbee ZigBee PRO is sending	611	105.000	1,78E-02	17820,83
5	Temperature Sensor	47	6	7,83E-08	0,08
6	Humidity Sensor	48	380	5,07E-06	5,07
7	Pressure Sensor	51	7.000	9,92E-05	99,17
8	CO2 Sensor	0	50.000	0,00E+00	0,00
9	Battery	11	0	0,00E+00	0,00

Total energy consumed in this event (including the repeats, excluding the sleep times)		5,64E-02	56.436,58
Total event duration (excluding the sleeping time (s)	3,07E+00		

Calculations

Energy consumed in the event (described above), excluding the sleep	5,64E-02	mAH (Active Energy per event)
Energy consumed during sleep mode over one month period	44,640	mAH (Per month)
Energy consumed during hibernate mode over one month period	0,504	
Average energy wasted due to battery self leakage in a month	132,00	mAH (Per month)

		SLEEP MODE		HIBERNATE MODE				SLEEP	E
Sleep duration (in seconds) between the events	Active energy used in one month (mA)	Sleep Mode energy used in one month	Total energy used in one month, sleep (mAH)	Hibernate Mode energy used in one month	Total energy used in one month, hibernate (mAH)	Self Leakage usage	Battery efficiency (%)	Battery life with sleep (years)	with hibernate (years)
10	11189,75	34,15	11355,90	0,39	11322,14	132,00	95	0,05	0,05
20	6340,03	38,69	6510,73	0,44	6472,47	132,00	95	0,08	0,08
60	2319,27	42,47	2493,74	0,48	2451,75	132,00	95	0,21	0,21
120	1188,59	43,53	1364,12	0,49	1321,08	132,00	95	0,38	0,40
300	482,67	44,19	658,86	0,50	615,17	132,00	95	0,79	0,85
600	242,56	44,41	418,98	0,50	375,07	132,00	95	1,25	1,39
1200	121,59	44,53	298,12	0,50	254,09	132,00	95	1,75	2,06
3600	40,60	44,60	217,20	0,50	173,10	132,00	95	2,41	3,02
108000	1,35	44,64	177,99	0,50	133,86	132,00	95	2,94	3,90
200000	0,73	44,64	177,37	0,50	133,24	132,00	95	2,95	3,92

Figure 36: Energy usage in High performance mode (3.3.3)

D.3 Power Saver Mode, without ZigBee sleep

For these calculations Power Saver Mode is supposed. The Waspmote will switch on to measure sensors but not to send the data. The values will only be sent once every 60 samples, saving energy. The results are shown in figures 37, 38 and tables 9 and 39. Finally table 8 summarizes the battery duration in years of both performance and power saver mode.

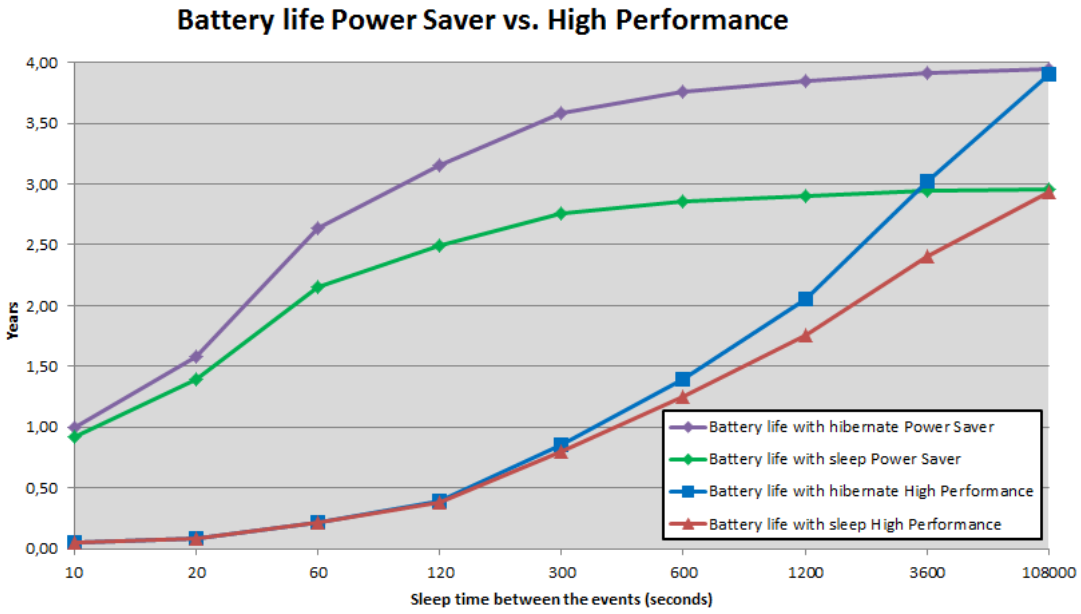


Figure 37: Battery life High Performance vs. Power Saver (↻ 3.3.4)

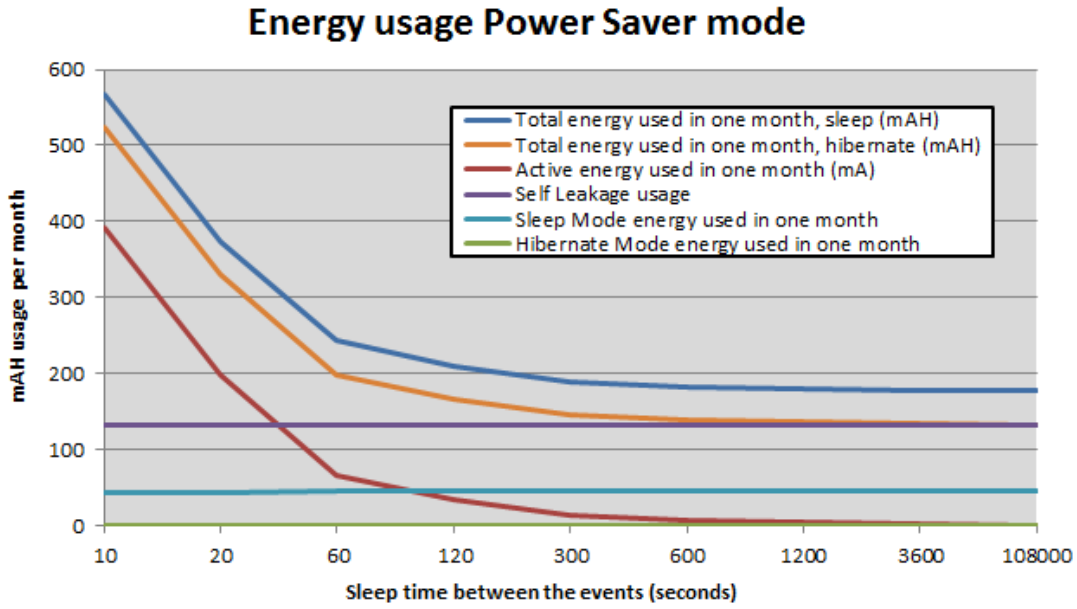


Figure 38: Energy usage in Power Saver mode (↻ 3.3.4)

	Deep Sleep		Hibernate	
Sleep duration	High Performance	Power Saver	High Performance	Power Saver
10s	0,05	0,92	0,05	1,00
1min	0,21	2,15	0,21	2,63
3min	0,79	2,75	0,85	3,59
10min	1,25	2,85	1,39	3,76
20min	1,75	2,90	2,06	3,85
1h	2,41	2,94	3,02	3,91
3h	2,94	2,96	3,90	3,94

Table 8: Battery life in years for High Performance and Power Saver

Nr of samples per sensor	**Average ON time (ms)**
10	210
3	194

Table 9: Time needed to sample and store 4 sensors

Battery Characteristics

Battery capacity (nominal)	6600	mAH
Battery efficiency (for the event below)	50	%
Battery capacity (actual)	6600	mAH
Average battery self discharge per month	2	%

Sleep time limits

Minimum sleep time between the even	10	Seconds
Maximum sleep time between the ever	608.800	(7 days)

Application Scenario: 29 x measuring and storing

Step	Action	Duration (ms)	Average Current (uA)	Energy (mAH)	Energy (nAH)
1	Waspmote is in hibernate	Varies	0,7	Varies	Varies
2	Waspmote is in sleep	Varies	62	Varies	Varies
3	Waspmote is ON (194 - 210 ms)	194	9.000	4,85E-04	485,00
4	Xbee ZigBee PRO is ON	0	45.560	0,00E+00	0,00
4	Xbee ZigBee PRO is sending	0	105.000	0,00E+00	0,00
5	Temperature Sensor	47	6	7,83E-08	0,08
6	Humidity Sensor	48	380	5,07E-06	5,07
7	Pressure Sensor	51	7.000	9,92E-05	99,17
8	CO2 Sensor	0	50.000	0,00E+00	0,00
Total energy consumed in this event (including the repeats, excluding the sleep times)				5,89E-04	589,31
Total event duration (excluding the sleeping time (s)		1,94E-01			

Application Scenario: 1 x measuring and sending

Step	Action	Duration (ms)	Average Current (uA)	Energy (mAH)	Energy (nAH)
1	Waspmote is in hibernate	Varies	0,7	Varies	Varies
2	Waspmote is in sleep	Varies	62	Varies	Varies
3	Waspmote is ON	3073	9.000	7,68E-03	7682,50
4	Xbee ZigBee PRO is ON	2436	45.560	3,08E-02	30828,93
4	Xbee ZigBee PRO is sending	611	105.000	1,78E-02	17820,83
5	Temperature Sensor	47	6	7,83E-08	0,08
6	Humidity Sensor	48	380	5,07E-06	5,07
7	Pressure Sensor	51	7.000	9,92E-05	99,17
8	CO2 Sensor	0	50.000	0,00E+00	0,00
9	Battery	11	0	0,00E+00	0,00
Total energy consumed in this event (including the repeats, excluding the sleep times)				5,64E-02	56.436,6
Total event duration (excluding the sleeping time (s)		3,07E+00			

Calculations

Energy consumed in the event (described above), excluding the slee	5,89E-04	mAH (Active Energy per event)
Energy consumed during sleep mode over one month period	44,640	mAH (Per month)
Energy consumed during hibernate mode over one month period	0,504	
Average energy wasted due to battery self leakage in a month	132,00	mAH (Per month)
Nr of values stored (max 30 per ZB packet)	60,00	

		SLEEP MODE		HIBERNATE MODE				SLEEP	HIBERNATE
Sleep duration (in seconds) between the events	Active energy used in one month (mA)	Sleep Mode energy used in one month	Total energy used in one month, sleep (mAH)	Hibernate Mode energy used in one month	Total energy used in one month, hibernate (mAH)	Self Leakage usage	Battery efficiency (%)	life with sleep (years)	Battery life with hibernate (years)
10	392,41	43,79	568,20	0,49	524,90	132,00	95	0,92	1,00
20	197,54	44,21	373,76	0,50	330,04	132,00	95	1,40	1,58
60	66,01	44,50	242,51	0,50	198,51	132,00	95	2,15	2,63
120	33,03	44,57	209,59	0,50	165,53	132,00	95	2,49	3,16
300	13,22	44,61	189,83	0,50	145,72	132,00	95	2,75	3,59
600	6,60	44,63	183,23	0,50	139,11	132,00	95	2,85	3,76
1200	3,30	44,63	179,94	0,50	135,81	132,00	95	2,90	3,85
3600	1,10	44,64	177,74	0,50	133,61	132,00	95	2,94	3,91
10800	0,37	44,64	177,01	0,50	132,87	132,00	95	2,95	3,93

Figure 39: Energy usage in Power Saver mode (⟲ 3.3.4)

E Waspmote Flow Chart

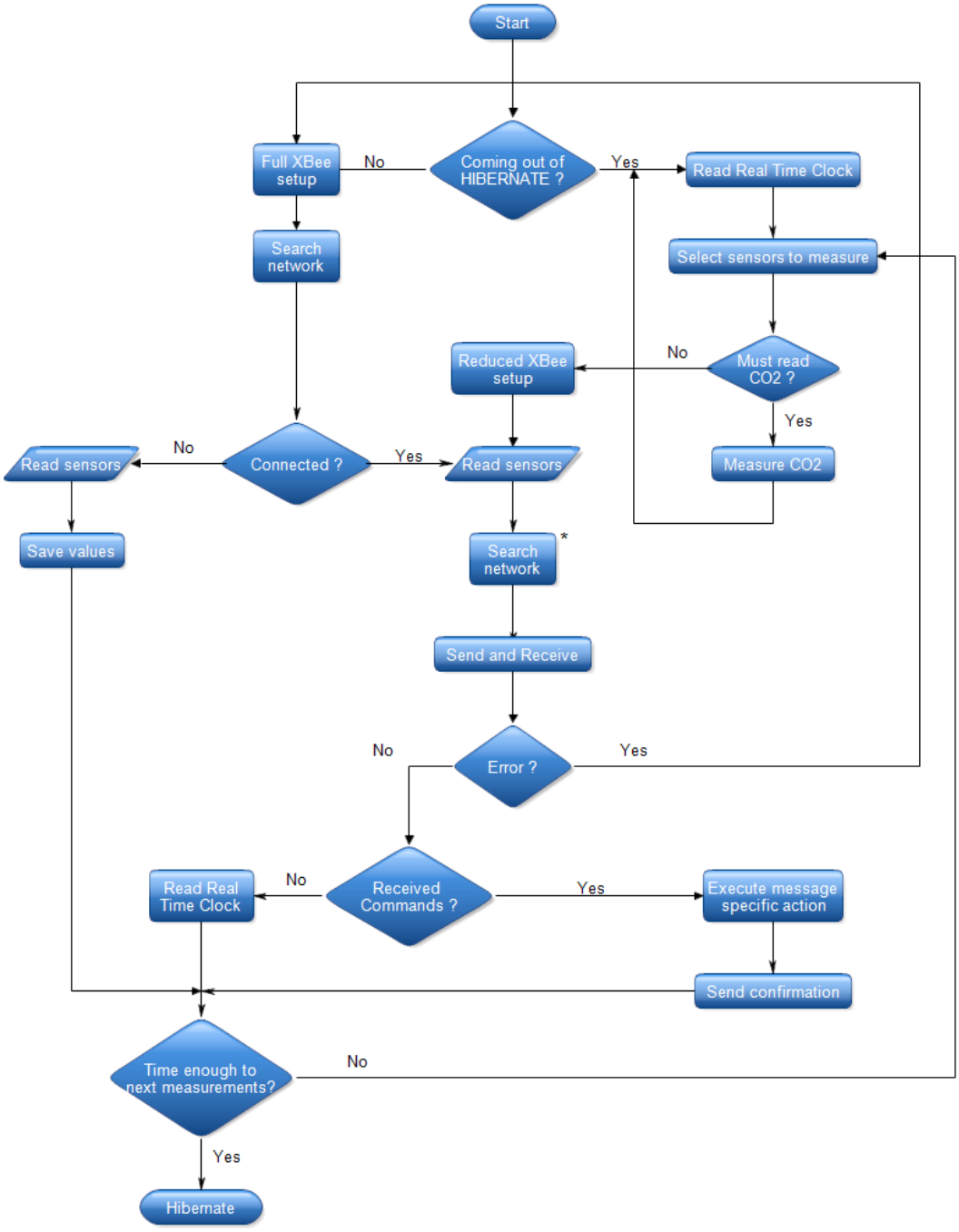


Figure 40: Flow chart of the Waspmote program for end devices (↻ 4.2)

F ZigBee Packet Structure

F.1 API Frames

API Frame Names and Values

API Frame Names	API ID
AT Command	0x08
AT Command - Queue Parameter Value	0x09
ZigBee Transmit Request	0x10
Explicit Addressing ZigBee Command Frame	0x11
Remote Command Request	0x17
Create Source Route	0x21
AT Command Response	0x88
Modem Status	0x8A
ZigBee Transmit Status	0x8B
ZigBee Receive Packet (AO=0)	0x90
ZigBee Explicit Rx Indicator (AO=1)	0x91
ZigBee IO Data Sample Rx Indicator	0x92
XBee Sensor Read Indicator (AO=0)	0x94
Node Identification Indicator (AO=0)	0x95
Remote Command Response	0x97
Over-the-Air Firmware Update Status	0xA0
Route Record Indicator	0xA1
Many-to-One Route Request Indicator	0xA3

Figure 41: API Frame Names and Values (↻ 3.2.2)

F.2 ZigBee Transmit Request

	Frame Fields		Offset	Example	Description
API Packet	Start Delimiter		0	0x7E	
	Length		MSB 1	0x00	Number of bytes between the length and the checksum
			LSB 2	0x16	
	Frame-specific Data	Frame Type	3	0x10	
		Frame ID	4	0x01	Identifies the UART data frame for the host to correlate with a subsequent ACK (acknowledgement). If set to 0, no response is sent.
		64-bit Destination Address	MSB 5	0x00	Set to the 64-bit address of the destination device. The following addresses are also supported: 0x0000000000000000 - Reserved 64-bit address for the coordinator 0x000000000000FFFF - Broadcast address
			6	0x13	
			7	0xA2	
			8	0x00	
			9	0x40	
			10	0x0A	
			11	0x01	
			LSB 12	0x27	
		16-bit Destination Network Address	MSB 13	0xFF	Set to the 16-bit address of the destination device, if known. Set to 0xFFFE if the address is unknown, or if sending a broadcast.
			LSB 14	0xFE	
		Broadcast Radius	15	0x00	Sets maximum number of hops a broadcast transmission can occur. If set to 0, the broadcast radius will be set to the maximum hops value.
		Options	16	0x00	Bitfield of supported transmission options. Supported values include the following: 0x01 - Disable retries and route repair 0x20 - Enable APS encryption (if EE=1) 0x40 - Use the extended transmission timeout Enabling APS encryption presumes the source and destination have been authenticated. I also decreases the maximum number of RF payload bytes by 4 (below the value reported by NP). The extended transmission timeout is needed when addressing sleeping end devices.It also increases the retry interval between retries to compensate for end device polling.See Chapter 4, Transmission Timeouts, Extended Timeout for a description. Unused bits must be set to 0.
		RF Data	17	0x54	Data that is sent to the destination device
			18	0x78	
			19	0x44	
			20	0x61	
			21	0x74	
			22	0x61	
			23	0x30	
			24	0x41	
	Checksum		25	0x13	0xFF - the 8 bit sum of bytes from offset 3 to this byte.

Figure 42: ZigBee Transmit Request Frame Structure (⟲ 3.2.2)

F.3 ZigBee Receive Packet

	Frame Fields		Offset	Example	Description
API Packet	Start Delimiter		0	0x7E	
	Length		MSB 1	0x00	Number of bytes between the length and the checksum
			LSB 2	0x11	
	Frame-specific Data	Frame Type	3	0x90	
		64-bit Source Address	MSB 4	0x00	
			5	0x13	64-bit address of sender. Set to 0xFFFFFFFFFFFFFFFF (unknown 64-bit address) if the sender's 64-bit address is unknown.
			6	0xA2	
			7	0x00	
			8	0x40	
			9	0x52	
			10	0x2B	
			LSB 11	0xAA	
		16-bit Source Network Address	MSB 12	0x7D	16-bit address of sender
			LSB 13	0x84	
		Receive Options	14	0x01	0x01 - Packet Acknowledged 0x02 - Packet was a broadcast packet 0x20 - Packet encrypted with APS encryption 0x40 - Packet was sent from an end device (if known) Note: Option values can be combined. For example, a 0x40 and a 0x01 will show as a 0x41. Other possible values 0x21, 0x22, 0x41, 0x42, 0x60, 0x61, 0x62.
		Received Data	15	0x52	Received RF data
			16	0x78	
			17	0x44	
			18	0x61	
			19	0x74	
			20	0x61	
	Checksum		21	0x0D	0xFF - the 8 bit sum of bytes from offset 3 to this byte.

Example: Suppose a device with a 64-bit address of 0x0013A200 40522BAA, and 16-bit address 0x7D84 sends a unicast data transmission to a remote device with payload "RxData". If AO=0 on the receiving device, it would send the above example frame out its UART.

Figure 43: ZigBee Receive Packet Frame Structure (⟲ 3.2.2)

F.4 Waspmote Application Header in ZigBee API Frame Structure

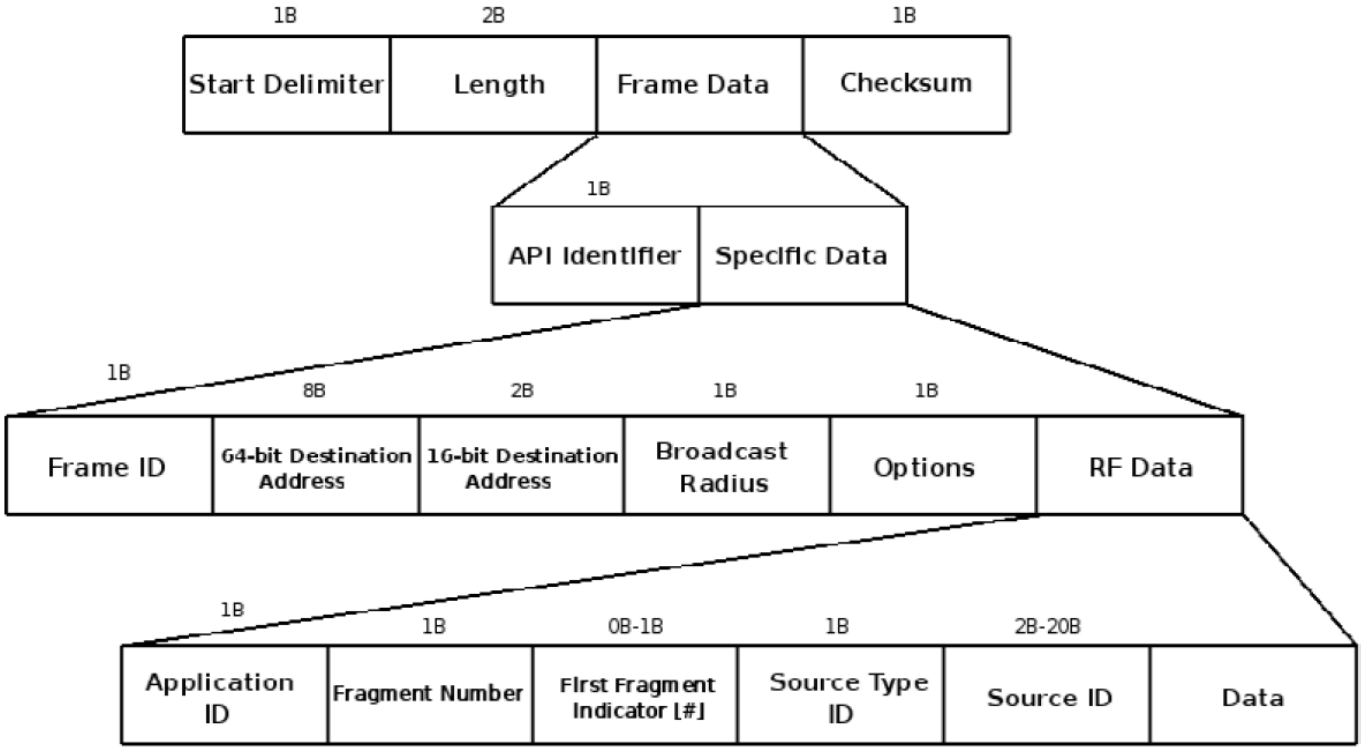
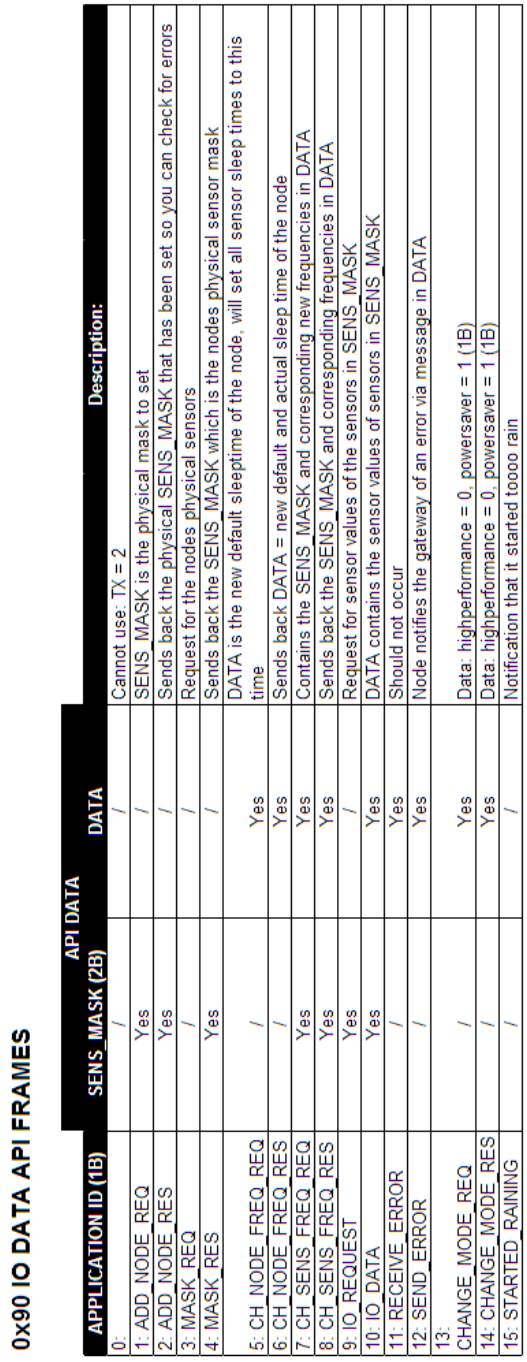


Figure 44: Waspmote Application Header in ZigBee API Frame Structure (⟲ 4.2.3 ⟲ 3.2.2)

F.5 IO API Data Frames for Waspmote and Gateway



0x90 IO DATA API FRAMES

APPLICATION ID (1B)	API DATA		Description:
	SENS_MASK (2B)	DATA	
0:	/	/	Cannot use: TX = 2
1: ADD_NODE_REQ	Yes	/	SENS_MASK is the physical mask to set
2: ADD_NODE_RES	Yes	/	Sends back the physical SENS_MASK that has been set so you can check for errors
3: MASK_REQ	/	/	Request for the nodes physical sensors
4: MASK_RES	Yes	/	Sends back the SENS_MASK which is the nodes physical sensor mask
5: CH_NODE_FREQ_REQ	/	Yes	DATA is the new default sleeptime of the node, will set all sensor sleep times to this time
6: CH_NODE_FREQ_RES	/	Yes	Sends back DATA = new default and actual sleep time of the node
7: CH_SENS_FREQ_REQ	Yes	Yes	Contains the SENS_MASK and corresponding new frequencies in DATA
8: CH_SENS_FREQ_RES	Yes	Yes	Sends back the SENS_MASK and corresponding frequencies in DATA
9: IO_REQUEST	Yes	/	Request for sensor values of the sensors in SENS_MASK
10: IO_DATA	Yes	Yes	DATA contains the sensor values of sensors in SENS_MASK
11: RECEIVE_ERROR	/	Yes	Should not occur
12: SEND_ERROR	/	Yes	Node notifies the gateway of an error via message in DATA
13: CHANGE_MODE_REQ	/	Yes	Data: highperformance = 0, powersaver = 1 (1B)
14: CHANGE_MODE_RES	/	Yes	Data: highperformance = 0, powersaver = 1 (1B)
15: STARTED_RAINING	/	/	Notification that it started toooo rain

extra:
SET_TEXT_NODE_ID
GET_TEXT_NODE_ID
SET_NEW_GATEWAY
GET_GATEWAY
SET_DEVICE_ROLE
GET_DEVICE_ROLE

Figure 45: IO API Data Frames for Waspmote and Gateway (↻ 4.2.3)

F.6 Web service packets

An XML example of the different HTTP POST request that can be made to the web service. The full url becomes: `(webserviceIP)/url`
Url is replace by the urls found below for each request. The web service IP is currently not fixed and has to be determined upon installation of the gateway. It is also possible that a domain name will be added. This will be something like WSN.groupt.be. The XML is sent as POST data in a HTTP request. All Each ID relates to the corresponding ID in Ipsum. The client will create installations, sensor groups and sensors in Ipsum and then send all these IDs to the gateway so the gateway knows to which Ipsum sensor to upload data or from which sensor change frequencies.

Type	Add node request
Description	This request will add a new node to the gateway. Installation ID, sensor group ID and ZigBee address are required. The zigbee address can be found on the bottom of the xbee radio.
URL	/addNode/authenticationcode
POST Data	<addNode> <installationID>32</installationID> <sensorGroupID>546</installationGroupID> <zigbeeAddress>0013A2004069737C</zigbeeAddress> </addNode>

Figure 46: An Add node request

Type	Add sensor request
Description	This request will add one or more sensors of a certain type, to the gateway. The sensor group ID of the node is required. Possible sensor types are: zigbeeTemp, zigbeeHum, zigbeePres, zigbeeBat, zigbeeCO2, zigbeeAnemo, zigbeeVane, zigbeePluvio.
URL	/addSensor/authenticationcode
POST Data	<addSensor> <sensorGroupID>546</sensorGroupID> <sensor> <sensorID>5654</sensorID> <sensorType>Temperature, Humidity, Pressure, CO2, Battery,.. </sensorType> </sensor> <sensor> <sensorID>5654</sensorID> <sensorType>Temperature, Humidity, Pressure, CO2, Battery,.. </sensorType> </sensor> ... </addSensor>

Figure 47: An Add sensor request

Type	Change frequency request
Description	This request will change the sampling interval of one or more sensor.. The sensor group ID of the node is required as well as all ID's of all frequencies to be changed and the new frequency. These frequencies should always be a multiple of 10.
URL	/changeFrequency/authenticationcode
POST Data	<changeFrequency> <sensorGroupID>546</sensorGroupID> <sensor> <sensorID>5654</sensorID> <frequency>10</frequency> // must be a multiple of 10 and is expressed in seconds </sensor> <sensor> <sensorID>5656</sensorID> <frequency>10</frequency> </sensor> ... </changeFrequency>

Figure 48: A change frequency request

Type	Request data
Description	This request ask for data from certain sensors of a certain node. The sensor group ID of the node where these sensors are located, is required. Also each sensor ID should be provided
URL	/requestData/authenticationcode
POST Data	<requestData> <sensorGroupID>546</sensorGroupID> <sensorID>5654</sensorID> <sensorID>5656</sensorID> ... <requestData>

Figure 49: A data request

G Practical remarks

G.1 Application ID / Packet ID

The following Application IDs are reserved by Libelium and may not be used to identify packets:

- 0x00
- 0xF8
- 0xF9
- 0xFA
- 0xFB
- 0xFC
- 0xFD
- 0xFE
- 0xFF

G.2 Read the Programming Style Guide available at:

`http://www.libelium.com/development-v11/.product=waspmote&zone=tutorial0004`.

G.3 Follow the 19 instructions from the Waspmote checklist available at:

`http://www.libelium.com/forum/viewtopic.php?f=14&t=7700`.

G.4 Ready?

1. Enable the `#define FINAL` pre-processor instruction in order to remove `USB.begin()` instructions from the program and to save battery power.
2. Remove the 'Programming enabling jumper' in order to reduce the power consumption of Waspmote to the minimum.
3. Remove the 'RSSI indicator enabling jumper' to avoid extra consumption by the three RSSI LEDs.