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

Co-supervisor(s): <Names>
<Department,. . . >, <Company>, <Company Address>

ABSTRACT

The abstract should be viewed as a miniature version of the paper. Since potential readers should be able to make their decision on the personal relevance based on the Abstract, the Abstract should clearly tell the reader what information he can expect to find in the paper. The Abstract should (1) state the principal objectives and scope of the investigation, (2) describe the methods employed, (3) summarize the results, and (4) state the principal conclusions. Most of the Abstract will be written in the past tense, because it refers to work done. The Abstract should never give any information or conclusion that is not stated in the paper. The authors should always keep in mind, that the Abstract is the most frequently read part of a paper. It should contain at most 200 words. Today, the Abstract is generally printed in one paragraph. Define all symbols used in the Abstract. Do not cite references in the Abstract. Do not include or refer to tables and figures.

Keywords

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

1 Introduction to Wireless Sensor Networks

1.1 Introduction

The ENIAC, the first electronic computer designed by the 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 less the size of the ENIAC, still Turing-complete, and still military roots. The origins of the research to Wireless Sensor Networks (WSNs) can be traced back to DARPA[1], 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 network such as ad hoc networking, dynamic querying and tasking, reprogramming and multitasking. At the same time IEEE 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 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 programmes (Program 973)[Zennaro, 2008]. The 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]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 GUI [Wikipedia, 2013b].

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[2] in 1999 and answers the question introducing this chapter. It refers to uniquely identifiable objects (things) and their virtual representations in an Internet-like structure. It is a vision of a network of Internet-enabled objects, together 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 qualitative and quantitative shift in several sectors: logistics, domotics, healthcare, entertainment, and so on.

WSNs can 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[3] 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 to bridge the gap between the virtual electronic world and our 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 basic components:

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

[2]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]

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

The main controller options are Microcontrollers, DSPs, FPGAs or ASICs. Microcontrollers are the best option for a WSN node. They are general purpose and optimized for embedded applications, so they use little power consumption. 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 but have no flexibility. The Libelium Waspmotes used for the test-bed developed for Group T have an 8-bit Atmel controller (see 4.3.2).

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 3.4 the power consumption of the WSN we developed will be analyzed thoroughly. Other challenges often fall back to this limited amount of available energy, security for example.
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 over more than 100 nodes, this leads to scalability issues 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

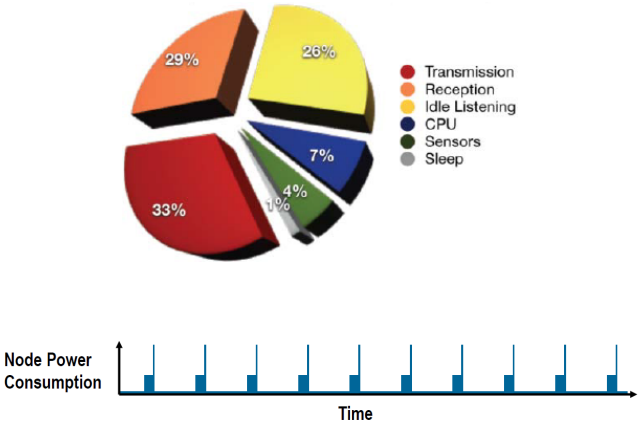


Figure 1. Typical power consumption of a node

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 3.6.2.3), effective dynamic power management (section 3.4.1) and optimize duty cycles are key to conserve power.

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 efficiency increases, 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 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

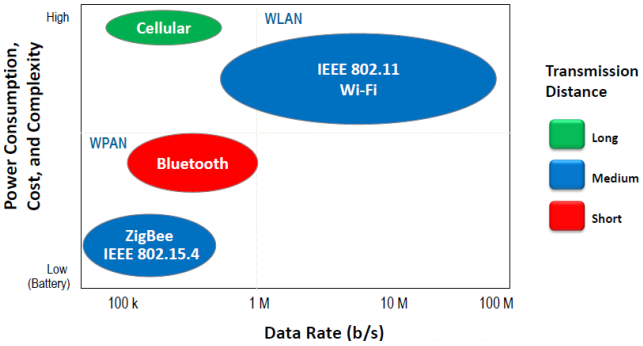


Figure 2. Comparing the ZigBee standard with Cellular, Bluetooth and WiFi

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 make digital communication more flexible, is the use of networking layers. Figure 14 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 layer on top of IEEE 802.15.4.

	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 Cellular, Bluetooth and WiFi

2 Design of the WSN Test-Bed

2.1 Presentation of the thesis goal

The purpose of this thesis is creating a wireless sensor network in the GroupT Vesalius building. It is not our goal to cover the entire building with sensors but rather establish a basic wireless sensor network which reaches out throughout the entire building. So that later on sensor nodes can be added and they will be easy to integrate with the existing network. The data coming from these sensors should also be stored safely and a web interface should allow easy access to all data. For data storage Ipsum is used, this is a cloud storage system developed by Ruben Taqc. Ipsum is already used to log data coming from other sensors and is still being updated and maintained. The web interface is a master thesis of Matthias Verhelst. For normal users the web interface should fetch data from Ipsum. But for admin or privileged users it should also possible to communicate directly with the wireless sensor network. So the gateway has to have a web service running that can be contacted by clients via a web browser. Since wireless sensor nodes are battery power also the power consumption of these nodes has to be examined thoroughly to try and function for as long as possible without having to recharge the batteries. For a sensor network it is crucial that data storage happens correctly at all times. No data can be lost. So if Ipsum is unreachable, the gateway has to temporally store incoming data.

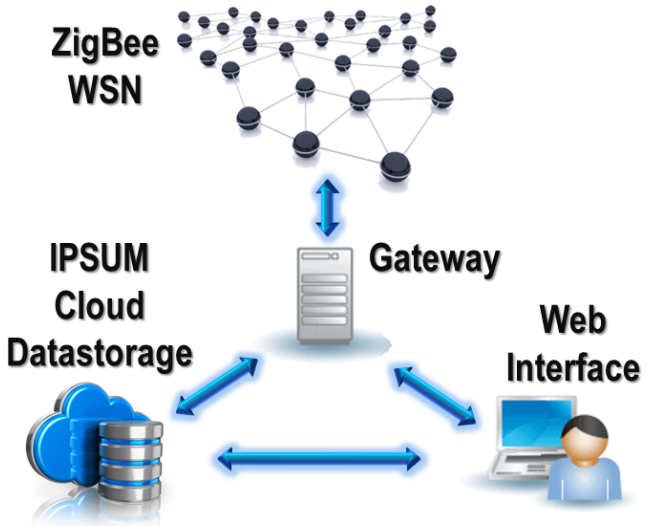


Figure 3. Presentation of the subject

2.2 API Frame Specifications

To communicate with the ZigBee radio there are 2 modes, AT (transparent mode) and API (application programming interface). AT mode means that what you send to the zigbee radio using RS232, the ZigBee radio will send to its default destination. Unless you send +++, wait for the ZigBee to reply with OK, and then send an AT command. AT commands are used to change the configuration of the ZigBee radio. For instance the AT command OP requests the operating PAN ID.
AT mode is fairly limited and only good for point to point communication since you cant 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 zigbee 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 4. It starts with 0x7E as start delimiter and is followed by the length of the data excluding checksum. Then a API-specific structure follow which depends on the type of packet.

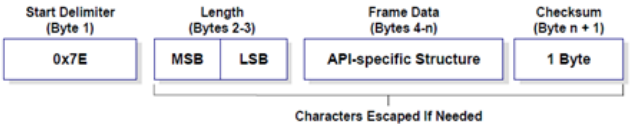


Figure 4. UART Data Frame Structure

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

Table 2. API Frame Names and Values

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 configurations of the ZigBee radio. This can of course also be done in API mode. For details about all the packets, please consult the datasheet. The only packets types used in this project are: 'ZigBee transmit request, 0x10' and 'Zigbee receive packet, 0x90'.
A ZigBee transmit request is shown in figure 33. This packet is used to send data from this ZigBee radio to a remote one.

All that has to be known is the remote ZigBee address. 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 the gateway. Libelium has its own specific format for the RF Data as will be explained in section **??**. 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 and coordinator to be the same is that the coordinator receives a lot of traffic due to its role as coordinator and the same goes for the gateway. So these 2 devices should be in the center of the network for efficiency reasons. Assigning one device for these 2 roles and trying to position this device as central as possible will ask for the least amount of routing overhead.
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 34. Again the received data has an additional format as specified by Libelium.

3 A WSN with Waspmotes: Theoretical aspects

3.1 Introduction

Waspmote is more than just another piece of hardware. In fact it is an open source platform for wireless sensors, specially focusing on low consumption and autonomy. Waspmotes promise to offer a variable lifetime between 1 and 5 years, depending 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 sensor device than wireless sensor device. The 3.3V - 5V regulator could not be turned off, as a result there is 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 source wireless sensor device was definitely set.
In 2009 the Waspmote was born, meeting all the above requirements: low consumption and meeting three radio certification requirements (CE for Europe, FCC for the US and IC for Canada). 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 have been using 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 improved API, 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 the XBee radio must no longer be removed to do so. A new interrupt enables the XBee to wake the Waspmote PRO whun XBee cyclic sleep modes are used [Waspmote (v1.1) vs Waspmote PRO (v1.2), 2013]. To do this on the old 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, ACC, etc.) will be masked by the XBee 13 pin output and thus will be lost. The new version also no longer has jumpers and there is no need of a coin battery. Regarding the API, Libelium claims it is much more robust and easier to use than the previous one

Figure 5. Waspmote going from ON to Deep Sleep

and they also improved their programming guides. One big feature 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 this chapter's introduction, Waspmote is based on a modular architecture, doing so optimizing costs and able 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
- Storage

Figure 6 indicates the Waspmote main components.

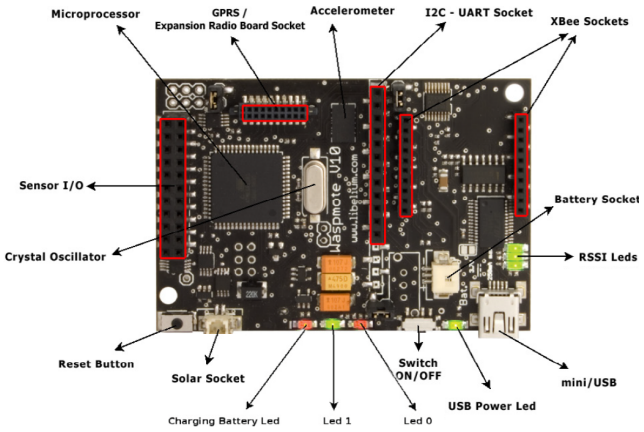


Figure 6. Main Waspmote components

3.2.2 Microcontroller and memory

Because of the modular design of the Waspmote the block diagram (see figure 23) is very simple. Just like on any other PCB, the CPU is the heart of the module. Waspmote integrates an 8-bit ATmega 1281 microcontroller with 128KB programmable flash, 8KB SRAM runtime memory and 4KB EEPROM memory. Since SRAM is 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 was developed in 1996 by Atmel. It is a modified Harvard architecture 8-bit RISC 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).

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. Systems can have much more instruction memory than data memory, so instruction address zero might indicate a 24-bit value for a 16-bit instruction, while data address zero might identify an 8-bit value for 8-bit wide data. 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 is modified Harvard but the separate address space nature of a Harvard machine is preserved. In contrast to systems that add CPU cache, in which data and instructions are unified, providing the von Neumann model. The adaptation is much more subtle. The ATmega has an Extended Load Program Instruction which can allocate constant tables within the program memory address space (see figure 24). So the contents of the instruction memory can be accessed as data, saving scarce runtime memory. This creates however certain difficulties in programming, since the C Language was not designed for Harvard architectures (see section 4.3.2).

3.2.3 Timers

The Waspmote's system clock is an 8MHz quartz oscillator. This means that every 125ns a machine language instruction is executed by the microcontroller. Keep in mind that one C++ instruction consists of several instructions in machine language. To generate interrupts the Waspmote has an internal watchdog and a Real Time Clock (RTC).

3.2.3.1 Watchdog The Watchdog is integrated on the Atmega 1281 and counts the clock cycles generated by a 128KHz oscillator. When the WDT counter reaches a set value it generates an interruption signal. The WDT is used to awake the microcontroller from *Sleep* mode, because of its high precision. Thus, *Sleep* mode allows small intervals, going from 16 milliseconds to a maximum of 8 seconds.

3.2.3.2 Real Time Clock To store an absolute time base the RTC can be used. Alarms programmed in the RTC specify days, hours, minutes and seconds. For the RTC the waspmote uses a Maxim DS3231SN 32.768Hz oscillator. Because this clock has an internal compensation mechanism for variations caused by temperature changes this is one of the most accurate clocks on the market. This clock is used to wake the Waspmote from the higher energy saving modes *Deep Sleep* and *Hibernate*, with intervals from 8 seconds to even days. 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 probably it is recommended to measure the button battery's voltage and possibly must be replaced.

3.3 Programming

To develop on the Waspmote, Libelium offers an API and IDE. For more information on the API, please see section 4.4.2. Waspmote uses the same IDE (compiler and core libraries) as Arduino, so as long as things like pin layout and I/O schemes are adjusted, code should be compatible in both platforms. So Waspmote and Arduino are pretty much the same, however don't forget that Waspmote has Radio Certifications and Arduino doesn't.
When the Waspmote is started, the microcontroller will execute the bootloader and start loading the compiled program from FLASH into the SRAM working memory.
The code is divided into two basic parts: **setup** and **loop**, each with sequential behaviour. When the Waspmote is switched on or reset, the code starts at the setup function and then enters the loop function. Because the second part forms an infinite loop a common technique to save energy is to block the program until some interruption is detected.
Since in *Hibernate* mode the Waspmote is completely disconnected from the main battery also the program is interrupted. This means the SRAM has lost all variable values and at wake up the code restarts at the **setup** function. To store values during hibernate cycles it is necessary to write them to EEPROM or to the SD card.

3.4 Power considerations

3.4.1 Waspmote power modes

from intro:

Waspmotes have a very low power consumption in *Hibernate* mode but for shorter sleep times and the conservation of program variables *Deep Sleep* mode is more advisable. Unfortunately it is not easy to combine and switch between the modes depending on the next time to sleep.
Although OTA programming 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 1.1nAH per byte [Fischione, 2011].

3.4.1.1 PHY Layer Responsibilities

- Enabling and disabling the radio transceiver ... ...
- Transmit and receive data ... ...
- Select CH ... ...
- Estimating signal energy ... ...
- Providing RSSI, LQI ... ...

3.4.1.2 MAC Layer Responsibilities

- Generating beacons ... ...
- CSMA-CA, BPSK vs. O-QPSK, vs. ASK ... ...
- Providing a reliable link ... ...
- PAN association and disassociation services

3.4.1.3 Network Layer Responsibilities

- Setting up a network ... ...
- Allow joining and leaving a network ... ...
- Configuring new devices ... ...
- Discover and maintain routes ... ...

3.4.1.4 Application Layer Responsibilities

- Application support ... ...
- Address management and mapping ... ...
- Define the role of the device ... ...
- Security related tasks ... ...

3.5 Networking concepts

3.5.1 Device Types

3.5.2 Device Roles

In table **??** from [Dynamic C: An Introduction to ZigBee User Manual, 2008] an overview of the device responsibilities on network layer is given.

3.5.2.1 Coordinator Operation

3.5.2.2 Router Operation

3.5.2.3 End Device Operation sleep...

3.6 Parent - Child relationship

The libelium Waspmote has 4 operational modes: ON, Sleep, Deep Sleep and Hibernate. They differ from which type of interruptions they can be woken up and duration interval. For our application we want sleep intervals of 30 seconds and more, so only Deep Sleep and Hibernate mode are of interest. Table 3 summarizes the Waspmotes operational modes.

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 3. Operational modes of Libelium Waspmote V1.1

3.6.0.4 Deep Sleep In deep sleep mode the main program is paused and the CPU passes to a latent state. Triggers are as well synchronous interruptions (RTC) as asynchronous interruptions. Examples of asynchronous interruptions are low battery level or a sensor that reaches a certain trigger value.

In figure **??** the process from going to operational mode ON to Deep Sleep is shown. The main advantage of this mode is that the program is only paused, so the program stack and thus all variable values keep their values. When the Waspmote is turned back on it simply executes the next instruction.

3.6.0.5 Hibernate Hibernate mode consumes roughly 100 times less energy than Deep Sleep. This is made possible by disconnecting all the Waspmote's modules, including the microcontroller. The RTC gets his power through the auxiliary battery. So if hibernate mode stops working it is probably necessary to replace the Waspmote's button battery. Figure 7 demonstrates the process from ON to hibernate.
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. To be able to wake up from hibernate mode the hibernate jumper must be removed correctly. See section .... for remarks on this issue.
Because not all Libelium's API functions regarding hibernate in combination with the different alarm modes work, it is advised to use the functions provided in **WaspXBeeZBNode.h**.

3.6.1 Sampling sensors

To measure the sensors, originally we took 10 samples with 100 milliseconds recommended delay between the measurements and calculated the average. Since we want to make the energy consumption as low as possible we now do the iterations without delay. Appendix A.4 contains 60 test samples per sensor and indicates that there is no significant difference on the average by removing this delay. Except for CO_2 measurements, removing the delay saves about 1000 milliseconds per measured sensor.
Because the first sample often shows a slight deviation, the program takes 11 samples but bases the average on the last 10 values.

3.6.2 Battery life estimation

In order to be able to give recommended sensor measuring intervals this section will analyse the estimated battery life of the Waspmotes. Table 8 enumerates the most common components typical consumption. The batteries included

Action	Average Current
XBee, sending,	105mA
CO_2	50mA
XBee, ON	45mA
Waspmote, ON	9mA
Pressure	7mA
Humidity	380μA
Waspmote, sleep	62μA
Temperature	6μA
Waspmote, hibernate	0,007μA

Table 4. Operational modes of Libelium Waspmote V1.1

with our Waspmotes are rechargeable Lithium-ion batteries with a capacity of 6600mAH. The Waspmote that will be deployed on the roof of Group T, campus Vesalius will also have a 12V solar panel with a charging current up to 280 mA to extend its battery life. The other batteries can by charged manually or by USB (5V, 100mA). Lithium-ion have a self-discharge rate of typcially 1 to 2 percent per month [Buchmann, 2013] and since the used batteries are new we expect a high battery efficiency.

3.6.2.1 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.

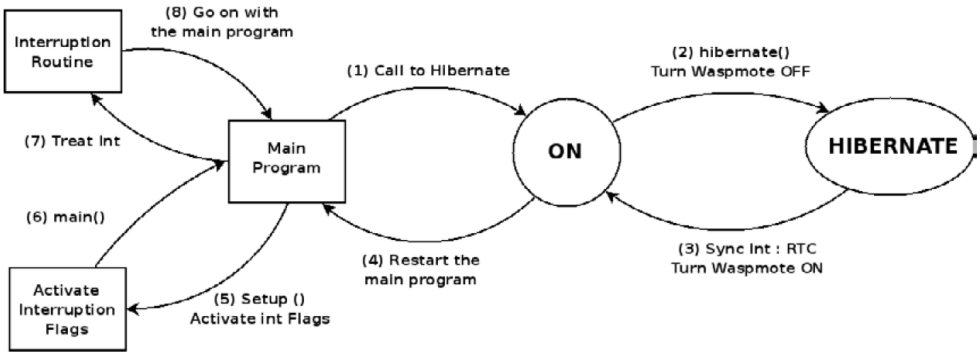


Figure 7. Waspmote going from ON to Hibernate

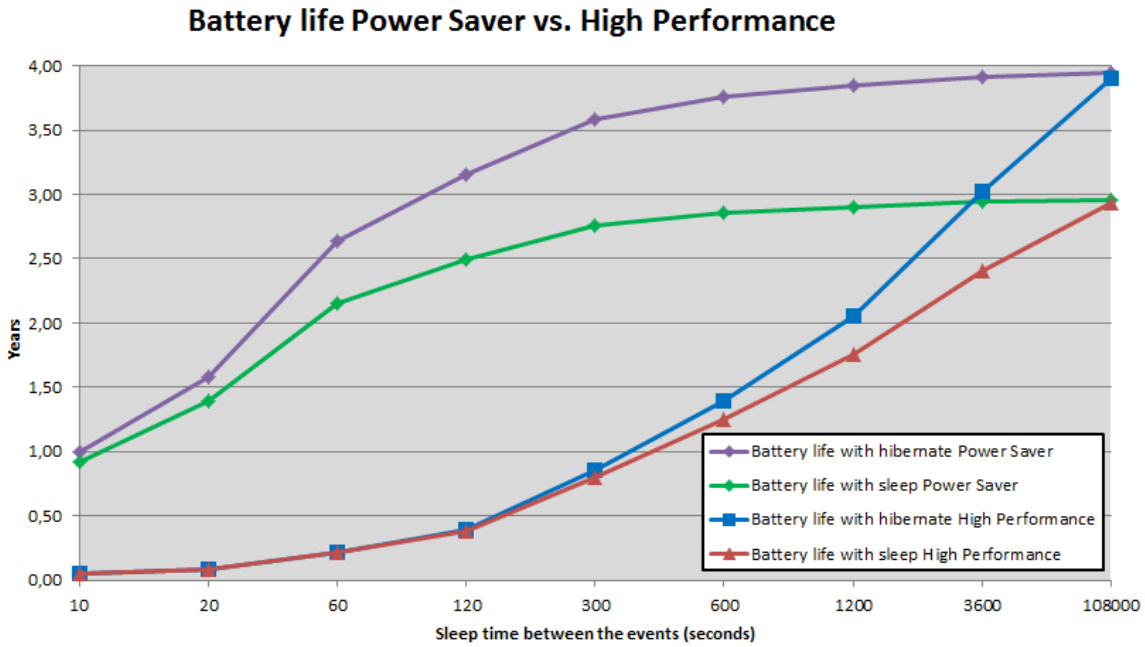


Figure 8. Battery life High Performance vs. Power Saver

4. The sending time increases if there are more obstructions between the antennas.

With these characteristics we came to results discussed in the next sections. For the validation of these conclusions please see appendix A. Table 5 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 5. 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.

3.6.2.2 Battery life with standard program optimizations The application scenario for this battery test is as follow: 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 sent to the gateway. By adapting the sleep time between the event we came to the rather disappointing results shown in figure 25.

The graph in figure 26 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 starting at 20 minutes sleep time the self-discharge becomes dominant and from 3 hours on the sleep mode current also becomes dominant.

Since the Waspmotes use this much energy when applied this way we will call this the High Performance mode from now on. The next section calculates an alternative approach,

referred to as Power Saver mode. This nomenclature is continued in the program:

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

3.6.2.3 Battery life with extra optimizations

As shown earlier in this section, sending values requires that the Waspmote is on for at least 3 seconds. In addition the XBee uses about five times the energy of the Waspmote. For end devices it is obviously recommended to turn on the XBee as little as possible, within a user defined limit.
Figure 28 shows the same results as the application scenario discussed in section 3.6.2.2 and adds the results for a mode further referred to as Power Saver.
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 28.
As visible on figure 28 *Hibernate* has more influence in Power Saver mode, already extending battery life significantly at a 20 seconds interval comparing to a 10 minutes interval in High Performance mode. Also the energy breakdown graph in figure 29 shows that the interval times must be increased much less before the dominant factor is self-discharge and sleep mode energy consumption, compared to figure 26.
By reducing the sensor measurement accuracy battery life can be extended with modest 3 - 4%, best case scenario. Please see appendix **??** for details.
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.

3.6.3 Sleeping Mesh

What is *really* low power about ZigBee? The answer is already deducible from the previous sectionsn, 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.

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 and a fairly extensive API. The IDE uses the same compiler (AVR) and core libraries as the Arduino IDE. The IDE is ideal to upload small examples and test programs. However, to facilitate programming and to obtain more C/C++ support, expanding the API is a better approach. This will be discussed more in section 4.4.2.
After experimenting with the ZigBee sleep modes it became clear that some unstable results caused by delayed pending messages for end devices could not be avoided. The program does support XBee sleep modes but this section will only discuss stable operation modes. 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 [**?**]. So a ZigBee router forced to sleep as well as a ZigBee end device will be considered as an 'End Device'.

4.2 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) to analyse the data received from the other nodes, which will be discussed in section 5. All the other nodes run either a 'Router' program or an 'End Device' program and are designed to collect and send data to the gateway. The only difference between these last programs is that a 'Router' program does not implement the sleep modes. Before continuing to the next sections, please have a look at the full 'End Device' program flowchart shown in figure 31.

4.2.1 Initialization

Figure 9 shows the simplified underlying logic of a dynamic C implementation of ZigBee at the application level. In one 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 'End Device' program running.

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 entered via the Libelium IDE. To ensure stability it takes about 8 seconds to write the

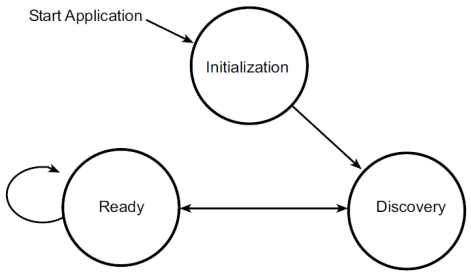


Figure 9. ZigBee Application State Machine

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 its physical sensor layout settings. Until such an 'ADD_NODE_REQUEST' packet is received the sleeping time will gradually be increased in order to save battery power.

4.2.1.2 Next cycles: Reduced initialization By not resetting the PAN ID but by fetching it from the XBee's memory, the joining process only takes about two seconds[4] (Please see appendix **??** 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 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 measured sensors will be saved and tried to send during the next cycles. In order not to lose results, a system administrator will be notified if a node fails to report for several consecutive cycles.

4.2.2 Measuring sensors

sensor float to bytes conversion... ...

4.2.2.1 Sending samples escaping zeros... ... app ID = 10: 'IO_DATA'... ... see appendix ?? for all our packets... ...

4.2.2.2 Wait for received messages Since this mode does not use ZigBee sleep, end devices can only receive messages when the are turned on. To coordinate this, end devices will check messages for a fixed amount of time just after sensor data has been sent.
This means the Waspmote will only wake up when sensors have to be measured. Whereas in the next section ZigBee sleep is enabled and the Waspmote may also wake for polling, in order to avoid that messages buffered in routers get lost (since Waspmote v1.1 cannot be interrupted by available XBee RF Data).

[4]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 so stability can no longer be guaranteed.

4.2.3 Sending and Receiving

In the API of Waspmote v1.1 Libelium uses an 'Application Header' which is shown in 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 (see figure 35).

The Waspmote and gateway programs use the 'Application ID' field in this header to distinguish different data in sent packets. In a first approach this extra layer we developed also serves as acknowledgement in the communication protocol. Depending on the 'Application ID' the layer contains a sensor mask is (a 16 bit-flag) and data. Examples are 'ADD_NODE_REQ' and 'CH_SENS_FREQ_REQ'. Appendix F.5 shows a complete overview off all options.
To easily switch between the different requests and select the sensors contained in the received mask the program uses static function pointers. An example is given below:

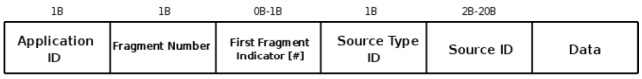


Figure 10. Application Header

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

4.2.4 Without ZigBee sleep mode

The user can set the device role 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 identifier (see [Waspmote ZigBee Networking Guide, 2012] and '**WaspXBeeZBNode.h**'). To better understand the next sections please have a quick look at the program flow chart in figure 31.

4.3 AVR compiler

4.3.1 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 11.

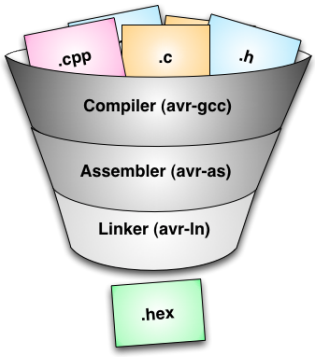


Figure 11. Overview of the AVR toolchain

4.3.1.1 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.

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

4.3.1.3 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.

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

4.3.2 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. Initialized variables and static data such as `char message[] = "An error 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 12. Since there is no hardware supported memory management, separate regians can overwrite each ohter. 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.

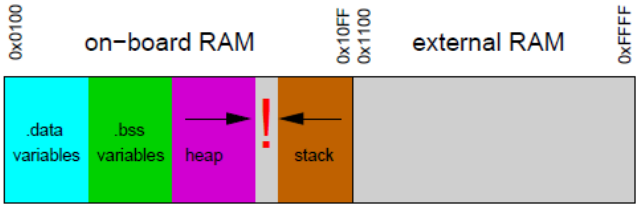


Figure 12. AVR / ATmega1281 standard RAM layout

As discussed in section 4.3.2 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.

4.3.3 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 the message needs to be put in RAM memory first also there precious memory is lost (see the assembly extract in appendix ??).
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 into the heap by doing the following:

```
char str[5];
str[0] = 't';
str[1] = 'e';
str[2] = 's';
str[3] = 't';
str[4] = '\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 matrix which does this operation for us. However, standard C Language macro's 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 Libelium IDE and API

4.4.1 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 (¿ 120KB), 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. Their IDE is far from perfect however, 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 recompile 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.

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.

4.4.2 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. In this section, the most important classes are indicated with a red box and are recommended to explore before starting future programming.

4.4.2.1 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. Please see appendix **??** for a complete overview.
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 13 displays the relationship with the AVR-libc libraries and the Waspmote hardware. For example `typedef unsigned char uint8_t` can be found in 'stdint.h'.
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 including all libraries despite not using them. As a fix Libelium recommends to remove all classes you do not use, and there fields that are used in other classes, 'just' going through the compiler errors one by one. After this our program had enough free memory and showed normal behaviour.

4.4.2.2 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'.

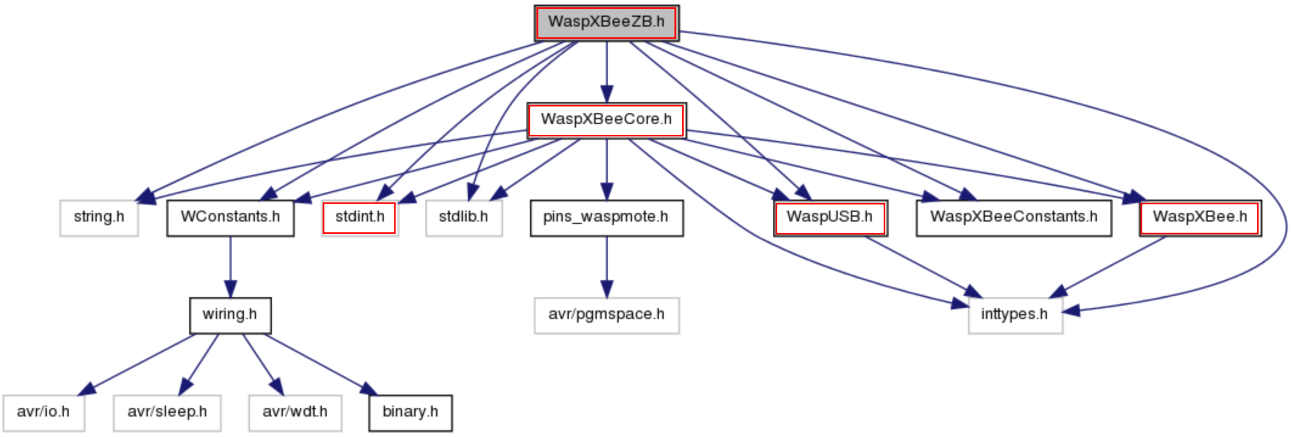


Figure 13. Reduced dependency of Waspmote core libraries

4.5 Sleep options

After default configuration the Waspmote will send only its battery level to the default gateway, check for received commands and go into hibernate mode for 1 minute. In a first approach, which focusses on longer battery life, ZigBee sleep options are not taken into account. With ZigBee sleep enabled, routers and coordinators can buffer incoming RF data for their end device children. However, they can only do this up to 30 seconds [XBee/XBee-Pro ZB RF Modules User Manual, 2012]. When an end device sleeps longer than 30 seconds, they should send a transmission when they wake to inform other devices that they are awake and can receive data. This is exactly what the first approach does, except it completely disconnects the XBee and sleep intervals are much longer.
To make the web service more reactive, this latency has been removed in the second approach. Here XBee sleep is enabled and end devices will respond to commands within 30 seconds. In a last approach we have a look at how the energy usage of the first approach can be optimized even more.

4.5.1 Without ZigBee sleep

This section discusses algorithms which can be used to measure sensors at variable times. It supposes the Waspmote uses either *Hibernate* or *Sleep* mode, completely disconnecting the XBee. Since it is not possible to combine RTC for both hibernate and sleep mode, a tweak has been implemented to use the Watchdog instead. This way the program can automatically chose which sleep mode to invoke, depending on the next duration to sleep. In hibernation mode the node is completely disconnected from the main battery and the program stops. This makes that all variables lose their values and must be stored in EEPROM memory. Each of the next techniques present with benefits and drawbacks and since we are working with embedded systems with limited possibilities, one should also consider to limit the users options to facilitate the calculations.

4.5.1.1 Calculate only the next time to sleep Each algorithm will have to store the individual sleep times per sensor. 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. For an example which demonstrates this process please see appendix C.
This process is fast and simple. However, the main advantage 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:

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

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:

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

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:

$$\frac{4\text{KB}}{32\text{B}} = 125 \text{ writes} \quad (3)$$

4.5.1.2 Calculate all next times to sleep Another possibility is calculate as much as possible or maybe even all sleep necessary times. This algorithm first calculates the least common divider of the given measuring intervals. Afterwards memory is allocated to store the multiples of the values. When the LCM divided by the smallest measuring interval is smaller than the size of the allocated space, all values can be stored in EEPROM. If this is not the case also the last multiplier of the measuring interval must be stored, so when the last value is reached, the next series can be calculated.

Each time the Waspmote wakes up, it will compare its current RTC value with the times stored in EEPROM. The biggest stored time that is an integer multiple of the RTC value is the current position in the array. With this time the sensors to measure and the next time to sleep can be determined.

4.5.1.3 Limit user control Depending on the next time to sleep the program could decide for itself to go into hibernate, deepsleep or sleep with XBee sleep mode. Also, 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.

4.5.2 With ZigBee sleep

4.5.2.1 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 3.6.2.1. End devices can choose between two sleep modes, discussed in the next sections.

4.5.2.2 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

4.5.2.3 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

4.5.2.4 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.
  SP = 30 SN = 3E8

... ...

5 Extracting the data

5.1 Programming language choice

To start programming a choice had to be made. Which programming language and what operating system? Since we only knew Java, C++ and C these languages had our preference. Another requirement was that running the program on an embedded device. Linux has more options is the embedded domain then windows. For instance the raspberry pi is a cheap embedded device with an Ethernet connection and 2 usb ports and a 700mhz ARM processor that can run different linux distributions.
The easiest language is definitely java. It is easy because a lot of libraries are already included in the language. For a gateway several libraries are necessary. A HTTP library to send requests to ipsum and receive requests directly from the web application. Second, an XML or JSON library to format data in these HTTP requests. Since the cloud storage system, Ipsum, uses XML the rest of the gateway also uses XML instead of JSON.
On the other side there is the zigbee network which communicates using RS232. In java this serial communication is also build in. In C or C++ it depends on the OS, for linux it is possible to use read and write to and from file descriptors. So on first sight java seems to be the preferred choice. But since it has to run embedded and processing power might be limited java is less ideal. On a raspberry pi computing power is no problem. Maybe in the future more services should run on the same device so there is an advantages in using C or C++.

So this leaves C and C++. They are both equally fast in execution but C++ is easier to develop since it support more libraries (libraries for C van also be used in C++ but not the other way around) and it has a lot more typechecking so that code that compiles will probably also execute without errors. In C there is a lot of static casting to void pointers and this often results in harder to find bugs. C++ also supports objects and classes that make it easier to create a larger program. There are several other advantages in C++ over C. An entire chapter is dedicated to the advantages of C++ over C in the book thinking in C++ by Bruce Eckel. Also we wanted to have some more experience in C++ and writing this program in C++ sure improved our programming.

5.2 Implementation aspects

For the gateway it took some time to come up with the correct structure. Since a gateway generally has to wait a lot for incoming messages on different connections it is logical to construct a lot of threads which can wait while other threads keep on running. An option would be to use the pipeline pattern where 1 thread generates packets. This would be the zigbee receiver or the webservice. Then a few threads would act as filters on these packets, doing the necessary processing.
One simple pipe would not work since there are more threads generating packets. For instance the ipsum thread can also generate packets to indicate that some packets could not be processed correctly. If ipsum is down this thread needs to store messages in the sql database.
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 program at different places.
A more flexible pattern is the thread pool pattern. This pattern is often used for webservers. Each thread then handles an incoming connection. Although mongoose uses this pattern in this way, it is not entirely the same for the gateway program. The gateway has a few threads, one for each type of connection.

5.2.1 Ipsum

There is one thread for ipsum where the connection to the ipsum database is maintained. Uploading data, changing frequency or in use setting of sensors are all tasks this thread has. The ipsum thread has an incoming queue and an outgoing queue. The incoming queue can contain several packets to indicate what data should be uploaded, what sensor frequencies should change or what nodes should be activated. If for some reason something goes wrong, ipsum will push packets on the outgoing queue to the main thread.

5.2.2 ZigBee

On the zigbee side there are 2 threads, one for sending 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. 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: changes sample frequencies, request IO data, activate a node.

5.2.3 Web service

As mentioned before, mongoose is used as webserver. Mongoose sets up some threads to handle incoming connections. All these threads can push packets onto a queue going to the main thread. In the main threads the XML data is analyzed and the right actions are performed. Possible webservice requests are: add node, add sensor, request IO and change frequency.

5.2.4 Main thread

All intelligence can be found in the main thread. There it is decided what to do with incoming packets. Most often this means creating packets and putting them in the appropriate queues.
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 webservice receives add node and add sensor requests.
The local database also stores Ipsum packets that could not be sent. This could happen when the connection to Ipsum is down or the Ipsum service has crashed. Since losing data is not acceptable, it is also stored in in the sql database.
Also errors could be logged in the database. For now errors are printed to `cerr`.

5.3 Hardware

The gateway consists of a zigbee radio connected to a computer via RS232. This computer could evolve into an embedded linux device such as the raspberry pi (http://www.raspberrypi.org/). It is important that it runs on linux since some libraries are necessary and the serial communication is based on system calls to the kernel. For instance xerces for XML parsing or boost for the multithreading and some other small features.
The library for the webserver is mongoose and the one for en sql database is sqlite. Both libraries are written in C and consist out of 1 header and 1 source file 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. After that you can use read and write functions to receive and send data from the zigbee radio.

6 Discussion

7 Future Work

add security

8 Conclusion

9 Acknowledgements

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-design/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.

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., 900009761 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.

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.

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

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

Wikipedia. Auto-id labs. March 2013a.

Wikipedia. Darpa. April 2013b.

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

A XBee join and send time experiments

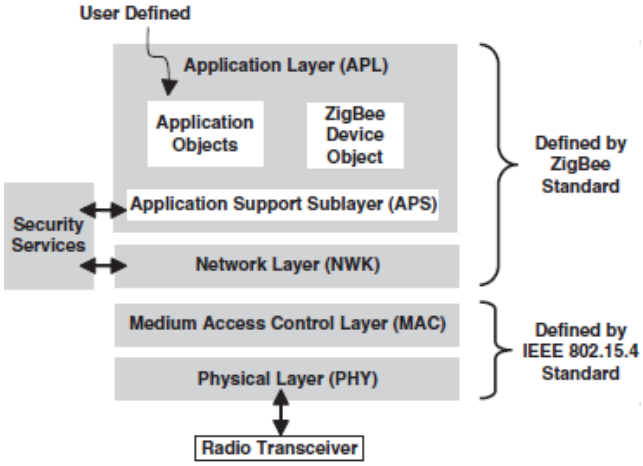


Figure 14. ZigBee Protocol Layers

A.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 15. 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 16. Measurements of join and send time at different distances

A.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 17. XBee join and send times: first time reduction

A.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 18. XBee join and send times: second time reduction

A.4 Sensor measuring experiments

A.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 19. Temperature measurements with and without delay between the readings

A.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 20. Humidity measurements with and without delay between the readings

A.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 21. Pressure measurements with and without delay between the readings

A.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 22. Battery measurements with and without delay between the readings

B Waspmote Architectural Overview

B.1 Waspmote Block Diagram

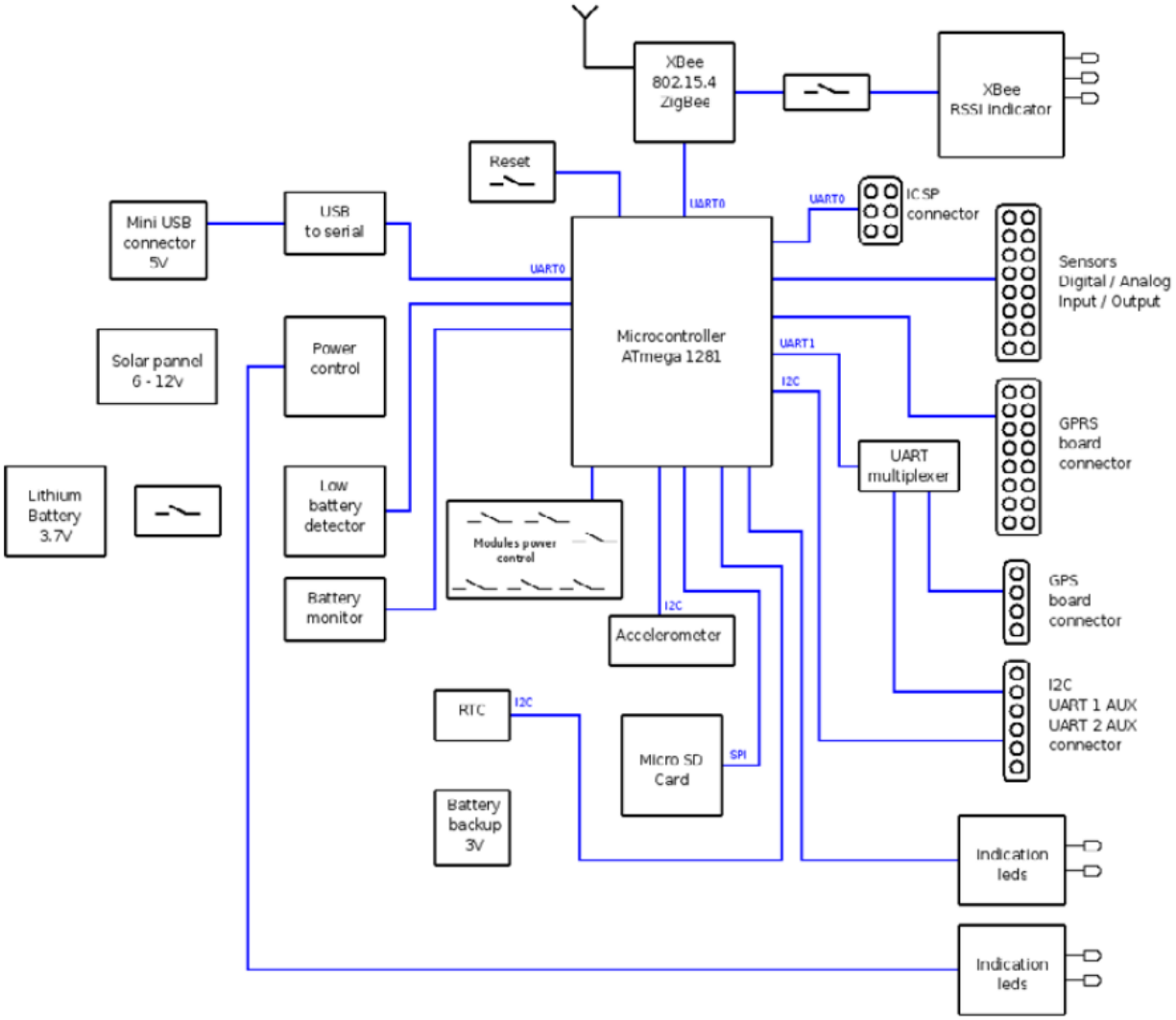


Figure 23. Waspmote block diagram (See3.2.2)

B.2 AVR CPU Core Block Diagram

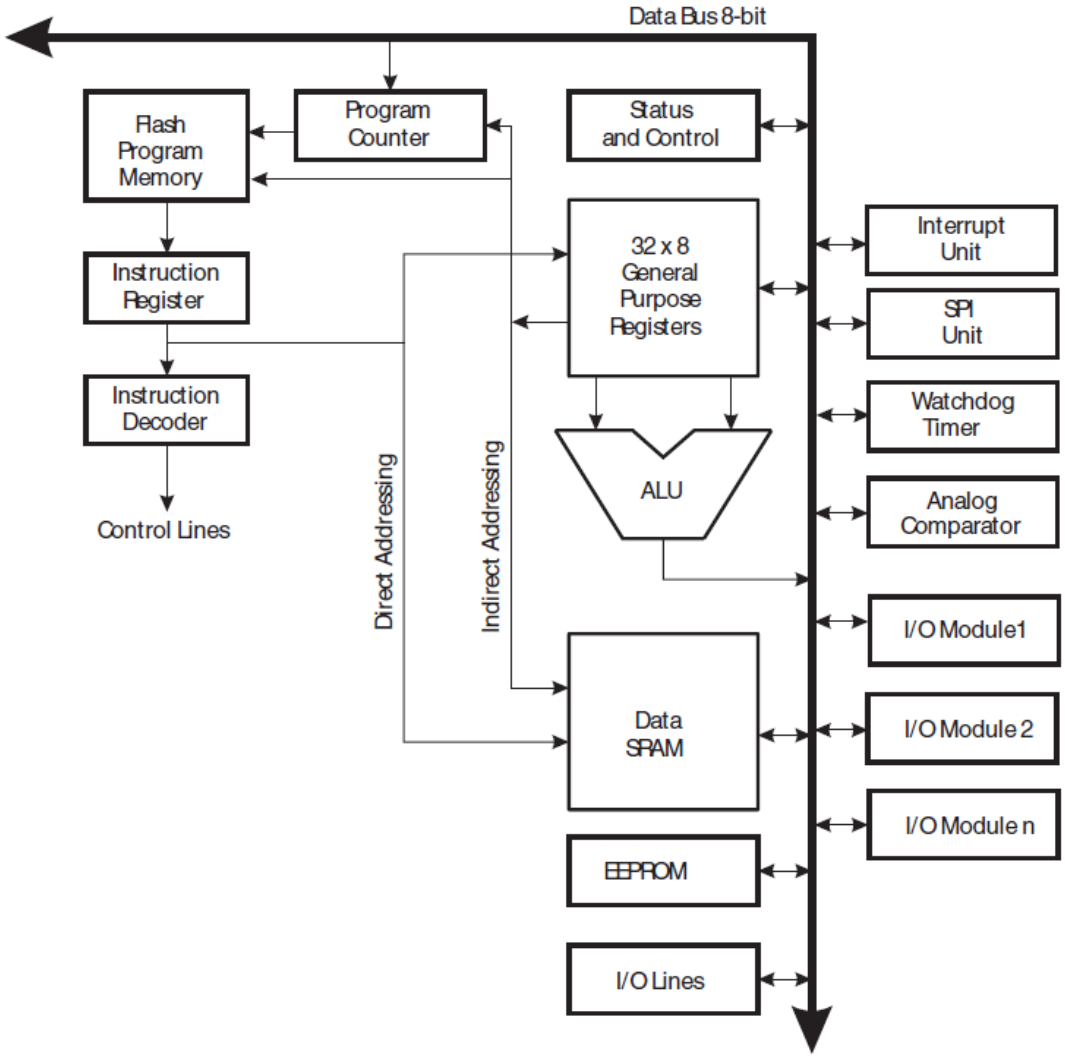


Figure 24. AVR CPU block diagram (See3.2.2)

C Waspmote Variable Sleep Algorithms

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

Table 6. 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 7. Example of sleep algorithm 1

D Waspmote Battery Life Analysis

D.1 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 A.3, meaning that the Waspmote uses *Hibernate* or *Deepsleep* and the XBee is completely disconnected from the network. The results are shown in figures 25, 26 and table 27.

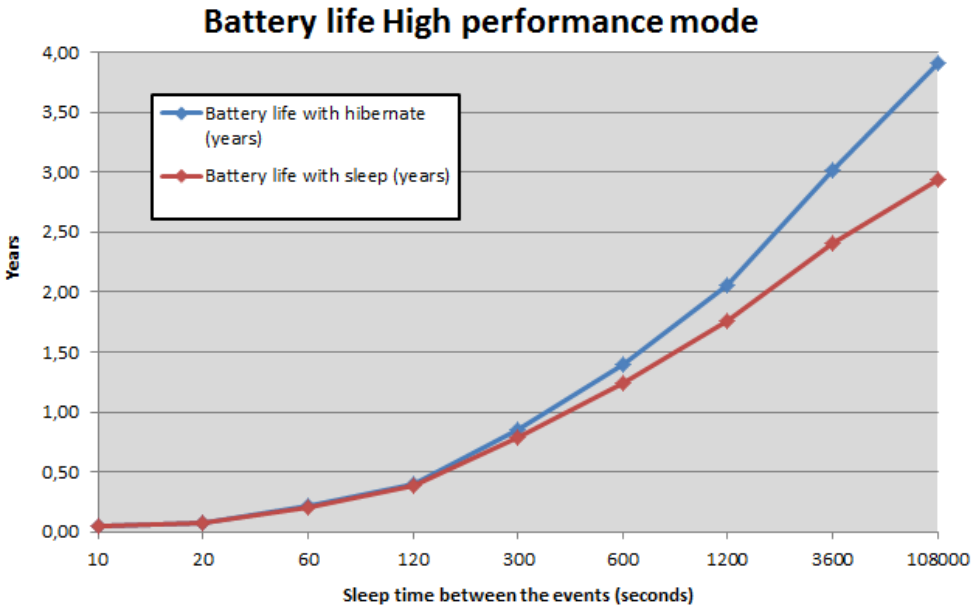


Figure 25. Battery life in High performance mode

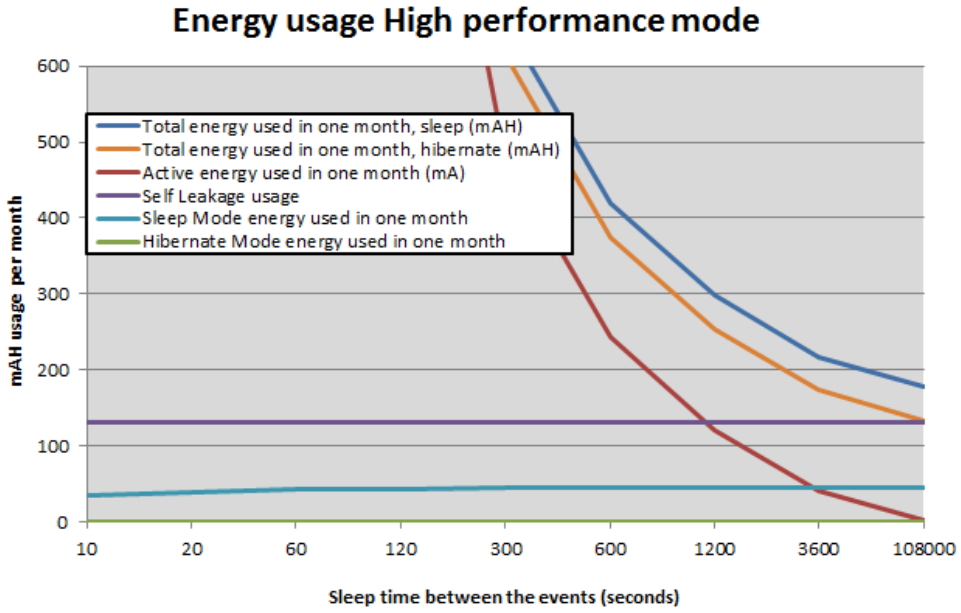


Figure 26. Energy usage in High performance mode

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 27. Energy usage in High performance mode

D.2 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 28, 29 and tables 9 and 30. Finally table 8 summarizes the battery duration in years of both performance and power saver mode.

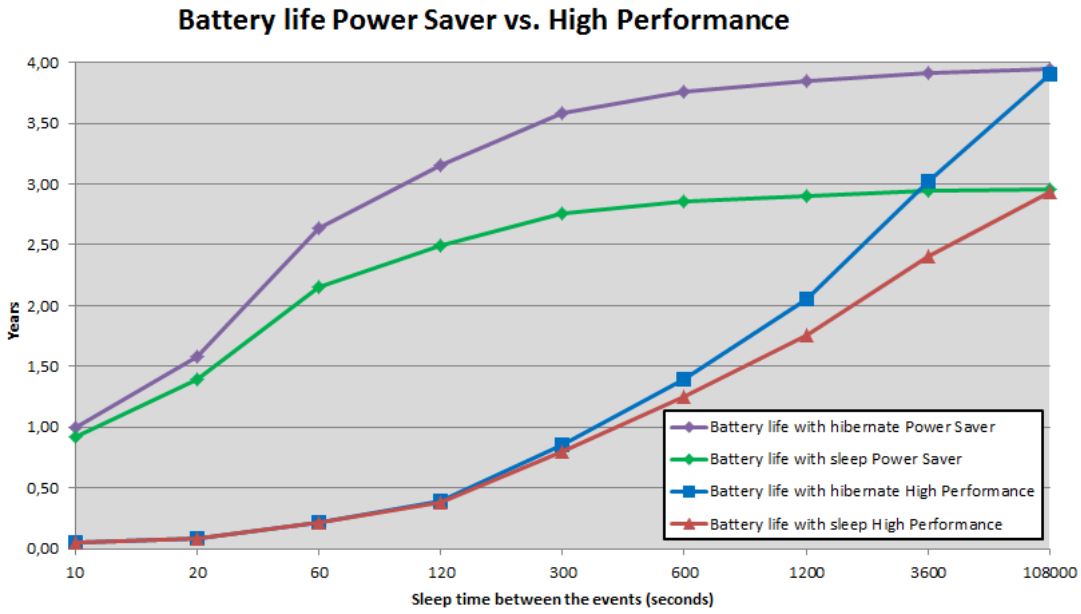


Figure 28. Battery life High Performance vs. Power Saver

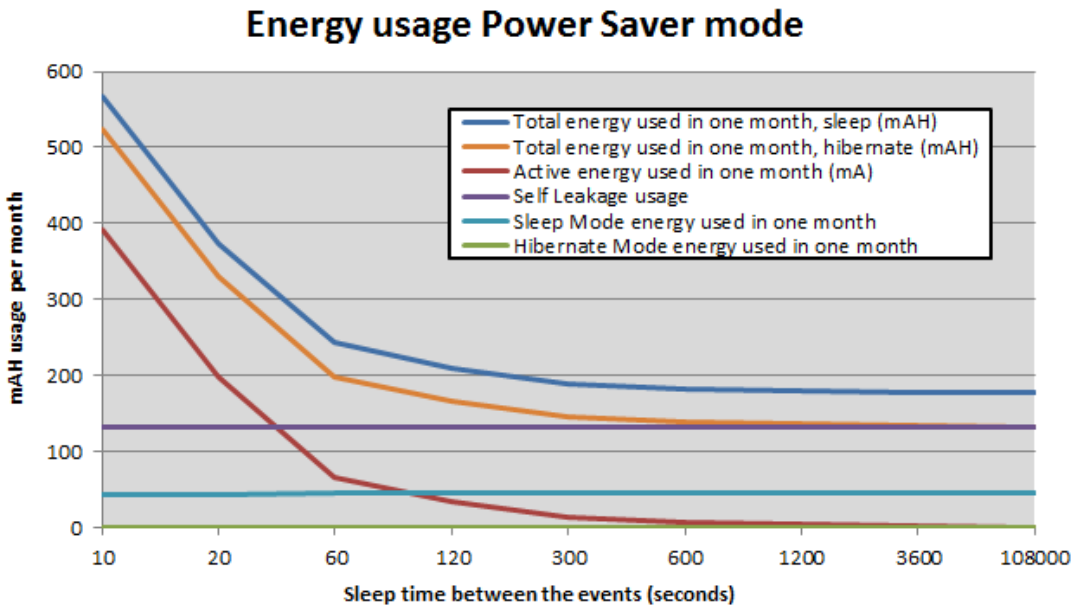


Figure 29. Energy usage in Power Saver mode

	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 30. Energy usage in High performance mode

D.3 High Performance vs. Saver Mode, with ZigBee sleep

E Waspmote Flow Chart

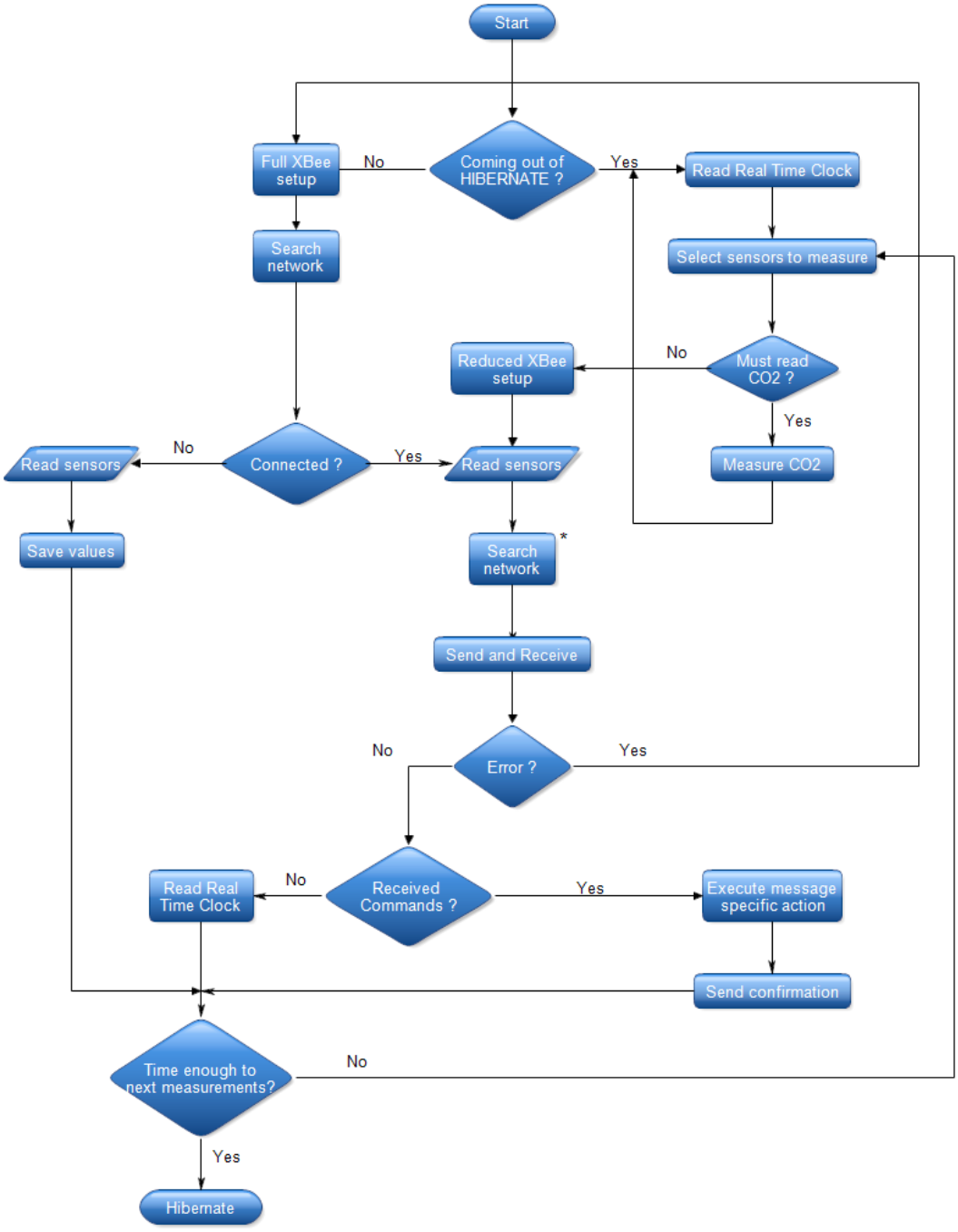


Figure 31. Flow chart of the Waspmote program for end devices

F ZigBee Packet Structure

F.1 API Frames

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 32. API Frame Names and Values (see 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 33. ZigBee Transmit Request Frame Structure (see 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 34. ZigBee Transmit Request Frame Structure (see 2.2)

F.4 Waspmote Application Header in ZigBee API Frame Structure

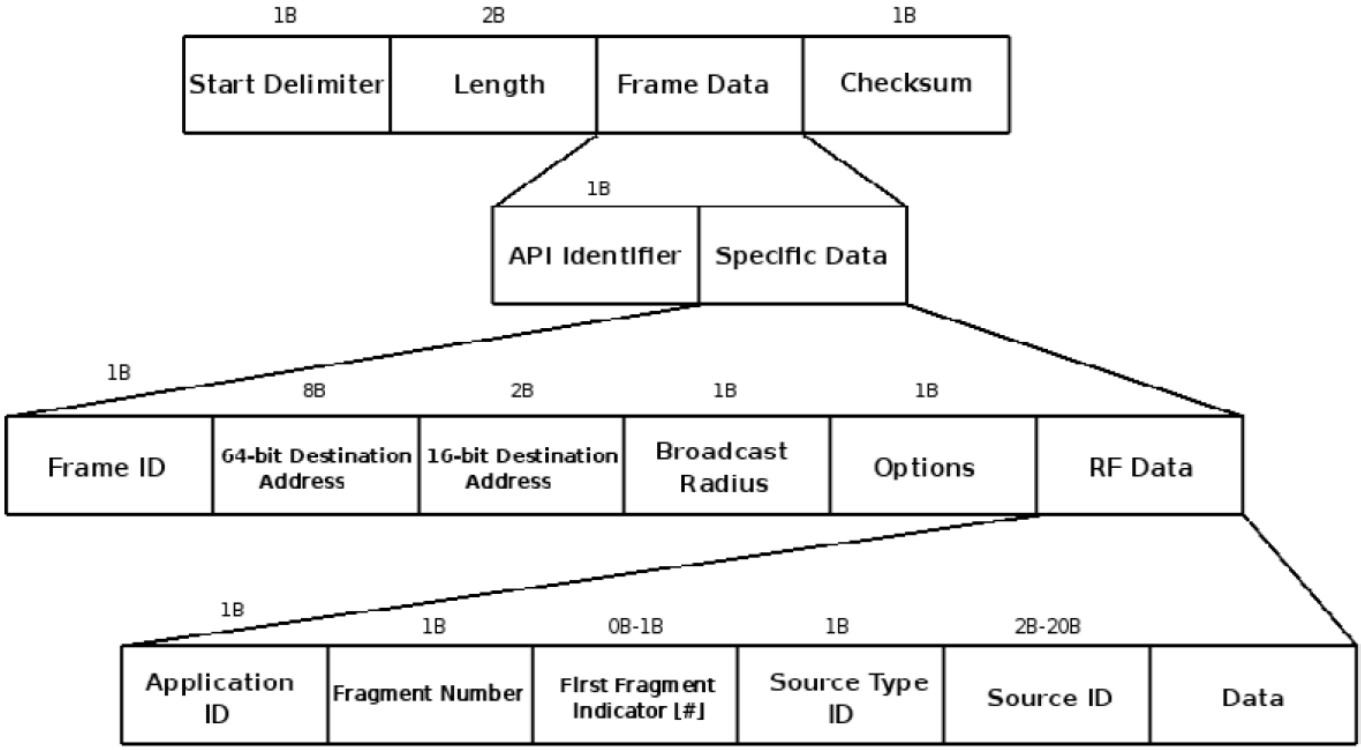
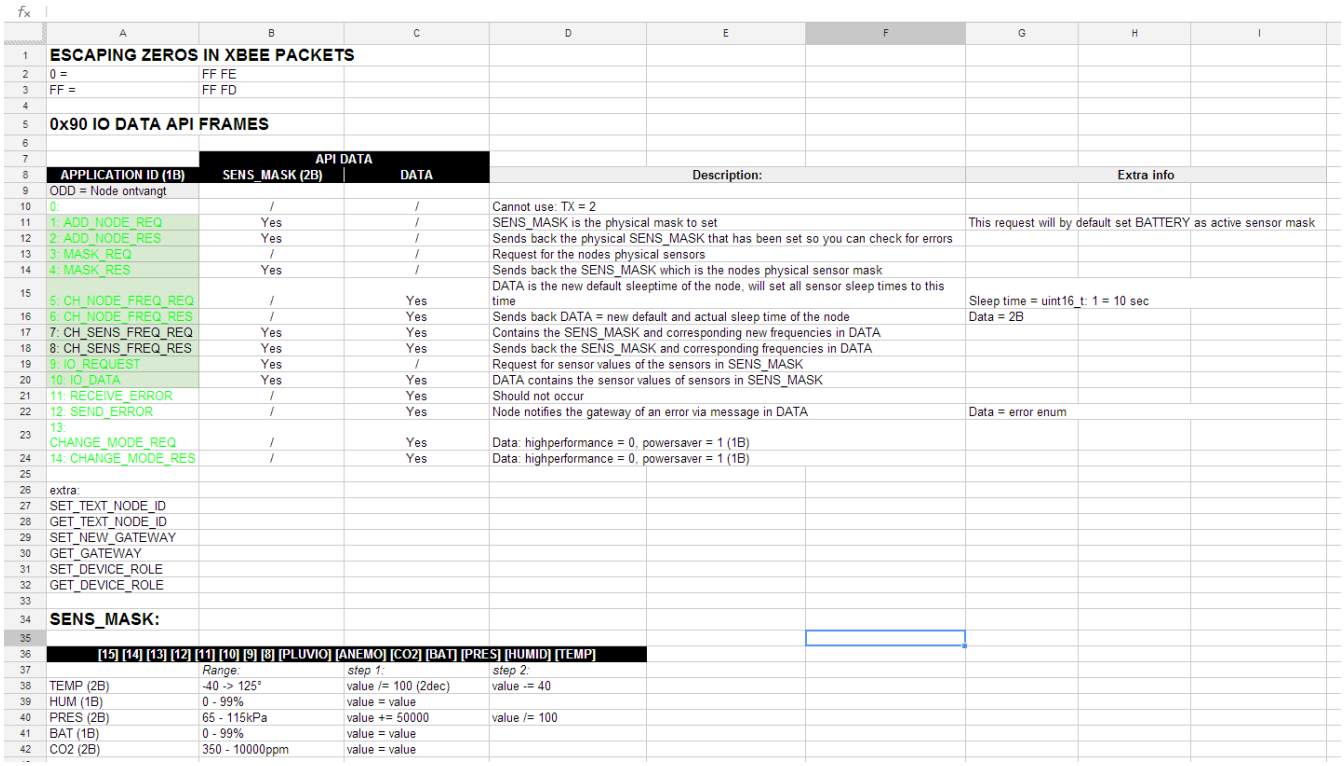


Figure 35. Waspmote Application Header in ZigBee API Frame Structure (see 4.2.3)

F.5 IO API Data Frames for Waspmote and Gateway



	A	B	C	D	E	F	G	H	I
1	**ESCAPING ZEROS IN XBEE PACKETS**								
2	0 =	FF FE							
3	FF =	FF FD							
4									
5	**0x90 IO DATA API FRAMES**								
6									
7		API DATA							
8	**APPLICATION ID (1B)**	**SENS_MASK (2B)**	**DATA**	**Description:**			**Extra info**		
9	ODD = Node ontvangt								
10	0:	/	/	Cannot use: TX = 2					
11	1: ADD_NODE_REQ	Yes	/	SENS_MASK is the physical mask to set			This request will by default set BATTERY as active sensor mask		
12	2: ADD_NODE_RES	Yes	/	Sends back the physical SENS_MASK that has been set so you can check for errors					
13	3: MASK_REQ	/	/	Request for the nodes physical sensors					
14	4: MASK_RES	Yes	/	Sends back the SENS_MASK which is the nodes physical sensor mask					
15	5: CH_NODE_FREQ_REQ	/	Yes	DATA is the new default sleeptime of the node, will set all sensor sleep times to this time			Sleep time = uint16_t: 1 = 10 sec		
16	6: CH_NODE_FREQ_RES	/	Yes	Sends back DATA = new default and actual sleep time of the node			Data = 2B		
17	7: CH_SENS_FREQ_REQ	Yes	Yes	Contains the SENS_MASK and corresponding new frequencies in DATA					
18	8: CH_SENS_FREQ_RES	Yes	Yes	Sends back the SENS_MASK and corresponding frequencies in DATA					
19	9: IO_REQUEST	Yes	/	Request for sensor values of the sensors in SENS_MASK					
20	10: IO_DATA	Yes	Yes	DATA contains the sensor values of sensors in SENS_MASK					
21	11: RECEIVE_ERROR	/	Yes	Should not occur					
22	12: SEND_ERROR	/	Yes	Node notifies the gateway of an error via message in DATA			Data = error enum		
23	13: CHANGE_MODE_REQ	/	Yes	Data: highperformance = 0, powersaver = 1 (1B)					
24	14: CHANGE_MODE_RES	/	Yes	Data: highperformance = 0, powersaver = 1 (1B)					
25									
26	extra:								
27	SET_TEXT_NODE_ID								
28	GET_TEXT_NODE_ID								
29	SET_NEW_GATEWAY								
30	GET_GATEWAY								
31	SET_DEVICE_ROLE								
32	GET_DEVICE_ROLE								
33									
34	**SENS_MASK:**								
35									
36	[15] [14] [13] [12] [11] [10] [9] [8] [PLUVIO] [ANEMO] [CO2] [BAT] [PRES] [HUMID] [TEMP]								
37		*Range:*	*step 1:*	*step 2:*					
38	TEMP (2B)	-40 -> 125°	value /= 100 (2dec)	value -= 40					
39	HUM (1B)	0 - 99%	value = value						
40	PRES (2B)	65 - 115kPa	value += 50000	value /= 100					
41	BAT (1B)	0 - 99%	value = value						
42	CO2 (2B)	350 - 10000ppm	value = value						

Figure 36. IO API Data Frames for Waspmote and Gateway (see 4.2.3)

G Waspmote Program 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 ??.

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

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.