# Wireless Indoor Climate Sensor

## Implementing the Control Unit at Ultra Low Power

J.A. Angevare, I. Jager

Bachelor of Science Thesis

TU Delft Delft University of Technology

Electronic Instrumentation Laboratory

# Wireless Indoor Climate Sensor
## Implementing the Control Unit at Ultra Low Power

Bachelor of Science Thesis

For the degree of Bachelor of Science in Electrical Engineering at Delft University of Technology

J.A. Angevare, I. Jager

June 25, 2012

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS) · Delft University of Technology

Delft University of Technology
Department of

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS) for acceptance a thesis entitled

Wireless Indoor Climate Sensor

by

J.A. Angevare, I. Jager

in partial fulfillment of the requirements for the degree of

Bachelor of Science Electrical Engineering

Dated: June 25, 2012

Supervisor(s):

dr.ir. M.A.P, Pertijs

ir. Z.Y. Chang

Reader(s):

dr.ing. I.E. Lager

dr.ir. R.C. Hendriks

# Preface

The Electronic Instrumentation Laboratory department at Delft University of Technology has designed the sensor parts of a multi-sensor IC called the MIST1431[1], which is manufactured by NXP. This is an ultra-low power sensor chip that measures relative humidity, temperature and ambient light intensity. Because of the low power characteristic the chip is extremely suitable for applications where information about the ambient climate is needed, but energy is scarce.

One specific application for the MIST chip is to be part of wireless climate sensor modules that could be used to implement a smart climate system in buildings. Such modules can provide additional information about the climate in a room.

Now that the MIST chip is being finished at NXP, the need arises for a demonstrator for the smart climate control application. We, a total of six students, have been asked to design and build a Wireless Indoor Climate Sensor demonstrator with the MIST chip. This demonstrator should be fully wireless; furthermore, it should be able to take measurements in a timely fashion and send it to a computer. The demonstrator should also be able to work for at least a year without intervention. The complete set of requirements is found in appendix A.

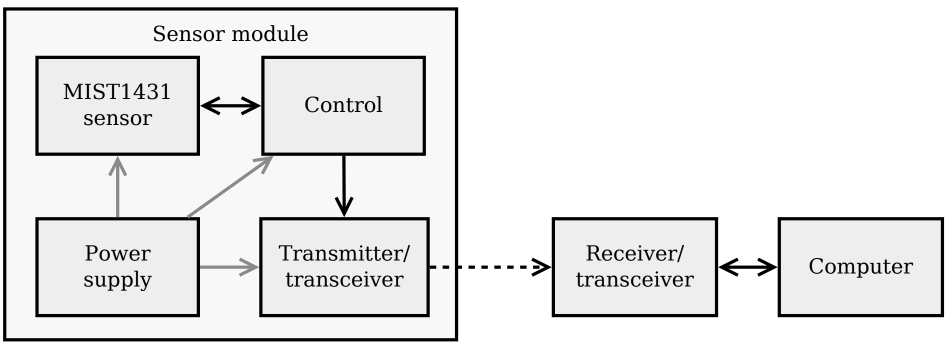


**Figure 1:** Block diagram of the to be designed system

As the system is to be designed by a group of six students and time is limited, a proper subdivision of design tasks was necessary. An additional constraint to the project was that three theses were to be written. The team was split into three groups of two students each.

We have divided the design into subparts (see figure 1), in order to have smaller and better solvable problems and be able to work together. We have divided the students into three groups of 2 students each. Wireless communication is presented in [2], energy management and harvesting in [3] and third, control of the demonstrator, is done by us. The reasoning behind this subdivision was that the three tasks were expected to take approximately equal amounts of time to complete. Additionally, they could be completed in parallel as their design is largely independent of one another.

# Abstract

With the completion of the MIST1431 multi sensor IC a demonstrator developed by the Electronic Instrumentation Laboratory of Delft University of Technology and NXP, a demonstrator that shows the capabilities of the MIST chip was needed. This thesis describes the implementation of the control unit, which has to control the MIST chip and wireless communication module residing in the demonstrator. Temperature and relative humidity had to be transmitted wirelessly at low power consumption.

A method of analysis and assessment of the different methods and hardware was developed. Then a suitable controller was chosen: the LPC1114. After choosing the controller, a study about the functionalities of the controller was done. Software was designed and written. Finally measurements were taken proving the functionalities of the demonstrator.

Testing showed that the demonstrator is capable of sending its measurements once per second to a computer while using a mean of $50.807\mu A$. The result is that the demonstrator meets its requirements.

# Contents

# List of Tables

# List of Figures

# Listings

# Chapter 1

# Introduction

The MIST1431 is an ultra-low-power sensor chip partly made by the Electronic Instrumentation Laboratory, which has recently been finished. For the development of a wireless indoor climate sensor demonstrator, based on the MIST1431 chip and the XBee Series 2 ZigBee module, a control unit is needed. This bachelor thesis presents the process of methodologically selecting a way to control the demonstrator, and the implementation of the subsystem.

The control unit has to communicate with the MIST chip to retrieve the measurement data. The XBee Series 2 module has to be issued to send the measurements to a computer. A power budget of $100\mu$A is available for the entire demonstrator.

In chapter 2 the requirements for the subsystem are presented and the resulting research statements. When the requirements are clear, two ways of controlling the demonstrator are evaluated in chapter 3. Chapter 4 will look into methods for power savings. Subsequently, the implementation of the demonstrator's control unit is treated in chapter 5. Finally, measurements are done to validate the system against the program of requirements. This is presented in chapter 6.

# Chapter 2

# Project Description

In this chapter, the requirements for the project are enumerated in section 2-1. Then, in section 2-2 the research statement, which forms the basis for the literature study is composed.

## 2-1 Requirements

For wireless communication, the XBee Series 2 module[4] was selected[2]. This module interfaces through UART with the control unit. The MIST chip communicates via Serial Peripheral Interface(SPI). The MIST also needs an external clock at 1 MHz. The mean power budget for the entire demonstrator was rated at 100 $\mu A$. This was derived from the requirement that the demonstrator should last at least one year without changing a battery. This section, together with the program of requirements (Appendix A) establishes the requirements for the control unit:

- UART interface
- SPI Master interface
- 1 MHz external PWM signal (for MIST clock)
- Two operation modes: normal mode and demo mode. In normal mode, the sample and transmission rate must be at least once per minute. In demo mode, the sample and transmission rate must be at least once per second.
- Deliver the measured data, while retaining the accuracy of the MIST chip used.
- Measure and transmit relative humidity and temperature.
- Power budget of 100 $\mu A$

## 2-2 Research Statement

The requirements lead to the main research statement that is evaluated in this thesis:

*Provide a means to send data, collected from the MIST chip, and receive commands, through the wireless communication module, at ultra low power.*

This statement is treated in three parts by the following sub statements:

1. *Compare ways to control the wireless communication module and the sensors at ultra low power.*
2. *Find methods to decrease power consumption of the wireless communication module, while maintaining the specified data-rate (see appendix A).*
3. *Find methods to decrease power consumption of the sensors, while maintaining full accuracy.*

Statement 1 is treated in chapter 3, statement 2 and 3 in chapter 4.

# Chapter 3

# Selecting the Control Unit for the Demonstrator

To serve the first research statement, this chapter evaluates two possibilities of controlling the MIST chip via SPI and the XBee module through UART. In section 3-1, FPGAs (field programmable gate arrays) are analyzed. Subsequently, microcontrollers are evaluated in section 3-2. Finally, a method to control the demonstrator is selected.

## 3-1 FPGA Analysis

FPGAs are generally not considered for low power applications, since FPGAs are less energy efficient than dedicated logic due to the overhead of reconfiguration[5]. They are especially suited for complex algorithms and digital signal processing as they can do a lot of calculations concurrently.

FPGAs usually do not have low power modes as contemporary microcontrollers do. However, efforts (e.g. Xilinx Pika Technology[5]) are made by FPGA producers to catch up with the microcontrollers by adding low power features into FPGA cores.

In regular FPGAs, all transistor blocks are always on and running at full speed, consuming a lot of power. On average, only 20% of the transistors in a FPGA are timing critical[6]. So the power usage can be significantly reduced by putting the non-critical 80% of the transistors blocks in low power mode. Power usage improvements such as these and the vast performance and flexibility of FPGAs make them more and more interesting for mobile applications. However, this flexibility comes with its drawbacks. FPGAs do not offer dedicated SPI and UART peripherals. These functionalities have to be built in separately, bringing in more complexity and increased development time.

FPGAs are still less energy efficient than microcontrollers, and they do not ship with dedicated support for SPI and UART. So custom hardware has to be written in order to use these protocols. However, time for this bachelor project is limited. Thus, FPGAs are not suitable for this project, since rapid development and low power consumption are essential.

## 3-2 Microcontroller Analysis

Microcontrollers are fast, cheap and very dynamic. They are dynamic in the sense that the functionality of the microcontroller can be changed with little effort, and without any modification to the physical circuit. This makes microcontrollers an ideal solution to the research statement introduced in section 2-2.

This section describes the comparison of different microcontrollers. First, in section 3-2-1, the method used to compare the microcontrollers is presented. Then in section 3-2-2 microcontrollers are found and compared, using the method of comparison described in section 3-2-1.

### 3-2-1 Method of Comparison

Section 2-1 defined the requirements for the subsystem: a SPI interface is needed for communication with the MIST chip and UART is needed for the XBee module. Most modern microcontrollers implement an UART and SPI peripheral. Therefore, the low-power criterion together with ease of development will be the most important criteria.

This subsection starts with making a few assumptions on the microcontrollers to be compared. Then it gives the method of power consumption estimation. Finally it will list the other criteria, including ease of development.

No benchmarks (a test to determine the calculation power of a CPU) have been found with the calculation power of all the CPU's. In fact, only ARM states a Dhrystone [7, 8] result for its microcontrollers. Therefore all CPU's are assumed to be of equal calculation power.

Assuming that all CPU's are equal in processing power makes it easier to do a power consumption prediction. The microcontroller has to wait for either the ZigBee module or the MIST chip most of the time. In between a lot of data copying, bitwise operations and branches have to be done.

With the assumption of equal processing power, these operations can be modeled as a series of clock cycles. Also, the waiting for either the ZigBee module or the MIST chip can be modeled as a series of clock cycles. Note that all operations can only be modeled as clock cycles because the clock frequency is set to 1 Mhz. Taking the highest clock frequency available will result in more power consumption on behalf of the CPU, since it has to wait most of the time.

Measurements have to be taken each second or each minute, depending on the operation mode of the demonstrator (see appendix A). Also the measurements have to be sent at the same rate for both modes. This means that the mode of operation is periodic, thus in order to predict the power consumption a single period of power consumption is needed. Each period consists of 4 stages:

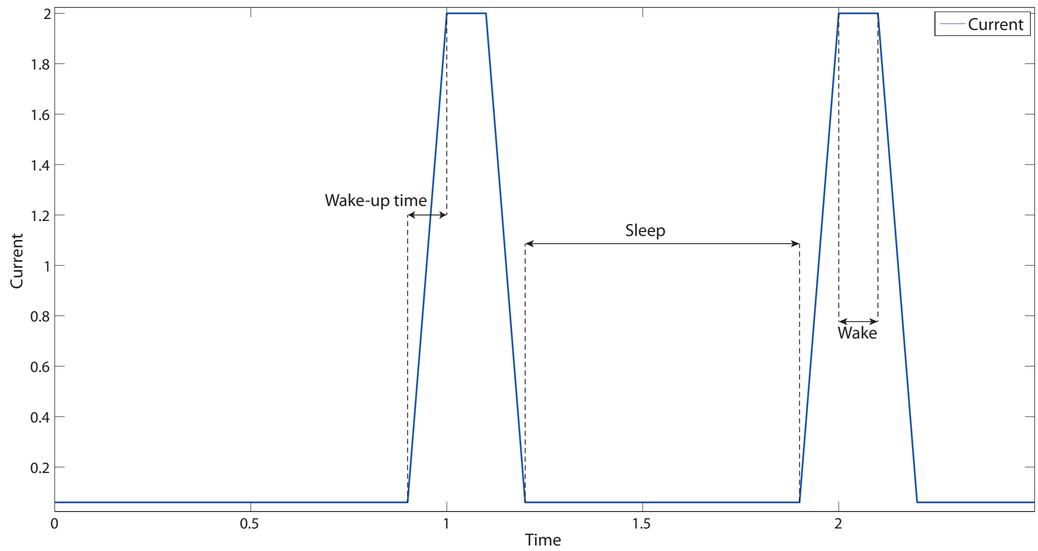


**Figure 3-1:** Power Modes Consumption Estimation

- Waking up
- Taking measurement and sending
- Going to sleep
- Sleeping until 60, or 1, seconds have passed

A graphic display of these stages can be found in figure 3-1. With this model, formulas to calculate the power consumption of each period can now be made:

$$C_{on\_period} = (T_{measurement} + T_{send}) * A_{on} * 1Mhz/F_{on} = T_{on} * A_{on} * 1Mhz/F_{on} \tag{3-1}$$

Where $C_{on\_period}$ is the power consumption of one period while the microcontroller is active, $T_{measurement} + T_{send} = T_{on}$ is the time duration the microcontroller is turned on each period, $A_{on}$ is the on-current given by the datasheet and $F_{on}$ is the frequency associated with the on current. The multiplication with 1Mhz divided by the frequency is done to calculate the power consumption of the microcontroller with a 1 Mhz clock.

$$C_{wake-up} = T_{wake-up} * 1/2 * (A_{on} * 1Mhz/F_{on} + A_{sleep}) \tag{3-2}$$

Where $T_{wake-up}$ is the time required to get out of the deep sleep mode and $A_{sleep}$ is the power consumption of the microcontroller in deep sleep mode.

$$C_{go-to-sleep} = A_{wake-up} \tag{3-3}$$

$C_{wake-up}$ and $C_{go-to-sleep}$ is the power needed to either to get out of deep sleep or into deep sleep mode, both are assumed equal.

$$C_{sleep} = (T_{period} - T_{on} - 2 * Twake - up) * A_{sleep} \tag{3-4}$$

Where $C_{sleep}$ is the deep sleep power consumption per period and $T_{period}$ is the period time, which is 60s.

$$C_{period} = (T_{on} + T_{wake-up}) * (A_{on} * 1Mhz/F_{on} - A_{sleep}) + T_{period} * A_{sleep} \tag{3-5}$$

Where $C_{period}$ is the power consumption per period.

During the measurements and transmitting phase the microcontroller is turned on, thus the power consumption of the microcontroller during this fase is the on state power consumption give by the datasheet. Since the microcontroller will operate at 1Mhz the on-current is multiplied by 1 Mhz and divided by the associated frequency (see equation 3-1).

Now that a model for the power consumption estimation of the microcontroller is found, other criteria can be assessed. Although low power consumption is the most important criterion it is also important that the time to development does not exceed the time restrictions (Time to development includes properties like: good compiler availability, good documentation a development board and available on-hands experience.). If the Electronic Instrumentation Laboratory wants to reuse the demonstrator some time in the future it is also preferred that they have some familiarity with the microcontroller. This will make it easier for them to work with the demonstrator and it also means that they already have the tools needed. The last factor included in the comparison is support (specifically in the form of fora and helpdesk). Support is a handy tool to help solve problems, which is what makes it different from time to development. These last criteria (the criteria excluding the power consumption) are not measurable, that is why a relative weight factor is attributed to them.

### 3-2-2 Microcontroller Comparison

In this section the microcontrollers are compared. The microcontrollers were selected by searching for microcontrollers which the manufacturer has labeled as ultra low power. This search has led to the microcontrollers listed in table 3-1. The data required for the power consumption estimation are found in the datasheets [9, 10, 11, 8, 12, 13, 14, 15, 16, 17] see table 3-1). It must be noted that some wake-up times were missing in the datasheets, these have been set to zero.

Figure 3-2 shows that the power consumption per clock cycle of the microcontrollers is the most significant property. The wake-up time has no major influence on the average power consumption, which was expected since it only has to wake-up once every minute. The sleep current is also negligible because it is small compared to the on state power consumption.

Applying the other criteria was done for the MSP430, the LPC11AXX, the c8051F9806 and the cc430 (see table 3-2). These microcontrollers have a similar power consumption, while the rest of the microcontroller have a higher power consumption. Since the power consumption criterion is still the most important one, the choice will go to one of these more energy efficient

**Table 3-1:** Microcontroller Power Consumption

| | on current [mA] | sleep current [uA] | wake up time [us] | frequency [MHz] |
|---|---|---|---|---|
| PIC16LF1823[9] | 2.2 | 0.03 | 0 | 8 |
| MSP430[10] | 3.6 | 1.3 | 150 | 25 |
| ATtiny1634[11] | 0.23 | 40 | 4 | 1 |
| LCP11AXX[8] | 7 | 2 | 0 | 50 |
| ATiny2313A[12] | 0.2 | 4 | 6 | 1 |
| Si1004[13] | 0.18 | 0.3 | 2 | 1 |
| c8051F980[14] | 0.15 | 0.3 | 2 | 1 |
| cc430[15] | 3.1 | 1.3 | 150 | 25 |
| ATmega128RFA1[16] | 3.7 | 1.2 | 0.38 | 16 |
| efm32g210[17] | 5.67 | 0.59 | 2 | 32 |

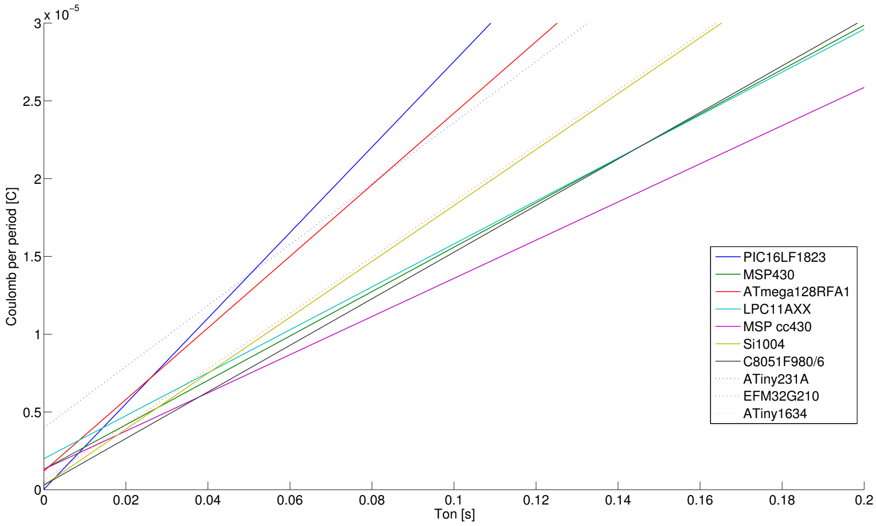


**Figure 3-2:** Microcontroller Power Estimation

**Table 3-2:** Microcontroller Comparison Table

| | PIC16LF1823 | MSP430 | ATiny1634 | LPC11AXX | ATiny2313A | Si1004 | c8051F9806 | cc430 | ATmega128RFA1 | efm32g210 |
|---|---|---|---|---|---|---|---|---|---|---|
| Power Consumption | − | + | - | + | - | - | +- | ++ | − | - |
| Development Time | | - | | ++ | | | + | - | | |
| Support | | + | | ++ | | | +- | +- | | |
| Familiarity | | + | | ++ | | | +- | + | | |

microcontrollers.

The MSP430 and cc430 do not come with a freely available compiler, that is why development time has been judged negatively. A support site for the MSP430 has been set up, however the cc430 (which is an MSP430 with a wireless communication module) does not have much support in the form of libraries or example code. The LPC11AXX has an ARM cortex-M0 core, therefore a lot of coding examples and free compilers are available. Also, the LPC11AXX has a cheap development board which comes with IDE, compiler and a support site. The c8051F9806 has a 8051 core, which is old and long-lasting core and also has a free compiler. Because of it's long life there is a lot to be found on different internet fora.

From table 3-2 follows that the LPC11AX chip is the most favorable. The MSP430 is slightly more power efficient, but decisive was the fact that the LPC1114 is already used on the Electronic Instrumentation Laboratory department and has a development board with free unrestricted compiler, IDE and support site. So LPC1114 was chosen to be the heart of the demonstrator.

# Chapter 4

# Power Saving Techniques

This chapter evaluates different methods for saving power and treats the second and third research statement given in section 2-2. The program of requirements states that the demonstrator should last at least one year on a battery in normal mode. The sensors should do one measurement per minute and transmit the data at least once per minute over the wireless link.

Since a measurement and transmitting combined takes about a 100 miliseconds, the demonstrator will be idle most of the time. Thus, there is a lot to gain in minimizing power consumption when the demonstrator is waiting. This can be achieved by literally turning off certain parts of the device.

The LPC1114 microcontroller has several low power modes built in that provide this functionality. In section 4-1 these low power modes are investigated. Then the sleep modes supported by the XBee module are described in section 4-2. Section 4-3 evaluates the power saving by the MIST chip. Efficient package handling is described is section 4-4 and buffer optimization in section 4-5.

## 4-1 LPC1114 Sleep Modes

The LPC1114 supports four different levels of power modes[18]:

**Active Mode** is the normal, non-power-saving mode. The Cortex-M0 core and the memory are on and are clocked by the system clock. Peripherals are clocked by the system clock or a dedicated peripheral clock.

**Sleep Mode** is the first level power saving mode. The system clock is stopped, so the Cortex-M0 core is off as well as the memory, the related controllers and the internal buses. This diminishes the dynamic power consumption. Peripherals however can still be used if selected in the SYSAHBCLKCTRL register. Also, the processor state and registers are maintained. Waking up can be achieved by interrupts generated by peripherals.

**Deep Sleep Mode** powers down all analog blocks in addition to the Sleep Mode power savings. Only the brown out detector and the watchdog can be selected to remain active. Waking up from deep sleep mode can be done either by an external signal, brown out reset or by a watchdog timer reset.

**Deep Power Down Mode** shuts off the entire chip. Only the data in five general purpose registers is saved. Waking up from deep power down mode can only be achieved by pulling the WAKEUP pin low.

The demonstrator has to transmit the measurements periodically. Also there is no external device to wake up the LPC1114. That means Deep Power Down Mode cannot be used. This leaves Sleep Mode and Deep Sleep Mode. The latter can be used if no peripherals are needed, which is the case when measurements are done and sent over the ZigBee connection. Then the LPC has to wait until it can start the measurement cycle again. While waiting nothing has to be done, so the peripherals can be powered off.

Sleep Mode can be used if peripherals need to be powered. This is the case when the LPC1114 has issued the MIST chip to do a measurement. The MIST chip needs a tenth of a second (see equation 4-1) to finish a measurement. In the meantime, the LPC can go into Sleep Mode while maintaining the clock signal for the MIST chip. Sleep Mode can also be used while sending packets through UART. Sending a character at 9600 BAUD takes equals 1250 CPU instructions at a clock frequency of 12 MHz. That is 1250 instructions of potential sleep time.

Note that it is required to implement asynchronous functionality for the UART and MIST(SPI) (i.e. using interrupts instead of polling) to be able to go into Sleep Mode while the demonstrator is idle.

## 4-2 XBee Sleep Modes

The XBee series 2 module also has two sleep modes for power saving[4].

**Pin/Host Controlled Sleep** is the first low power mode. In this configuration, sleep mode is entered by asserting (logical high) the Sleep_RQ pin. When the XBee is signaled to go to sleep it will finish any transmit or receive operation before entering a low power state. When the same pin is de-asserted (pulled to ground) the module will wake up again.

**Cyclic Sleep** sets an interval for the XBee module to wake up periodically to check for or to send RF data.

The LPC will be controlling the state of the demonstrator. In order for the LPC to have more control, the XBee module will be configured in Pin/Host Controlled Sleep. The LPC will issue the XBee module when it is time to transmit or sleep.

At least each half minute the ZigBee module needs to poll the coordinator in order to not loose its connection. This means that in demonstrator mode at least the LPC and the ZigBee module need to wake up. If the program of requirements permits it, it will be interesting to see if in demonstrator mode the demonstrator can send three measurements per minute.

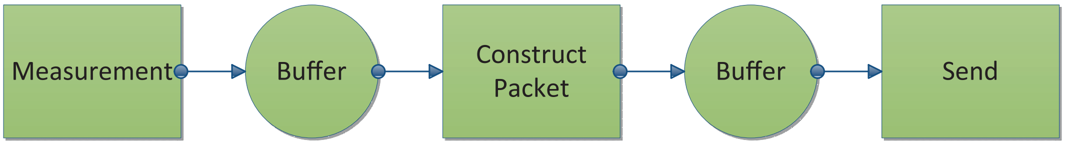


**Figure 4-1:** Circular Buffer Implementation

## 4-3 MIST power saving

So far power saving features for the control part and the wireless communication part of the demonstrator have been assessed. Contributing for the sensor part, the MIST chip offers a deep power down mode to save power[1]. This mode is entered by pulling the RESETn pin to ground. When in deep power down mode the MIST consumes $< 0.3nA$. De-asserting the RESETn pin results in rebooting of the MIST. After $1.5ms$ the chip is ready to accept commands again. A temperature measurement takes, at full resolution (12 bit), 100 miliseconds. Humidity measurements take about 18 miliseconds, which results in:

$$T_m = T_{wake\_up} + T_{temperature} + T_{humidity} = 1.5 + 100 + 18.0 = 119.5ms \tag{4-1}$$

These waiting times are ideal for sleep mode, since deep sleep mode cannot sustain the peripherals. Making the MIST code perform asynchronously will enable the microcontroller to also perform other tasks.

## 4-4 Efficient Package Handling

The demonstrator would perform optimally if 10 measurements are sent in each packet. However the program of requirements states that the demonstrator has to send at least one sample each minute. In order to keep the power consumption to a minimum it was decided that the demonstrator sends one measurement each minute in normal mode.

The program of requirements requires the demonstrator to send at least one measurement each second in demo mode. Since measurements take a little over 119.5 milliseconds, accounting for microcontroller times also, it is technically possible to send 4 measurements each second. However this would not increase functionality while decreasing performance by consuming more power. Therefore it is decided that in demonstrator mode one measurement is sent each second.

## 4-5 Buffering Optimization

Circular buffers are often used in low level applications. Circular buffers are fast first in first out streams. Therefore they seem ideal for use in the demonstrator.

However it was decided that the demonstrator is not going to use circular buffers. Streaming would be very nice to have during the sending of packets or the storing and translating

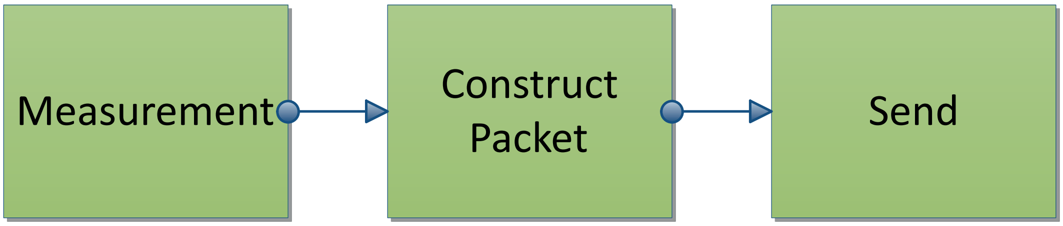


**Figure 4-2:** Buffer Handle Implementation

of sensor data. But as can be seen in figure 4-1 it would require more data copying than necessarily needed. Since sending data is only done once every time the controller is awake no stream is needed.

It was decided that the demonstrator needs to have a pool of buffers. If some part of the code is in need of a buffer it can request a buffer from the pool. Passing data from one point to another would be done by passing the buffer handle (a pointer to the first address of the buffer along with the size of data in the buffer). Passing along the buffer handle also means passing allong ownership of the buffer. Since the buffer pool has an endless amount of buffers it is also required that buffers are released (returned to the buffer pool) after usage.

The buffer pool with buffer handles are an excellent solution for this application, however when large data streams have to be handled, it is recommendable to use circular buffers.

Chapter 5

# Control Unit Implementation

This chapter describes how the control unit of the demonstrator has been implemented. Chapter 3 described the reasons to go with the LPC1114 as control unit. Chapter 4 explained how energy can be saved by the microcontroller, the MIST chip and the XBee module. The software designed for the LPC will be explained top-down. First the high level software design will be explained in section 5-1. Then the lower level software is described in sections 5-2 to 5-4. Section 5-5 describes the data protocol used for transmitting the measurement data. All corresponding source code can be found in Appendix B

## 5-1 High Level Software Design

On the highest level, the software design consists of four parts: WICS Main Loop, Power, ZigBee and MIST. This is illustrated in figure 5-1. The main loop controls the MIST driver to acquire the measurement data, it uses the ZigBee driver to make the XBee module transmit data and it uses the Power functions to manage sleep and deep sleep mode in order to save power. The main loop also makes it possible to switch between normal mode and demonstrator mode. Switching can be done via a wireless message over ZigBee. In addition to the two modes, the sample rate and transmitting rate are configurable individually as well.

The main loop cycle is illustrated in figure 5-2. It starts with waking up from deep sleep and turning on the XBee module, which checks for incoming messages. If there are any, they will be processed. Then, the measurement process starts, and concurrently any available data will be transmitted (data from earlier cycles). After that, XBee will be turned off and the LPC goes into deep sleep.

## 5-2 MIST Implementation

The MIST code is responsible for dealing with the MIST chip. The MIST chip communicates through SPI with the LPC(see figure 5-3). In addition, the MIST chip also needs an external

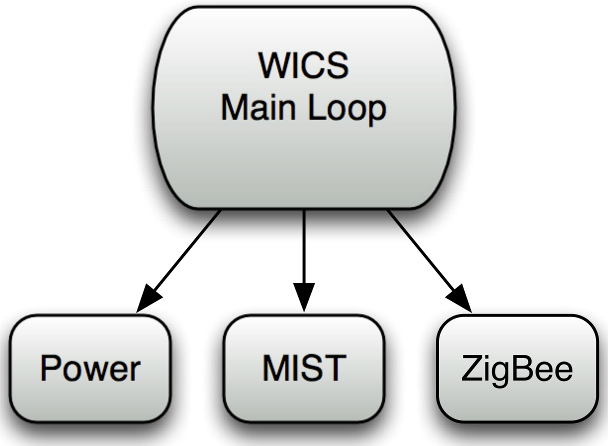


**Figure 5-1:** High level software design

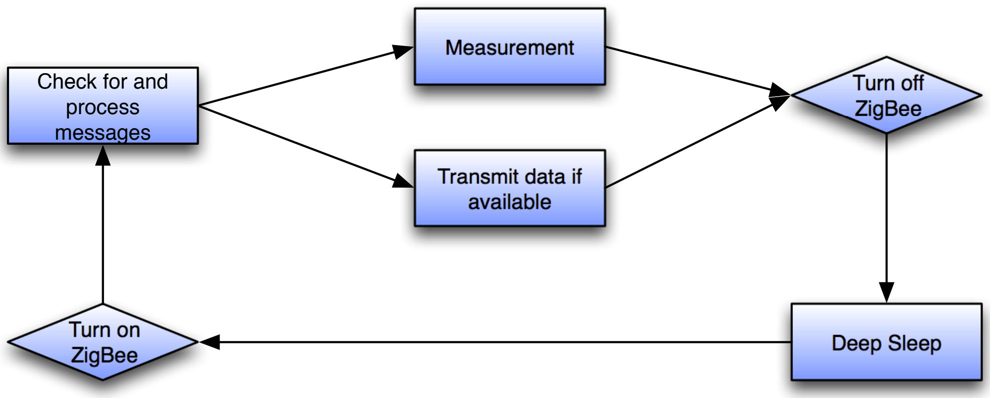


**Figure 5-2:** High level flow chart

1MHz clock, which is also generated by the LPC. When measurement data is needed, MIST follows the steps as illustrated in figure 5-4. First the MIST is enabled. This means the clock is generated on a PWM pin. Then the temperature measurement is started. Since this measurements takes over 100ms the LPC goes into sleep mode during this measurement. The 1MHz clock is still generated in sleep mode. When the temperature measurement is finished, the data is retrieved and the humidity measurement is started. During this measurement sleep mode is invoked as well. After retrieving the humidity data the MIST is disabled.

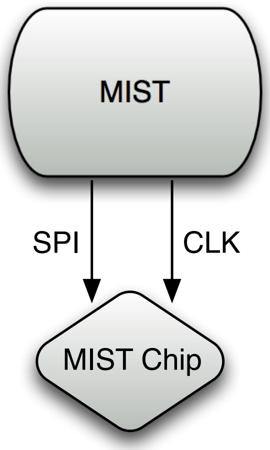


**Figure 5-3:** MIST topology

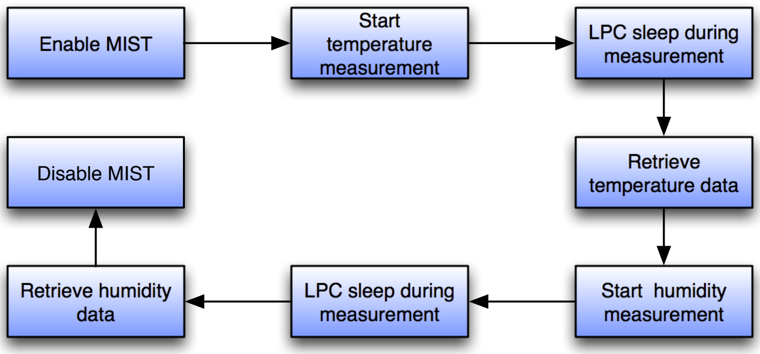


**Figure 5-4:** MIST Driver flow chart

## 5-3 ZigBee Implementation

ZigBee is the driver for the XBee module. It realizes the communication with the XBee module via the LPC UART peripheral, see figure 5-5. All processes in the ZigBee driver run asynchronously. So they work with interrupt requests in stead of polling. When ZigBee receives data to send, first a buffer is allocated. Then the ZigBee packet is constructed and stored in the buffer. Then a transmit is requested for the data in the buffer, see figure 5-6.

The ZigBee packet (see figure 5-7) is constructed around every data set to be sent. Every packet starts with 0x7E as start byte, followed by two bytes that indicate the packet size. Then, an arbitrary ID byte can be assigned just before the actual data is inserted. Finally a checksum is calculated over the entire packet except the start byte. Also all bytes that happen to match a XBee control byte are escaped[4].

The transmit routine waits for a transmit request. At a request it will issue the UART to send

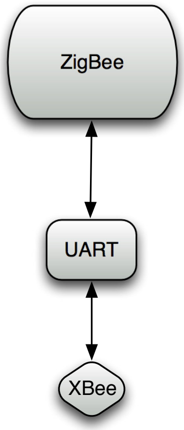


**Figure 5-5:** ZigBee Topology

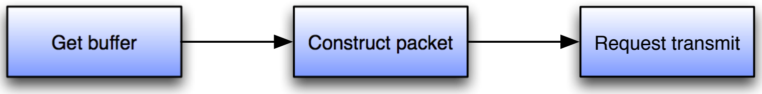


**Figure 5-6:** ZigBee preparation for transmit

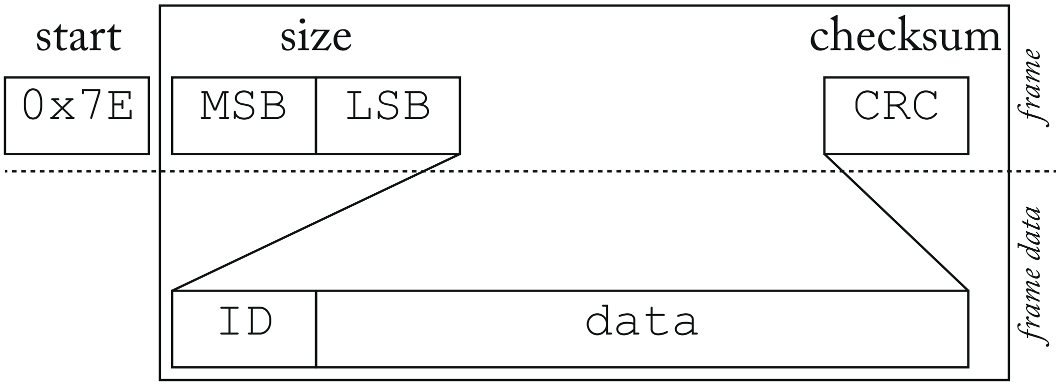


**Figure 5-7:** ZigBee Packet

the first byte. Then it checks if there are any bytes left to send. This is illustrated in figure 5-8.

The XBee module can also receive data. If data is received, UART triggers an interrupt in ZigBee, which will store the data byte for byte in a buffer. When the entire packet is received, the buffer is sent to the handler, which will decode the packet. See figure 5-9 for the corresponding flow chart.

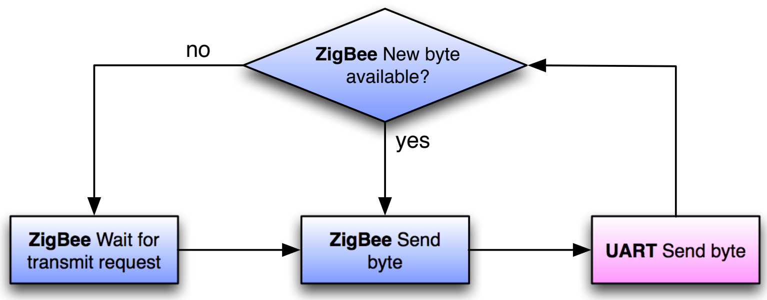


**Figure 5-8:** ZigBee transmit flow

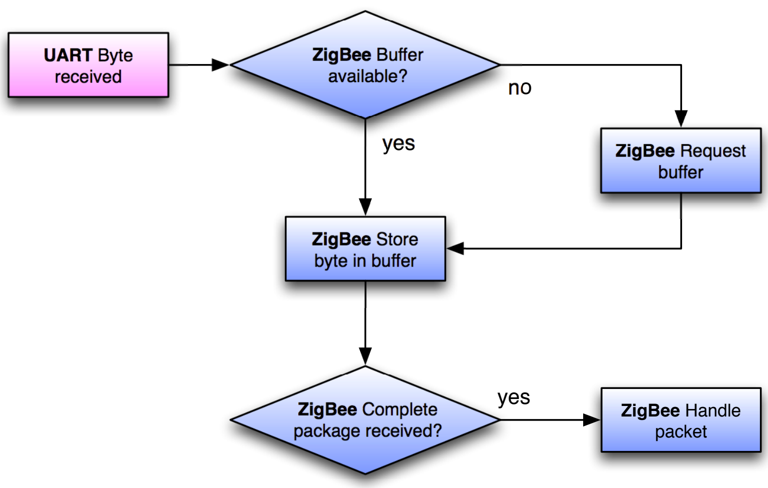


**Figure 5-9:** ZigBee receive flow

## 5-4 Power Implementation

As explained in chapter 4 the LPC needs to be able to enter Sleep Mode and Deep Sleep Mode. The Power part of the microcontroller source code has three functions: One function to initialize the watchdog oscillator and the counter needed to wake up from sleep and two functions to enter either sleep or deep sleep mode.

The watchdog oscillator runs at the lowest possible frequency, 0.5/64MHz in which 64 is the frequency divider and 0.5 is the lowest possible RC oscillator frequency.

Waking up from sleep mode is done through either a timer interrupt or any other interrupt that occurs during sleep. For the timer interrupt, timer 32B1 is used driven by the watchdog oscillator. The timer generates this interrupt when the timer value matches the value stored in the timer match register. This register is used to control the time the LPC is in sleep mode.

Deep sleep mode uses timer 32B0 to drive an external pin to invoke the start logic which in its turn wakes up the LPC. The timer is driven by the watchdog oscillator, which is the only oscillator running in deep sleep mode.

## 5-5 WICS data protocol

The measurements are sent to a pc in a WICS data frame. The data frame can hold a maximum of 10 measurements. This is because a XBee packet holds a maximum of 72 bytes of user data and one packet of measurement data is 7 bytes. Figure 5-10 shows the structure of the data frame. Sequence ID indicates the packets number in the packet sequence. Interval is the measurement interval in seconds. The status flags indicate MIST sensor errors and harvesting errors and battery status.

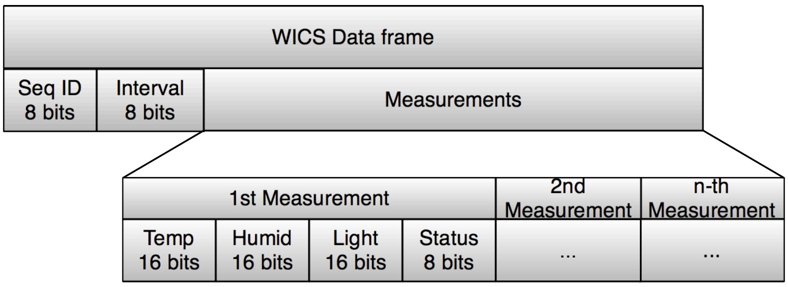


**Figure 5-10:** WICS Data Frame

# Chapter 6

# Measurements

This chapter explains how testing of the demonstrator and subsystems is done in section 6-1. Then section 6-2 presents and evaluates the test data. First power consumption will be looked into. Then the data rate will be checked against the specifications in the program of requirements (see appendix A) in section 6-3.

The demonstrator supports 2 different operation modes: Normal mode and Demo mode. In Normal mode measurements and data transmissions should be done once per minute. In Demo mode once per second.

## 6-1 Measurement method

The measurements were done with the Agilent 6613C power supply [19], Agilent 34401A digital multimeter [20] and the DSO6034A scope [21]. The digital multimeter was used for measuring the deep sleep mode current of the circuit (see figure 6-1). This was done because the scopes have an offset, which had to be compensated for. The measurements taken with the scope (see figure 6-2)start and end with the demonstrator in sleep mode, these parts are

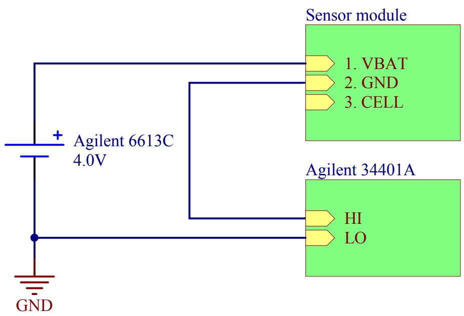


**Figure 6-1:** Average Sleep Mode Current Measurement Setup

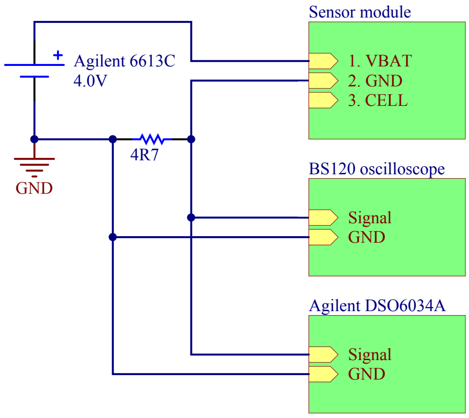


**Figure 6-2:** Average On Mode Current Measurement Setup

calibrated to zero. Then the measurements are transformed to a current, using Ohm's law: $I = U/R$, and finally the sleep current is added.

The measurements are taken this way because the digital multimeter is very accurate at measuring the demonstrator current, but not at high sample frequencies. The scope on the other hand can sample at high enough frequencies but has an offset. Also the scope has to measure the current by measuring the voltage over a shunt resistor, and thus loses some accuracy.

## 6-2 Power Consumption

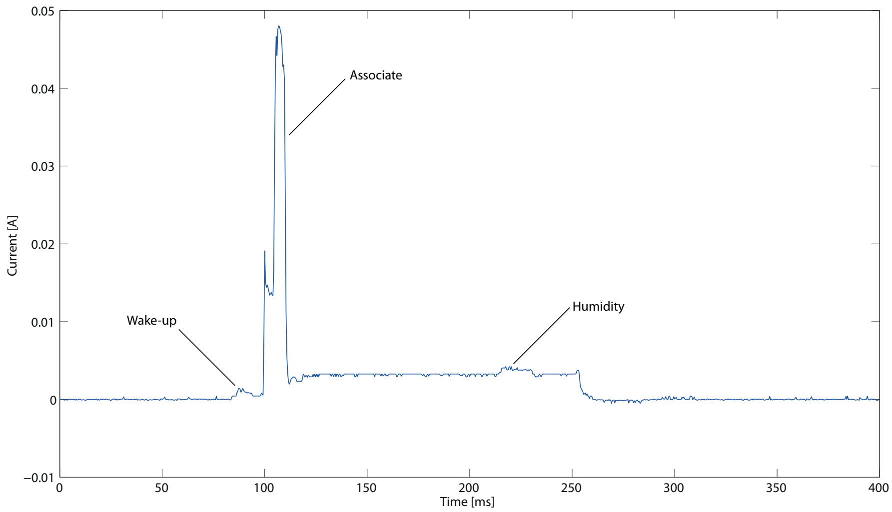


**Figure 6-3:** Association Period Current Consumption Graph

In figure 6-3 a measurement period can be seen. During these measurement periods the XBee module is only turned on to keep its association (connection) with the coordinator. This is done because after about 28 seconds of sleep time the ZigBee module will lose its association. Since the microcontroller is turned on it might as well gather temperature and humidity samples. Note that just before the ZigBee module power consumption peak occurs, a small peak can be seen, which is the microcontroller waking up. Also at the end of the period the MIST is done with it's temperature measurement, a small peak can be seen which is the microcontroller waking up to setup a humidity measurement.

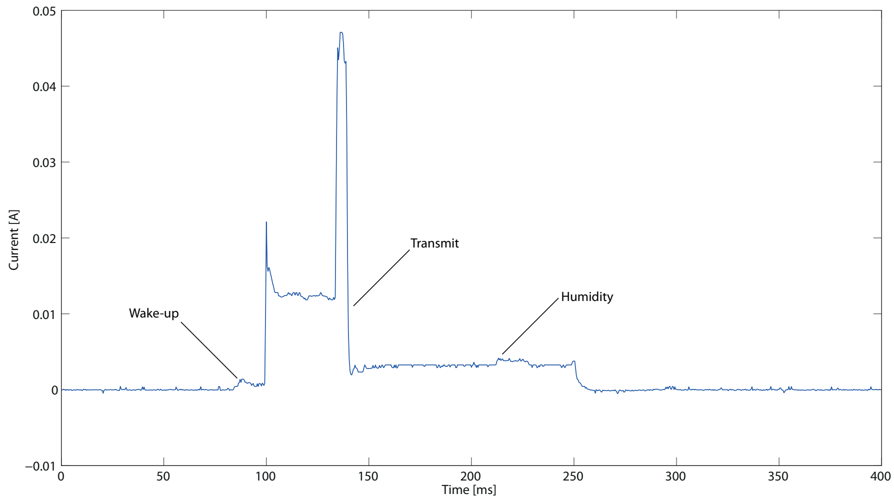


**Figure 6-4:** Send Period Current Consumption Graph

Figure 6-4 depicts a period in which the ZigBee module sends its data and the MIST takes a measurement. As can be seen the ZigBee is turned on longer than the duration of a measurement period.

Figure 6-5 shows the power consumption when one of more resends are necessary. The demonstrator uses considerably more power than without packet resending, because the ZigBee module is on for a longer time.

Finally all the sampled periods are averaged out and presented in figure 6-6. A lot of noise has been canceled by averaging the samples. But it is still possible to see the peak when the XBee module tries to keep associated, the transmit peak and the humidity measurement bump.

$$A_{average} = A_{sleep} + T_{measurement} / N_{samples} * k_{periods/minute} \sum_{i=1}^{N_{samples}} A_i = 6.027\mu + 44.780\mu = 50.807\mu A \tag{6-1}$$

Equation 6-1 sates the average power consumption. $A_{sleep}$ is the average deep sleep current as measured with fig 6-1. $T_{measurement}$ is the measurement duration which is $400ms$. $N_{smaples}$ is the amount of samples taken and $A_i$ is the $i^{th}$ sample. $k_{periods/minute}$ is the amount of mea-

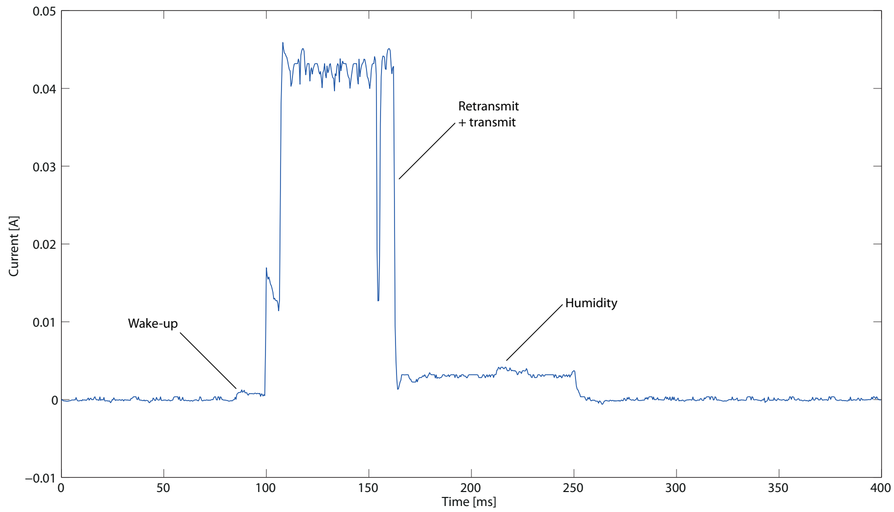


**Figure 6-5:** Resend Period Current Consumption Graph

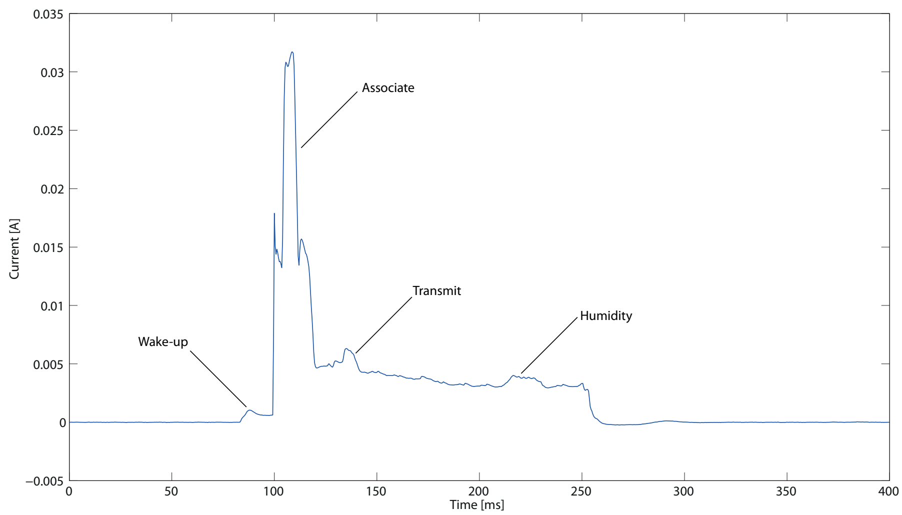


**Figure 6-6:** Average Period Current Consumption Graph

surements taken per minute which is: 3/60. The average current is lower than the maximum power budget of the entire demonstrator, which was $100\mu A$. Even with 3 measurements a minute the average power consumption is only $52\mu A$, which is slighlty more than half the power budget.

## 6-3 Data Rate

After extensive measuring it appeared that sometimes the ZigBee module is not able to deliver the packet in one try and on the next send period it has to send both the old data and the new. This means that the ZigBee module does not have the time during the measurement periods to both keep its association and send its data. This can be solved by allowing the ZigBee to be awake for just a little longer during a measurement period. However, because the ZigBee is not able to send the data in one go means that there is something wrong. The problem seems to be caused by a collapsing voltage supply. After extensive measuring it appears that in 5.2% of all cases a retransmit is necessary. The packet loss was 0%

Receiving data on the demonstrator(sent from a computer) is very unreliable. Using the demonstrator with the energy harvester a packet loss of 100% was measured. Using the demonstrator without the energy harvester resulted in very unreliable receiving. Packets could arrive minutes later, not at all, and even multiple times. The reason for the unreliability is probably the supply voltage drop, due to the power consumption of the ZigBee in combination with the voltage regulator.

When watching the data sent from the ZigBee module to the LPC it became apparent that the ZigBee resets itself due to the voltage drop out. Since the ZigBee module polls its coordinator for packets it is sometimes so that the ZigBee already has the packet but hasn't successfully send and acknowledgement back to the coordinator. This is why sometimes a single packet is received multiple times.

The data rate is configurable in the code. However, packets losses may affect the data rate negatively because of retransmits. The one transmit per second requirement for the demo mode was achieved. It is possible to transmit more frequently by adjusting the time the LPC goes into deep sleep mode every cycle.

# Chapter 7

# Conclusion

This chapter evaluates the measurement results in section 7-1. Then it continues to present recommendations on microcontroller comparison in section 7-2.

## 7-1 Conclusion

The purpose of this project was to find a method to control a wireless indoor climate sensor with a maximum mean power usage of 100 $\mu A$. First a way had to be found to control the MIST1431 multi sensor IC and the XBee Series 2 wireless communication module chosen in [2]. FPGAs proved not appropriate because of the high energy usage and long development time needed. Subsequently microcontrollers were compared on power consumption, performance and usability. The NXP LPC1114 microcontroller turned out the most suitable. Its low power characteristic and availability of on-hand experience were decisive. To reach the low power goal, the LPC1114 can be put into Sleep Mode and Deep Sleep Mode when it is idle. The LPC also controls the operation mode of the XBee module and the MIST chip by putting them in low power modes or by waking them up.

These low power modes were used to keep the control unit of the demonstrator within its power budget of $100\mu A$. The measurement results show that the power consumption of the demonstrator has been kept well within its power budget, even when packets had to be resend.

## 7-2 Recommendations

Comparing the microcontrollers was not a straightforward task. A lot of problems came up, most of them had something to do with the power consumption estimation and subsequently with determining the CPU calculation power.

The modeling of power consumption of embedded systems is an upcoming problem. With the need for low power microcontrollers comes the need for accurate power consumption prediction. A set of cycle-accurate tools have been developed to accurately predict the power consumption of microcontrollers. These tools simulate the processor with the code running on the processor and can predict the power consumption within 3-8%, depending on the simulator [22, 23]. The downside of these simulators is that they need a very accurate model of the processor and the code which will be running on the processor. Extensive testing will be needed. Therefore the usage of cycle-accurate simulators was not an option within a narrow time restriction. However no other tool has been found which predicts the power consumption of the CPU or microcontroller.

A new method to predict/compare the power consumption of microcontrollers is needed. Ideally at this moment a weight factor is found specifying the calculation power of the different CPUs. With these weight factors a CPU specific clock cycle can be found and the power consumption can be estimated. Although there are more than enough benchmarks [24] no benchmark could be found for the CPUs, only ARM gives the Dhrystone benchmark.

The datasheets of the manufacturers do not seem reliable, not only do they not state in what condition measurements are taken, measurements sometimes contradict each other. So a solution is needed for microcontroller benchmark.

Some recommendations on behalf of the ZigBee must also be made. Sometimes the ZigBee module needs to resend a packet to the coordinator. However, it is not able to find the time during an measurement period, in which the ZigBee is turned on only to keep its association. The ZigBee is probably turned off too soon, waiting much longer is not really desirable since the microcontroller is waiting. Therefore it is better to include a time-out in the ZigBee. Then the ZigBee could also be turned off asynchronously after a time-out has occurred.

One requirement was that our implementation could not affect the accuracy of the MIST chip measurements. This has not been tested. Since the XBee module is transmitting at the same time the MIST chip takes samples, the electromagnetic waves due to the ZigBee could affect the samples. To be sure that this requirement is met, this should be tested.

# Bibliography


[1] *MIST1431 Multi-Sensor IC with SPI interface*, 1st ed., NXP, April 2012.

[2] A. van Rijs and D. van 't Hof, "Wireless indoor climate sensor: Wireless communication at ultra low power," 2012.

[3] J. van Straten, "Energy management system for a wireless indoor climate sensor," 2012.

[4] *XBeeTM Series 2 OEM RF Modules*, Digi International, Inc., July 2007.

[5] T. Tuan, S. Kao, A. Rahman, S. Das, and S. Trimberger, "A 90nm low-power FPGA for battery-powered applications," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays.* ACM, 2006, pp. 3–11.

[6] Altera. (2012, May) Stratix series FPGA low power consumption features. [Online]. Available: http://www.altera.com/devices/fpga/stratix-fpgas/about/low-power-consumption/stx-power-about.html

[7] A. Weiss, "Dhrystone benchmark," *History, Analysis, Scores and Recommendations, White Paper, ECL/LLC*, 2002.

[8] *LPC1110/11/12/13/14 Product data sheet*, Rev. 6 ed., NXP, November 2011.

[9] *PIC16LF1823 Datasheet - 8/14-Pin Flash Microcontrollers with nanoWatt XLP Technology*, Microchip Technology.

[10] *MSP430G2x32 Datasheet - MIXED SIGNAL MICROCONTROLLER*, Rev. f ed., Texas Instruments, may 2012.

[11] *ATtiny1634 Datasheet*, Rev. 8303c ed., Atmel, March 2012.

[12] *ATtiny2313A Datasheet*, Rev. 8246b ed., Atmel, September 2011.

[13] *Si1000/1/2/3/4/5 Datasheet*, Rev. 1.0 ed., Silicon Labs, September 2010.

[14] *C8051F98x Datasheet*, Rev. 1.1 ed., Silicon Labs, May 2011.

[15] *CC430F613x Datasheet*, Texas Instruments, December 2011.

[16] *ATmega128RFA1 Datasheet*, Preliminary ed., Atmel, October 2011.

[17] *EFM32G210 Datasheet*, Rev 1.20 ed., Energy Micron, December 2010.

[18] *LPC111x/LPC11Cxx User manual*, Rev. 6 ed., NXP, August 2011.

[19] *6610C Series Single-Output, 40-50 W GPIB Power Supplies*, Agilent Technologies, April 2012.

[20] *Agilent 34401A Multimeter Uncompromising Performance for Benchtop and System Testing*, Agilent Technologies, February 2012.

[21] *Agilent Technologies InfiniiVision 6000 Series Oscilloscopes*, Agilent Technologies, May 2011.

[22] T. Simunic, L. Benini, and G. De Micheli, "Cycle-accurate simulation of energy consumption in embedded systems," in *Design Automation Conference, 1999. Proceedings. 36th.* IEEE, 1999, pp. 867–872.

[23] J. Russell and M. Jacome, "Software power estimation and optimization for high performance, 32-bit embedded processors," in *Computer Design: VLSI in Computers and Processors, 1998. ICCD'98. Proceedings. International Conference on.* IEEE, 1998, pp. 328–333.

[24] R. Weicker, "An overview of common benchmarks," *Computer*, vol. 23, no. 12, pp. 65–75, 1990.

# Appendix A

# Program of Requirements

*Note: the following appendix applies to the complete sensor system, not just the energy system presented in this thesis. See Preface for more information.*

The required product is a wireless indoor climate sensor. It is an autonomous sensor that transmits several parameters about it's environment wirelessly. The product will be used to demonstrate a set of energy efficient sensors, which are developed at the Electronic Instrumentation department at Delft University of Technology. This document lists all requirements and wishes as stated by the client.

In the requirements below, the following definitions apply.

1. *Sensor* - the sensor module to be designed.
2. *Host* - the system which the sensor communicates its data to.
3. *Sampling rate* - the rate at which the sensor takes sensor value samples.
4. *Transmission rate* - the rate at which the sensor transmits (previously recorded) sensor data.

## A-1 Usage Requirements

[1.1] The product must measure at least temperature and humidity. More measured quantities are encouraged.

[1.2] All communications between the sensor and the host must be done wirelessly.

[1.3] The product must function autonomously in terms of energy supply.

[1.4] If a battery is used, the user should be notified when the battery needs to be replaced.

[1.5] The measured quantities should be visible on a computer.

## A-2 Requirements according to the ecological situation of the system's environment

[2.1] The product must function indoors.

[2.2] The transmitter frequency, bandwidth and output power must fall within Dutch regulations.

[2.3] The product must be non-intrusive within it's operating environment, i.e. it should not draw attention to itself.

## A-3 System Requirements

[3.1] If a battery is used, the sensor must operate without battery replacement for at least a year. This requirement assumes the sensor is run in normal (not demo) mode.

[3.2] The range for wireless communication must be at least 5 meters.

[3.3] The sensor must have at least two operating modes in terms of sampling rate and transmission rate: a demo mode and a normal mode. In demo mode, the sample and transmission rate must be at least once per second. In normal mode, the sample and transmission rate must be at least once per minute.

[3.4] The operating mode must at least be selectable using a jumper or switch on the sensor. Being able to set the operating mode wirelessly is a nice to have. Being able to set more sampling and transmission rates is also a nice to have.

[3.5] Having the possibility to set minima and maxima for the measured quantities is a nice to have. If such a limit were to be exceeded, the sensor should wirelessly transmit the current sensor data regardless of transmission rate.

[3.6] To measure the temperature and humidity, the sensor chip developed by the Electronic Instrumentation department at Delft University of Technology and produced by NXP must be used.

[3.7] The chip mentioned above must be visible and influenceable during a demonstration. For instance, it must be possible to breathe on or touch the sensor to demonstrate that the measured quantities indeed change on the screen in such a case.

[3.8] The system must deliver the measured data in such a way that does not reduce the accuracy of the sensor chip(s) used.

## A-4 Installation Requirements

[4.1] It must be possible to install the product without changes to the environment.

[4.2] The installation must be as simple as inserting a battery and installing some software on a computer. In other words, it should be "Plug & Play". It is acceptable if something like a USB dongle is required for communications.

[4.3] Replacing a battery must be possible within a minute.

## A-5 Project Requirements

[5.1] All software written for this product must be well documented.

[5.2] All hardware designed for this product (circuits and circuit board layout) must be well documented.

[5.3] Writing platform independent software is encouraged. The "platform" is defined here as being the operating system for PC based software and the microcontroller (architecture) used for hardware based software/firmware.

# Appendix B

# Source Code

## B-1 ZigBee

**Listing B.1:** ZigBee.h

```c
/*
 * ZigBee.h
 *
 *  Created on: 4 mei 2012
 *      Author: Jan Angevare
 */

#ifndef ZIGBEE_H_INCLUDED
#define ZIGBEE_H_INCLUDED

#include "UART.h"
#include "ZigBee_buffer.h"
#include "ZigBee_constructor.h"
#include "ZigBee_receiver.h"
#include "ZigBee_sender.h"
#include "ZigBee_translator.h"
#include "power_modes.h"

// ZigBee initialze will initialize the ZigBee
// And all it's subparts plus the UART
void  ZigBee_init(void);
// Send a piece of data to the Coordinator
// This sending will be done asynchronously
void  ZigBee_send(char* data, int length);

// This function returns true if a message was ready
// and if a message was returned. If the return value
// is false no message was returned, otherwise a subsequent
// call must be made to ZigBee_done_reading_new_message
// to free up the buffer
int   ZigBee_check_for_new_message(char** data, int* size);
```

```
// Must be called after ZigBee_check_for_new_message returns
// a message. This function free's up the buffer
void  ZigBee_done_reading_new_message(void);

// Wait's for the ZigBee code to stop receiving and
// sending and then set's the ZigBee module to sleep
void  ZigBee_set_sleep(void);
// Wake's the ZigBee module up, the ZigBee module must
// be awake in order for ZigBee to be able to send and
// Receive
void  ZigBee_wake_up(void);

// Explicit request for connection status
// After the ZigBee module sends a reply
// The ZigBee_state will be updated
void  ZigBee_request_connection_status(void);

// Returns the State the ZigBee is in, which is either
// associated (false) or Disassociated (true)
int   ZigBee_get_state(void);

enum ZIGBEE_STATE {
  ZIGBEE_ASSOCIATED = 0,
  ZIGBEE_DISASSOCIATED = 1
};

#endif /* ZIGBEE_H_ */
```

**Listing B.2:** ZigBee.c

```
/*
 * ZigBee.c
 *
 *  Created on: 4 mei 2012
 *      Author: Jan Angevare
 */
#include "ZigBee.h"

// To keep apart status of sending frames
static char* _frames[4];
static int  _frames_size[4];
static int  _frame_id;

// For keeping incomming receives
// will be filled on interrupt
// Should be emptied by main loop
static char* volatile _data;
volatile static int _size;

volatile static int _state;

static void ZigBee_sleep(void) {
  LPC_GPIO2->DATA |= 0x040;
}

static int ZigBee_is_sending(void) {
  return ZigBee_sender_get_state();
```

```
}

static int ZigBee_is_receiving(void) {
  return (ZigBee_receiver_get_state() & 1);
}

// Initialize
void ZigBee_init() {
  _frame_id = 0;

  _data = 0;
  _size = 0;

  _state = ZIGBEE_DISASSOCIATED;

    // Initialize sleep pin
  LPC_IOCON->PIO2_6 = 0xC0; // set GPIO, no pullup
    LPC_GPIO2->DIR |= 0x040; // set output

  ZigBee_buffer_init();
  ZigBee_receiver_init();
  ZigBee_sender_init();
  UART_init();
}

// This handle gets called when ZigBee_sender completes a send request
void ZigBee_sender_send_complete_handle(char* data) {
  ZigBee_buffer_release_buffer(data);

  if (!ZigBee_is_receiving())
    ZigBee_sleep();
}

// Send an amount of data to the coordinator
void ZigBee_send(char* data, int length) {
  char* buffer;
  int size;

  // Just to be sure, wake the ZigBee up
  ZigBee_wake_up();

  // Try to get a buffer for filling
  while(!ZigBee_buffer_get_buffer(&buffer));

  // Construct message and set the frames ID etc, for resend if neccesary
  _frames[_frame_id++] = buffer;

  size = ZigBee_constructor_construct_message(data, length, buffer, _frame_id);
  _frames_size[_frame_id - 1] = size;
  _frame_id %= 4;

  // Loop until we get an ok for sending
  while(!ZigBee_sender_send_frame(buffer, size));
}

// Is called whenever there has been received an whole frame on UART
void ZigBee_receiver_new_message_handle() {
```

```
  char* buffer;
  int size;

  // Get the frame
  ZigBee_receiver_get_message(&buffer, &size);

  // And translate it
  if(ZigBee_translator_translate(buffer, size))
    ZigBee_buffer_release_buffer(buffer);
}

// If an transmit status: successful is received
// put the ZigBee module to sleep
void ZigBee_translator_send_success_handle(int data_frame) {
  if (!ZigBee_is_receiving() && !ZigBee_is_sending())
    ZigBee_sleep();
}

// If an transmit status: unsuccessful is received
// do the same as an successful send, this means
// the data is lost
void ZigBee_translator_send_failure_handle(int data_frame) {
  ZigBee_translator_send_success_handle(data_frame);
}

// Gets called if the ZigBee sends an associated status
void ZigBee_translator_associated() {
  _state = ZIGBEE_ASSOCIATED;
  if (!ZigBee_is_receiving() && !ZigBee_is_sending())
    ZigBee_sleep();
}

// If the ZigBee is dissacociated
void ZigBee_translator_disassociated() {
  _state = ZIGBEE_DISASSOCIATED;
}

// If an received message has been received
// put it in the queue
// The main loop should poll with check for new message
void ZigBee_translator_receive_data_handle(char* data, int size) {
  if (_data)
    ZigBee_buffer_release_buffer(_data);

  _size = size;
  _data = data;

  if (!ZigBee_is_receiving() && !ZigBee_is_sending())
    ZigBee_sleep();
}

// If a new message is ready, get it
// this function returns true if a new message has been received
// else this function returns false
// if this function returns true a subsequent call to
// ZigBee_done_reading_new_message should be done
int ZigBee_check_for_new_message(char** data, int* size) {
```

```
  if (_data) {
    *data = _data;
    *size = _size;

    return 1;
  } else return 0;
}

// This function empties the message queue and releases the buffer
void ZigBee_done_reading_new_message() {
  if (_data) ZigBee_buffer_release_buffer(_data);

  _data = 0;
  _size = 0;
}

// Wait until no more sending or receiving is done
// then put the ZigBee module to sleep
void  ZigBee_set_sleep() {
  while (ZigBee_is_receiving() || ZigBee_is_sending()) {
    ZigBee_wake_up();
    power_modes_sleep(10, 0);
  }

  ZigBee_sleep();
}

void  ZigBee_wake_up() {
    LPC_GPIO2->DATA &= ~0x040;
}

// Explicit poll for connection status
void  ZigBee_request_connection_status(void) {
  char* buffer;
  int size;
  unsigned short command = ('A'<<8) + 'I';

  // Try to get a buffer for filling
  while(!ZigBee_buffer_get_buffer(&buffer));

  // Construct message and set the frames ID etc, for resend if neccesary
  _frames[_frame_id++] = buffer;

  size = ZigBee_constructor_construct_at(command, buffer);
  _frames_size[_frame_id - 1] = size;
  _frame_id %= 4;

  // Loop until we get an ok for sending
  while(!ZigBee_sender_send_frame(buffer, size));
}

int   ZigBee_get_state(void) {
  return _state;
}
```

## B-2 ZigBee_buffer

**Listing B.3:** ZigBee_buffer.h

```c
/*
 * ZigBee_buffer.h
 *
 *  Created on: 9 mei 2012
 *      Author: bap
 */

#ifndef ZIGBEE_BUFFER_H_
#define ZIGBEE_BUFFER_H_

void  ZigBee_buffer_init(void);
// Tries to give a buffer back, all buffers are 128 bytes length
// return true if succeeded, returns false if no buffer could be found
// buffer parameter should point to a char pointer, the char pointer is set
// to the buffer, if succeeded, else is ignored
int   ZigBee_buffer_get_buffer(char** buffer);
// Release the buffer, buffer parameter should point to the buffer
void  ZigBee_buffer_release_buffer(char* buffer);

enum ZIGBEE_BUFFER_STATES {
  ZIGBEE_BUFFER_EMPTY,
  ZIGBEE_BUFFER_IN_USE
};

#endif /* ZIGBEE_BUFFER_H_ */
```

**Listing B.4:** ZigBee_buffer.c

```c
/*
 * ZigBee_buffer.c
 *
 *  Created on: 9 mei 2012
 *      Author: bap
 */

#include "ZigBee_buffer.h"

// Buffers and their states
static char _buffers[4][128];
int   _buffer_states[4];

// Initialize
void ZigBee_buffer_init() {
  int i;

  for (i = 0; i < 4; i++)
    _buffer_states[i] = ZIGBEE_BUFFER_EMPTY;
}

// Request a buffer
int ZigBee_buffer_get_buffer(char** buffer) {
  int i;

```

```
  // Check all buffers for availability
  for (i = 0; i < 4; i++) {

    // If available set buffer, set state and return true
    if (_buffer_states[i] == ZIGBEE_BUFFER_EMPTY) {
      *buffer = _buffers[i];
      _buffer_states[i] = ZIGBEE_BUFFER_IN_USE;

      return 1;
    }
  }

  // No free buffer was found, return false
  return 0;
}

// Set buffer state to empty
void ZigBee_buffer_release_buffer(char* buffer) {
  _buffer_states[((buffer - _buffers[0])>>7)] = ZIGBEE_BUFFER_EMPTY;
}
```

## B-3 ZigBee_constructor

**Listing B.5:** ZigBee_constructor.h

```
/*
 * ZigBee_constructor.h
 *
 *  Created on: 9 mei 2012
 *      Author: bap
 */

#ifndef ZIGBEE_CONSTRUCTOR_H_
#define ZIGBEE_CONSTRUCTOR_H_

// Constructs a message to the coordinator
int   ZigBee_constructor_construct_message(char* source, int size, char*
    destination, char frame_id);
int   ZigBee_constructor_construct_at(unsigned short command, char*
    destination);
#endif /* ZIGBEE_CONSTRUCTOR_H_ */
```

**Listing B.6:** ZigBee_constructor

```
/*
 * ZigBee_constructor.c
 *
 *  Created on: 9 mei 2012
 *      Author: bap
 */

#include "ZigBee_constructor.h"
#ifdef DEBUG
#include <stdio.h>
```

```
#endif

static int _checksum;
static int _offset;
static char _transmit_frame_header[14] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 // 64-bit address
    , 0xFF, 0xFE // 16-bit address
    , 0x00 // Broadcast radius
    , 0x00}; // Options

// Add data to the frame data
void ZigBee_constructor_add_data(char* source, int size, char* dest) {
  int i;

  for (i = 0; i < size; i++) {
    _checksum += (dest[_offset++] = source[i]);
  }
}

// Set size and checksum
void ZigBee_constructor_finalize(char* buffer) {
  int size = _offset - 2;
  buffer[0] = (char)(size>>8);
  buffer[1] = (char)(size);

  buffer[_offset++] = 0xFF - (0xFF & _checksum);
}

// Construct message
int ZigBee_constructor_construct_message(char* source, int size, char*
    destination, char frame_id) {
  _offset = 2;
  _checksum = 0;

  _checksum += (destination[_offset++] = 0x10);
  _checksum += (destination[_offset++] = frame_id);

  ZigBee_constructor_add_data(_transmit_frame_header, 12, destination);

  ZigBee_constructor_add_data(source, size, destination);

  ZigBee_constructor_finalize(destination);

  return _offset;
}

// Construct AT command request
int ZigBee_constructor_construct_at(unsigned short command, char* destination)
    {
  _offset = 2;
  _checksum = 0;

  _checksum += (destination[_offset++] = 0x08);
  _checksum += (destination[_offset++] = 1);
  _checksum += (destination[_offset++] = (command>>8));
  _checksum += (destination[_offset++] = (char)command);

```

```
  ZigBee_constructor_finalize(destination);

  return _offset;
}
```

## B-4 ZigBee_receiver

**Listing B.7:** ZigBee_receiver.h

```
/*
 * ZigBee_receiver.h
 *
 *  Created on: 9 mei 2012
 *      Author: Jan Angevare
 */

#ifndef ZIGBEE_RECEIVER_H_
#define ZIGBEE_RECEIVER_H_

#include "UART.h"
#include "ZigBee_buffer.h"

void  ZigBee_receiver_init(void);
int   ZigBee_receiver_get_state(void);
int   ZigBee_receiver_get_message(char** message, int* size);

// get called by ZigBee receiver whenever a
// new message was received
void  ZigBee_receiver_new_message_handle(void);

enum ZIGBEE_RECEIVER_STATES {
  ZIGBEE_RECEIVER_IDLE = 0,
  ZIGBEE_RECEIVER_RECEIVING = 1,
  ZIGBEE_RECEIVER_MESSAGE_READY = 2,
};

#endif /* ZIGBEE_RECEIVER_H_ */
```

**Listing B.8:** ZigBee_receiver.c

```
/*
 * ZigBee_receiver.c
 *
 *  Created on: 9 mei 2012
 *      Author: bap
 */

#include "ZigBee_receiver.h"

// Buffer for completeley received frames
static char* volatile _b_buffer;
volatile static int _b_size;

// Buffer for receiving of frames
```

```
static char* _buffer;
static int  _size;
static int _offset;
static int _state;

// State for escape characters
static char _escape;
static unsigned int _checksum;

// Initialize ZigBee_receiver
void ZigBee_receiver_init() {
  _b_buffer = 0;
  _b_size = 0;

  _buffer = 0;
  _size = 0;
  _offset = 0;
  _checksum = 0;
  _state = ZIGBEE_RECEIVER_IDLE;
}

// Return the ZigBee_receiver_state
int ZigBee_receiver_get_state() {
  return _state;
}

// Try to get the frame, this function returns true if a frame was ready
// else it returns false
int ZigBee_receiver_get_message(char** message, int* size) {
  if (_state & ZIGBEE_RECEIVER_MESSAGE_READY) {
    *message = _b_buffer;
    *size = _b_size;

    // reset frame buffer and state
    _b_buffer = 0;
    _b_size = 0;
    _state = _state & 1;
    return 1;
  } else return 0;
}

// The received data handle
void UART_RBR_handle() {
  char data;

    // Read the data
  data = LPC_UART->RBR;

  // If we don't have a buffer to file, get one
  if (!_buffer) {
        if (data == 0x7E) {
            while(!ZigBee_buffer_get_buffer(&_buffer))
                _state |= ZIGBEE_RECEIVER_RECEIVING;
        } else {
                return;
        }
    }
```

```

  switch (data) {
  // Started new frame
  case 0x7E:
    _size = 0;
    _offset = 0;
    _escape = 0;
        _checksum = 0;
    break;

  // escape next character
  case 0x7D:
    _escape = 0x20;
    break;

  // Nothing special, just add the byte
  default:
    {
        char t = _buffer[_offset++] = data ^ _escape;
        _checksum += (_offset>2)? t: 0;
    _escape = 0;
    }
  }

  // We now have the size delimiter, so read it
  if (_offset == 2) {
    _size = (_buffer[0]<<8) + _buffer[1];

  // If we have received the complete message, put it in the hold buffer
  } else if (_offset == _size + 3) {
    char* b = _buffer;
    _b_buffer = _buffer;
    _b_size = _offset;
    _state = ZIGBEE_RECEIVER_MESSAGE_READY;
    _buffer = 0;
    _offset = 0;
    if ((_checksum & 0xFF) == 0xFF) ZigBee_receiver_new_message_handle();
    else ZigBee_buffer_release_buffer(b);
        _checksum = 0;
  }
}
```

## B-5 ZigBee_sender

**Listing B.9:** ZigBee_sender.h

```
/*
 * ZigBee_sender.h
 *
 *  Created on: 9 mei 2012
 *      Author: Jan Angevare
 */

#ifndef ZIGBEE_SENDER_H_
#define ZIGBEE_SENDER_H_
```

```

#include "UART.h"
#include "ZigBee_buffer.h"

// Initialize ZigBee_sender
void  ZigBee_sender_init(void);
// Request send data, returns false if ZigBee_sender could not comply
// returns true if ZigBee sender is going to transmit
// transmits asynchronously
// No need to add 0x7E to buffer, this function automatically starts with 7E
// Characters which need to be escaped are automatically escaped
int   ZigBee_sender_send_frame(char* data, int size);
int   ZigBee_sender_abort(void);

int   ZigBee_sender_get_state(void);
// Define this function in your file
// This function is called when the transmit is completed
// the data parameter equals the data parameter form the send_frame function
// This function is called asynchronously
void  ZigBee_sender_send_complete_handle(char* data);

enum ZIGBEE_SENDER_STATES {
  ZIGBEE_SENDER_IDLE = 0,
  ZIGBEE_SENDER_SENDING = 1
};

#endif /* ZIGBEE_SENDER_H_ */
```

**Listing B.10:** ZigBee_sender.c

```
/*
 * ZigBee_sender.c
 *
 *  Created on: 9 mei 2012
 *      Author: bap
 */
#include "ZigBee_sender.h"

static char*  _buffer;
static int    _size;
static int    _offset;
static int    _escape;

volatile static int _state;

// Initialize state to zero
void ZigBee_sender_init() {
  _buffer = 0;
  _size = 0;
  _offset = 0;
  _escape = 0;
  _state = ZIGBEE_SENDER_IDLE;
}

// Request send frame
int ZigBee_sender_send_frame(char* data, int size) {
  // If a transmit request is in progress
```

```
  // return false
  if (_buffer) return 0;
  _state = ZIGBEE_SENDER_SENDING;

  // Set variables
  _buffer = data;
  _size = size;
  _offset = 0;
  _escape = 0;

  // Write new frame
  UART_write(0x7E);

  // return true
  return 1;
}

int ZigBee_sender_abort() {
  char* buffer = _buffer;

  _buffer = 0;
  _size = 0;
  _offset = 0;
  _escape = 0;
  _state = ZIGBEE_SENDER_IDLE;

  ZigBee_buffer_release_buffer(buffer);

  return 0;
}

int ZigBee_sender_get_state() {
  return _state;
}
// Event handle for Transmit Hold Register Empty interrupt
void UART_THRE_handle() {
  // Nothing to do here
  if (!_buffer) return;


  // If escaped and escape 0x7D sent
  if (_escape) {
    LPC_UART->THR = _buffer[_offset++] ^ 0x20;
    _escape = 0;


  // If not escaped sent next char
  } else {
    char data = _buffer[_offset];
    switch (data) {

    // If next char needs to be escaped
    case 0x7E:
    case 0x7D:
    case 0x11:
    case 0x13:
      LPC_UART->THR = 0x7D;
```

```
      _escape = 1;
      break;

    // Normal, just sent
    default:
      LPC_UART->THR = data;
      _offset++;
    }
  }

  // Sent all data
  if (_offset == _size) {

    // Call send_complete handler
    char* buffer = _buffer;
    _buffer = 0;
    _size = 0;
    _offset = 0;
    _state = ZIGBEE_SENDER_IDLE;
    ZigBee_sender_send_complete_handle(buffer);

  }
}
```

## B-6 ZigBee_translator

**Listing B.11:** ZigBee_translator.h

```
/*
 * ZigBee_translator.h
 *
 *  Created on: 9 mei 2012
 *      Author: Jan Angevare
 */

#ifndef ZIGBEE_TRANSLATOR_H_
#define ZIGBEE_TRANSLATOR_H_

// Translates the message and calls the specified handle function
int   ZigBee_translator_translate(char* message, int size);

// These functions get called by the ZigBee translator whenever it
// is translating the handles message
void  ZigBee_translator_send_success_handle(int data_frame);
void  ZigBee_translator_send_failure_handle(int data_frame);
void  ZigBee_translator_receive_data_handle(char* data, int size);
void  ZigBee_translator_associated(void);
void  ZigBee_translator_disassociated(void);

enum ZIGBEE_FRAME_ID {
  // Sent from ZigBee in specific conditions:
  ZIGBEE_MODEM_STATUS = 0x8A,

  // Allows for module parameter registers to be
  // queried or set
```

```
  ZIGBEE_AT_COMMAND = 0x08,

  // Same as AT_COMMAND but parameters not applied
  // until an AT_COMMAND or an APPLY_CHANGES
  ZIGBEE_AT_COMMAND_QUEUE = 0x09,

  // Sent from ZigBee, in response from AT_COMMAND
  ZIGBEE_AT_COMMAND_RESPONSE = 0x88,
  ZIGBEE_REMOTE_COMMAND_REQUEST = 0x17,

  // Let ZigBee send data
  ZIGBEE_TRANSMIT_REQUEST = 0x10,

  // Same as transmit but expl. adr.
  ZIGBEE_EXPLICIT_ADRRESSING = 0x11,

  // When a TX request is completed, the module
  // sends a TX message
  ZIGBEE_TRANSMIT_STATUS = 0x8B,

  // When the ZigBee receive an packet it sends
  // it to the UART with this message
  ZIGBEE_RECEIVE_PACKET = 0x90,

  // Received expl. adr. package
  ZIGBEE_EXPLICIT_RX = 0x91,


  ZIGBEE_SENSOR_READ = 0x94,
  ZIGBEE_NODE_IDENTIFICATION = 0x95
};

#endif /* ZIGBEE_TRANSLATOR_H_ */
```

**Listing B.12:** ZigBee_translator.c

```
/*
 * ZigBee_translator.c
 *
 *  Created on: 9 mei 2012
 *      Author: bap
 */

#include "ZigBee_translator.h"

int ZigBee_translator_translate(char* message, int size) {
  switch (message[2]) {

  // Only in debug mode use printf to print modem status
  case ZIGBEE_MODEM_STATUS:
    switch (message[3]) {

    case 2: // Associated
      ZigBee_translator_associated();
      break;

    case 3: // Disassociated
```

```
      ZigBee_translator_disassociated();
      break;

    case 0: // Hardware reset
    case 1: // Watchdog timer reset
    case 4: // Synchronization lost
    case 5: // Coordinator Realignment
    case 6: // Coordinator Started
    default:
      break;
    }
    break;


  case ZIGBEE_AT_COMMAND_RESPONSE:
    // If ZigBee Associated Indication Response
    if (message[4] == 'A' && message[5] == 'I') {

      // If no error
      if (message[6] == 0) {

        // If associated
        if (message[7] == 0) {
          ZigBee_translator_associated();
        // If not associated
        } else {
          ZigBee_translator_disassociated();
        }
      }
    }
    break;


  case ZIGBEE_TRANSMIT_STATUS:
    switch (message[7]) {
    case 0x00: //Success
      ZigBee_translator_send_success_handle((int)message[3]);
      break;

    case 0x22: //Not joined to network
            ZigBee_translator_disassociated();
            // continue
    case 0x02: //CCA Failure
    case 0x15: //Invalid destination endpoint
    case 0x21: //Network ACK failure
    case 0x23: //Self-addressed
    case 0x24: //Address Not Found
    case 0x25: //Route Not found
    default:
      ZigBee_translator_send_failure_handle((int)message[3]);
      break;
    }
    break;

  case ZIGBEE_RECEIVE_PACKET:
    ZigBee_translator_receive_data_handle(message + 14, size - 15);
    return 0;
```

```
    //break;

  default:
    break;
  }
  return 1;
}
```

## B-7 UART

**Listing B.13:** UART.h

```
/*
 * UART.h
 *
 *  Created on: 3 mei 2012
 *      Author: bap
 */
#include "LPC11xx.h"

#ifndef UART_H_INCLUDED
#define UART_H_INCLUDED

// Initialize the UART, after calling this
// function interrupts on the UART are turned on
void  UART_init(void);
// Write a char to the UART
int   UART_write(char);

// Get the UART state
int   UART_get_state(void);

// these functions get called by the UART
// after a char has been received and send
// respectively
void  UART_RBR_handle(void);
void  UART_THRE_handle(void);

enum UART_STATE {
  UART_receive_data = 0x01,
  UART_overrun_error = 0x02,
  UART_parity_error = 0x04,
  UART_framing_error = 0x08,
  UART_break_interrupt = 0x10,
  UART_transmit_hold_register_empty = 0x20,
  UART_transmiter_empty = 0x40,
  UART_receive_error = 0x80
};

enum RX_TRIGGER_LEVEL {
  BYTE_1 = 0x00,
  BYTES_4 = 0x40,
  BYTES_8 = 0x80,
  BYTES_14 = 0xC0
};
```

```
//const enum RX_TRIGGER_LEVEL TriggerLevel = BYTE_1;



#endif /* UART_H_ */
```

**Listing B.14:** UART.c

```
/*
 * UART.c
 *
 *  Created on: 3 mei 2012
 *      Author: bap
 */
#include "UART.h"

//extern const enum RX_TRIGGER_LEVEL TriggerLevel;
const enum RX_TRIGGER_LEVEL TriggerLevel = BYTE_1;

void UART_init() {
  // Initialize UART Con
  LPC_IOCON->PIO1_6 = 0xC1; // Set RXD
  LPC_IOCON->PIO1_7 = 0xC1; // Set TXD
  LPC_IOCON->PIO0_7 = 0xC1; // Set CTS
  LPC_IOCON->PIO1_5 = 0xC1; // Set RTS

  LPC_SYSCON->SYSAHBCLKCTRL |= 0x1000; //Turn on clock
  LPC_SYSCON->UARTCLKDIV = 1; //Set clock divider to 1
  LPC_UART->MCR = 0xC0; // Enable auto RTS and CTS
  LPC_UART->LCR |= 0x80; // Gain acces to DLL and DLM

  //LPC_UART->DLL = 71; // Set UART clock divider to 71  (baud = 9600)
  LPC_UART->DLL = 4; // Set UART clock divider to 3  (baud = 115200)

  LPC_UART->DLM = 0; // High byte
  LPC_UART->LCR = 0x03; // Stop acces to DL and set LCR in the meantime

  //LPC_UART->FDR = 0xA1; // Set prescaler to a good number (baud = 9600)
  LPC_UART->FDR = 0x85; // Set prescaler to a good number (baud = 115200)

  LPC_UART->IER = 0x03; // Data Receive Interupts enabled

  NVIC->ISER[0] |= (1<<21); //Enable UART interrupts

  // Clear FIFO Buffers and set interrupt trigger level
  LPC_UART->FCR = 0x07 | TriggerLevel;
}

// Write a char to the UART
int UART_write(char c) {
  // Wait until we can write a char to the transmit holde register
  while (!(LPC_UART->LSR & UART_transmit_hold_register_empty));

  // write it
  LPC_UART->THR = c;
  return 0;
```

```
}

int UART_get_state() {
  return LPC_UART->LSR;
}

void UART_IRQHandler (void) {
  int state = LPC_UART->IIR;
  state = LPC_UART->LSR;

  if (state & UART_receive_data)
    UART_RBR_handle();

  if (state & UART_transmit_hold_register_empty)
    UART_THRE_handle();
}
```

## 2-8 Power_modes

**Listing 2.15:** Power_modes.h

```
/*
 * power_modes.h
 *
 *  Created on: 22 mei 2012
 *      Author: bap
 */

#ifndef POWER_MODES_H_
#define POWER_MODES_H_

void power_modes_init(void);
// sleep for milliseconds, if interruptable this function will
// return on every interrupt, if not interruptable this function
// will only return when the timer expires.
// if the timer has expired this function return true
// if the timer has not expired this function returns false
// whenever milliseconds equals zero the timer is not reset
// but retains his value from last call
int  power_modes_sleep(int milliseconds, char interruptable);
void power_modes_deep_sleep(int timerValue);

#endif /* POWER_MODES_H_ */
```

**Listing 2.16:** Power_modes.c

```
/*
 * power_modes.c
 *
 *  Created on: 22 mei 2012
 *      Author: Ingmar Jager
 */
#include "power_modes.h"
#include <LPC11xx.h>
```

```c

void sampling_timer_init(void);
void wakeup_timer_init(int timerValue);


volatile int wakeup_clkctrl;
volatile int wakeup_nvic_0;
volatile int wakeup_nvic_1;

void power_modes_init() {
  // Enable the watchdog oscillator
  LPC_SYSCON->PDRUNCFG &= ~0x40;

  //Set watchdog oscillator frequency to 0.5MHz/64
  LPC_SYSCON->WDTOSCCTRL = 0x3F;


  // Setting timer for sleep
  LPC_SYSCON->SYSAHBCLKCTRL |= (1<<10); // Enable timer
  LPC_TMR32B1->PR = 12000; // Prescaler
  LPC_TMR32B1->MCR = 0x07; // Interrupt on MR0 and Stop on MR0, also TCR[0]
      reset
  LPC_SYSCON->SYSAHBCLKCTRL &= ~(1<<10); // Disable timer
  NVIC->ISER[0] |= (1<<19); // enable timer21b2 interrupt
}



int power_modes_sleep(int milliseconds, char interruptable) {
  // PCON DPDEN zero
  LPC_PMU->PCON &= ~0x02;

  // SCR SLEEPDEEP = 0;
  SCB->SCR &= ~0x04;

  // Check if we need to reset the timer
  if (milliseconds) {
    // Enable timer
    LPC_SYSCON->SYSAHBCLKCTRL |= (1<<10);

    // Reset timer
    LPC_TMR32B1->TCR = 0x02;

    // Set timeout value
    LPC_TMR32B1->MR0 = milliseconds;

    // Timer control = Timer Counter + Prescale Counter enabled
    LPC_TMR32B1->TCR = 0x01;

  } else {
    if (!(LPC_TMR32B1->TCR & 1)) return 0;
  }

  // Sleep
  do {
    __WFI();
  } while ((LPC_TMR32B1->TCR & 1) && !interruptable);
```

```


  return (LPC_TMR32B1->TCR & 1);
}

void TIMER32_1_IRQHandler() {
    LPC_TMR32B1->IR = 1;
  NVIC_ClearPendingIRQ(TIMER_32_1_IRQn);

  // Disable timer
  LPC_SYSCON->SYSAHBCLKCTRL &= ~(1<<10);
}

void power_modes_deep_sleep(int timerValue) {
    // WAKEUP CONFIG

    // Configure PDAWAKECFG to restore PDRUNCFG on wake up
    LPC_SYSCON->PDAWAKECFG = LPC_SYSCON->PDRUNCFG;

    // Store current value of SYSAHBCLKCTRL for restoration later
    wakeup_clkctrl = LPC_SYSCON->SYSAHBCLKCTRL;

  // Store current value of interrupt registers
  wakeup_nvic_0 = NVIC->ICER[0];
  wakeup_nvic_1 = NVIC->ICER[1];

  // Disable all interrupts
  NVIC->ICER[0] = 0xFFFFFFFF;
  NVIC->ICER[1] = 0xFFFFFFFF;


    // WAKEUP TIMER
    //init timer: select timer32MAT3 mode
    LPC_IOCON->R_PIO0_11 = 0xC3;
  LPC_GPIO0->DIR |= (1 << 11);

  // Enable clock to the timer
    LPC_SYSCON->SYSAHBCLKCTRL |= 0x200;

  // Configure timer

  // No interrupts
    LPC_TMR32B0->IR = 0x00;

  // Disable and reset timer
    LPC_TMR32B0->TCR = 0x2;

  // No prescaler
    LPC_TMR32B0->PR = 0;

  // Let the timer stop automatically when it matches
    LPC_TMR32B0->MCR = 0x800;

  // Set match register to timerValue
    LPC_TMR32B0->MR3 = (int)(500000.0 / 64.0) * timerValue;

  // No PWM functionality used
```

```
    LPC_TMR32B0->PWMC = 0x00;

  // Set external match thing to make the pin high at first and then drive the
      pin low upon match 3
    LPC_TMR32B0->EMR = 0x408;


  // WORKAROUND FOR RESET CURRENT CONSUMPTION

  // Make reset pin GPIO
  LPC_IOCON->RESET_PIO0_0 |= 0x01;

  // Make reset pin an output
  LPC_GPIO0->DIR |= 0x001;

  // Drive reset pin high
  LPC_GPIO0->DATA |= 0x001;


    // START LOGIC

    // Falling edge
    LPC_SYSCON->STARTAPRP0 &= ~(1 << 11);

    // Clear pending bit
    LPC_SYSCON->STARTRSRP0CLR = 1 << 11;

    // Enable Start Logic
    LPC_SYSCON->STARTERP0 |= 1 << 11;

    // Enable wakeup interrupt
    NVIC_EnableIRQ(WAKEUP11_IRQn);


    // GO TO SLEEP NOW

    // Shut down clocks to almost everything
    LPC_SYSCON->SYSAHBCLKCTRL = 0x215;

    // Configure deep sleep with WDT oscillator
    LPC_SYSCON->PDSLEEPCFG = 0x18BF;

    // Set SLEEPDEEP
    SCB->SCR |= 0x04;

    // Switch main clock to low-speed WDO
    LPC_SYSCON->MAINCLKSEL = 0x02;
    LPC_SYSCON->MAINCLKUEN = 0;
    LPC_SYSCON->MAINCLKUEN = 1;

  // Start the timer
    LPC_TMR32B0->TCR = 0x1;

    // Preload clock selection for quick switch back to XTAL on wakeup
    LPC_SYSCON->MAINCLKUEN = 0;
    LPC_SYSCON->MAINCLKSEL = 0x01;

```

```
    // Enter deep sleep mode
    __WFI();

    // Restore main clock to IRC 12 MHz
    LPC_SYSCON->MAINCLKUEN = 1;

    // Clear match bit on timer
    //LPC_TMR16B0->EMR = 0;

    // Clear pending bit on start logic
    LPC_SYSCON->STARTRSRP0CLR = 1 << 11;

    // Restore clocks to chip modules
    LPC_SYSCON->SYSAHBCLKCTRL = wakeup_clkctrl;

    // Restore interrupts
    NVIC_DisableIRQ(WAKEUP11_IRQn);
  NVIC_ClearPendingIRQ(WAKEUP11_IRQn);
  NVIC->ISER[0] = wakeup_nvic_0;
  NVIC->ISER[1] = wakeup_nvic_1;
}

void WAKEUP_IRQHandler(void) {
    // Clear pending bit on start logic

    // Disable start logic
    LPC_SYSCON->STARTERP0 &= ~(1 << 11);
    LPC_SYSCON->STARTRSRP0CLR = 1 << 11;

  // Disable start logic interrupt
    NVIC_DisableIRQ(WAKEUP11_IRQn);
  NVIC_ClearPendingIRQ(WAKEUP11_IRQn);
}
```

## 2-9 MIST

**Listing 2.17:** mist.h

```
// Project: WICS
// Author:  Jeroen van Straten
// Date:    20120503
// Purpose: High level access to MIST chip.

#ifndef mist_guard
#define mist_guard

#include <stdint.h>


// Struct containing the measurement result
__packed struct measurement {
    uint16_t temperature;
    uint16_t humidity;
    uint16_t ambience;
  uint8_t error;
```

```
};

#define ERRFLAG_BAT_MASK        0x07
#define ERRFLAG_MIST_ERROR        0x08
#define ERRFLAG_NOT_HARVESTING      0x10
#define ERRFLAG_INVALID_LIGHT       0x20
#define ERRFLAG_INVALID_HUMIDITY   0x40
#define ERRFLAG_INVALID_TEMPERATURE 0x80


// Initialize I/O and driver.
extern void initializeSensor(void);

// Takes a sample and writes this to data. Blocking!
extern uint8_t getMeasurement(struct measurement *data);



#endif
```

**Listing 2.18:** mist.c

```
#include "mist.h"

#include "LPC11xx.h"
#include <stdint.h>

#include "mist_lowlevel.h"
#include "mist_definitions.h"
#include "power_modes.h"
//#include "main.h"





uint8_t mist_enable(struct measurement *data);
void mist_read_temperature(struct measurement *data);
void mist_read_humidity(struct measurement *data);
void mist_read_light(struct measurement *data);
void mist_disable(struct measurement *data);
void adc_read_battery_and_pvcell(struct measurement *data);


// Sends a command, returns the reply code. The number of data words sent are
    defined by
// data_count (which must thus be 0, 1 or 2).
uint16_t mist_send_command(uint16_t command, uint16_t data1, uint16_t data2,
    uint8_t data_count);

// Receives a response code
uint16_t mist_receive_response(void);


volatile uint16_t debug;

void initializeSensor() {
```

```
    // Initialize I/O ports
    mist_ll_init();
}

uint8_t analog_trim_value;

uint8_t getMeasurement(struct measurement *data) {

  // Reset measurement data
  data->temperature = 0;
    data->humidity = 0;
    data->ambience = 0;
  data->error = ERRFLAG_INVALID_TEMPERATURE | ERRFLAG_INVALID_HUMIDITY |
      ERRFLAG_INVALID_LIGHT;

  // Read MIST stuff
  if (mist_enable(data)) {
    mist_read_temperature(data);
    mist_read_humidity(data);
    mist_read_light(data);
  }
  mist_disable(data);

  // Read battery stuff
  adc_read_battery_and_pvcell(data);

  return 1;
}



void adc_read_battery_and_pvcell(struct measurement *data) {
  // TODO
}


uint8_t mist_enable(struct measurement *data) {

    volatile uint32_t i;
    uint8_t analog_trim_value;

    // enable and init clock and SPI
    mist_clock_enable();
    mist_spi_enable();

    // reset
    mist_reset_assert();
    for (i = 200; i; i--); // just a short delay
    mist_reset_release();

    // wait for SINT (enable pulldown just for the first test to test
    // for disconnected sensor)
    LPC_IOCON->R_PIO1_0 |= 0x08; // enable SINT pulldown
    while (!mist_get_sint()); // TODO: timeout
    LPC_IOCON->R_PIO1_0 &= ~0x08; // disable the SINT pulldown

    // receive status word from the unit and save the factory calibration
```

```
    // value for the analog LDO
    analog_trim_value = (mist_receive_response() >> 8) & 0x0E;

    // enable analog LDO
    if (mist_send_command(POWER_LEVEL | analog_trim_value | POWER_LEVEL_LDO_ON, 0, 0, 0) != ACK) {
    data->error |= ERRFLAG_MIST_ERROR;
        return 0;
    }

  return 1;
}


void mist_read_temperature(struct measurement *data) {

    // power up the temperature sensor
    if (mist_send_command(SENSOR_POWER | SENSOR_TEMPERATURE, 0, 0, 0) != ACK) {
    data->error |= ERRFLAG_MIST_ERROR;
        return;
    }

    // setup sensor
    if ((debug = mist_send_command(SENSOR_SETUP | SENSOR_TEMPERATURE, TEMP_OPMODE, TEMP_OPMODE_OM_CALI | TEMP_OPMODE_RES_12B, 2)) != ACK) {
    data->error |= ERRFLAG_MIST_ERROR;
        return;
    }

    // start sensor read
    if (mist_send_command(SENSOR_START | SENSOR_TEMPERATURE, 0, 0, 0) != ACK) {
    data->error |= ERRFLAG_MIST_ERROR;
        return;
    }

    // wait for sensor to be done
    while (!(mist_send_command(STATUS, 0, 0, 0) & STATUS_TEMP_IRQ)) power_modes_sleep(10, 0);

    // fetch result
    data->temperature = mist_send_command(SENSOR_READ_OUTPUT | SENSOR_TEMPERATURE, 0, 0, 0);
  data->error &= ~ERRFLAG_INVALID_TEMPERATURE;

    // turn off sensor again
    mist_send_command(SENSOR_SHUTDOWN | SENSOR_TEMPERATURE, 0, 0, 0);

}


void mist_read_humidity(struct measurement *data) {

    __fp16 half;

    // power up the humidity sensor
    if (mist_send_command(SENSOR_POWER | SENSOR_HUMIDITY, 0, 0, 0) != ACK) {
    data->error |= ERRFLAG_MIST_ERROR;
```

```
            return;
        }

        // sensor opmode
        if ((debug = mist_send_command(SENSOR_SETUP | SENSOR_HUMIDITY, HUMID_OPMODE, HUMID_OPMODE_VAL, 2)) != ACK) {
        data->error |= ERRFLAG_MIST_ERROR;
            return;
        }

        if ((debug = mist_send_command(SENSOR_SETUP | SENSOR_HUMIDITY, HUMID_INPUT_SEL, HUMID_INPUT_SEL_1, 2)) != ACK) {
        data->error |= ERRFLAG_MIST_ERROR;
            return;
        }

        // sensor calibration
        if ((debug = mist_send_command(SENSOR_SETUP | SENSOR_HUMIDITY, HUMID_CAL_A0, mist_send_command(NVMEM_READ | RH_1_A0, 0, 0, 0), 2)) != ACK) {
        data->error |= ERRFLAG_MIST_ERROR;
            return;
        }

        if ((debug = mist_send_command(SENSOR_SETUP | SENSOR_HUMIDITY, HUMID_CAL_A1, mist_send_command(NVMEM_READ | RH_1_A1, 0, 0, 0), 2)) != ACK) {
        data->error |= ERRFLAG_MIST_ERROR;
            return;
        }

        if ((debug = mist_send_command(SENSOR_SETUP | SENSOR_HUMIDITY, HUMID_CAL_B0, mist_send_command(NVMEM_READ | RH_1_B0, 0, 0, 0), 2)) != ACK) {
        data->error |= ERRFLAG_MIST_ERROR;
            return;
        }

        if ((debug = mist_send_command(SENSOR_SETUP | SENSOR_HUMIDITY, HUMID_CAL_B1, mist_send_command(NVMEM_READ | RH_1_B1, 0, 0, 0), 2)) != ACK) {
        data->error |= ERRFLAG_MIST_ERROR;
            return;
        }

        half = (__fp16)((float)data->temperature * 0.015625f);
        if ((debug = mist_send_command(SENSOR_SETUP | SENSOR_HUMIDITY, HUMID_TEMP, *(uint16_t*) &half, 2)) != ACK) {
        data->error |= ERRFLAG_MIST_ERROR;
            return;
        }

        // start sensor read
        if (mist_send_command(SENSOR_START | SENSOR_HUMIDITY, 0, 0, 0) != ACK) {
        data->error |= ERRFLAG_MIST_ERROR;
            return;
        }
```

```c

    // wait for sensor to be done
    while (!(mist_send_command(STATUS, 0, 0, 0) & STATUS_HUMID_IRQ))
        power_modes_sleep(10, 0);

    // fetch result
    data->humidity = mist_send_command(SENSOR_READ_OUTPUT | SENSOR_HUMIDITY,
        0, 0, 0);
  data->error &= ~ERRFLAG_INVALID_HUMIDITY;

    // turn off sensor again
    mist_send_command(SENSOR_SHUTDOWN | SENSOR_HUMIDITY, 0, 0, 0);

}


void mist_read_light(struct measurement *data) {
  // TODO: not yet implemented
}


void mist_disable(struct measurement *data) {

    // disable analog LDO
    mist_send_command(POWER_LEVEL | analog_trim_value | POWER_LEVEL_LDO_OFF,
        0, 0, 0);

    // disable peripherals to reduce power consumption
    mist_reset_assert();
    mist_clock_disable();
    mist_spi_disable();

}




uint16_t mist_send_command(uint16_t command, uint16_t data1, uint16_t data2,
    uint8_t data_count) {
    volatile uint32_t delay;

    // Make sure no reply was pending still
    if (mist_get_sint()) {
        mist_receive_response();
    }

    // Something is wrong if the command is still pending, return ERR.
    if (mist_get_sint()) {
        return ERR;
    }

    // Select MIST chip
    mist_spi_select();

    // Give the chip a little time to get ready
    for (delay = 20; delay; delay--);

```

```
    // Send command
    mist_spi_transfer(command);
    if (data_count > 0) mist_spi_transfer(data1);
    if (data_count > 1) mist_spi_transfer(data2);

    // Deselect the MIST chip again
    mist_spi_deselect();

    // Wait for the MIST chip to get its data ready
    while (!mist_get_sint());

    // Wait for reply and return it.
    return mist_receive_response();
}


uint16_t mist_receive_response() {
    volatile uint32_t delay;
    uint16_t response;

    // Select MIST chip
    mist_spi_select();

    // Give the chip a little time to get ready
    for (delay = 20; delay; delay--);

    // Retrieve the response
    response = mist_spi_transfer(0x0000);

    // Deselect the MIST chip again
    mist_spi_deselect();

    return response;
}
```

**Listing 2.19:** mist_definitions.h

```
// Project: WICS
// Author:  Jeroen van Straten
// Date:    20120503
// Purpose: Defines a bunch of constants such as register numbers
//          and command IDs used by the MIST chip.

#ifndef mist_defs_guard
#define mist_defs_guard


// Return codes
#define ERR                 0xA1A1
#define ILLEGAL             0x5555
#define ACK                 0xAC00

// Command codes
#define RESET               0x0000
#define POWER_LEVEL         0x0100
#define STATUS              0x0200
#define IC_LOCK             0xC300
```

```
#define IC_UNLOCK             0xC400
#define SENSOR_POWER          0x0500
#define SENSOR_SHUTDOWN       0x0600
#define SENSOR_START          0x0700
#define SENSOR_SETUP          0xC800
#define SENSOR_READ_OUTPUT    0x0900
#define SENSOR_STATUS         0x0A00
#define NVMEM_UNLOCK_BANK     0xD000
#define NVMEM_LOCK_BANK       0xD200
#define NVMEM_SET_PASSWORD    0xD300
#define NVMEM_READ            0x1500
#define NVMEM_WRITE           0x5400

// Power level parameters
#define POWER_LEVEL_LDO_ON    0x0000
#define POWER_LEVEL_LDO_OFF   0x0001

// Sensor identifiers
#define SENSOR_TEMPERATURE    0x0001
#define SENSOR_HUMIDITY       0x0002
#define SENSOR_LIGHT          0x0003
#define SENSOR_ADC            0x0004

// Status masks
#define STATUS_TEMP_IRQ       0x1000
#define STATUS_HUMID_IRQ      0x2000
#define STATUS_LIGHT_IRQ      0x4000
#define STATUS_ADC_IRQ        0x8000

// Temperature sensor registers
#define TEMP_OPMODE           0x0020
#define TEMP_CAL_A            0x0021
#define TEMP_CAL_B            0x0022
#define TEMP_CAL_ALPHA        0x0023

// Temperature sensor opmode register values
#define TEMP_OPMODE_OM_RAW    0x0000
#define TEMP_OPMODE_OM_CALI   0x0001

#define TEMP_OPMODE_RES_5B    0x0000
#define TEMP_OPMODE_RES_6B    0x0002
#define TEMP_OPMODE_RES_7B    0x0004
#define TEMP_OPMODE_RES_8B    0x0006
#define TEMP_OPMODE_RES_9B    0x0008
#define TEMP_OPMODE_RES_10B   0x000A
#define TEMP_OPMODE_RES_11B   0x000C
#define TEMP_OPMODE_RES_12B   0x000E

#define TEMP_OPMODE_VCAL      0x0040

// Humidity sensor registers
#define HUMID_OPMODE          0x0020
#define HUMID_CAL_A0          0x0021
#define HUMID_CAL_A1          0x0022
#define HUMID_CAL_B0          0x0023
#define HUMID_CAL_B1          0x0024
#define HUMID_TEMP            0x0025
```

```
#define HUMID_INPUT_SEL      0x0026

// Humidity sensor opmode register values - only 1 value is supported
#define HUMID_OPMODE_VAL     0xEC07

// Humidity sensor input selection register values
#define HUMID_INPUT_SEL_1    0x0001
#define HUMID_INPUT_SEL_2    0x0002
#define HUMID_INPUT_SEL_3    0x0004
#define HUMID_INPUT_SEL_4    0x0008

// NVRAM calibration addresses
#define T_A                  0x0040
#define T_B                  0x0041
#define T_ALPHA              0x0042
#define RH_1_A0              0x0044
#define RH_1_A1              0x0045
#define RH_1_B0              0x0046
#define RH_1_B1              0x0047
#define RH_2_A0              0x0048
#define RH_2_A1              0x0049
#define RH_2_B0              0x004A
#define RH_2_B1              0x004B
#define RH_3_A0              0x004C
#define RH_3_A1              0x004D
#define RH_3_B0              0x004E
#define RH_3_B1              0x004F
#define RH_4_A0              0x0050
#define RH_4_A1              0x0051
#define RH_4_B0              0x0052
#define RH_4_B1              0x0053
#define AL_1_R_N             0x0054
#define AL_1_R_M             0x0055
#define AL_1_R_MIR           0x0056
#define AL_1_RB_N            0x0057
#define AL_1_RB_M            0x0058
#define AL_1_RB_MIR          0x0059
#define AL_2_R_N             0x005A
#define AL_2_R_M             0x005B
#define AL_2_R_MIR           0x005C
#define AL_2_RB_N            0x005D
#define AL_2_RB_M            0x005E
#define AL_2_RB_MIR          0x005F
#define AL_3_R_N             0x0060
#define AL_3_R_M             0x0061
#define AL_3_R_MIR           0x0062
#define AL_3_RB_N            0x0063
#define AL_3_RB_M            0x0064
#define AL_3_RB_MIR          0x0065
#define AL_4_R_N             0x0066
#define AL_4_R_M             0x0067
#define AL_4_R_MIR           0x0068
#define AL_4_RB_N            0x0069
#define AL_4_RB_M            0x006A
#define AL_4_RB_MIR          0x006B
#define AL_5_R_N             0x006C
#define AL_5_R_M             0x006D
```

```
#define AL_5_R_MIR          0x006E
#define AL_5_RB_N           0x006F
#define AL_5_RB_M           0x0070
#define AL_5_RB_MIR         0x0071
#define AL_6_R_N            0x0072
#define AL_6_R_M            0x0073
#define AL_6_R_MIR          0x0074
#define AL_6_RB_N           0x0075
#define AL_6_RB_M           0x0076
#define AL_6_RB_MIR         0x0077


#endif
```

**Listing 2.20:** mist_lowlevel.h

```
// Project: WICS
// Author:  Jeroen van Straten
// Date:    20120503
// Purpose: Low level access to MIST chip.

#ifndef mist_lowlevel_guard
#define mist_lowlevel_guard

#include <stdint.h>


// Initializes MIST chip I/O ports
extern void mist_ll_init(void);


/* SPI control */

// Enables and initializes the SPI peripheral.
extern void mist_spi_enable(void);

// Powers down the SPI peripheral.
extern void mist_spi_disable(void);

// Transfers a data word over the SPI bus.
extern uint16_t mist_spi_transfer(uint16_t data);

// Asserts /CSN (pulls it low).
extern void mist_spi_select(void);

// Releases /CSN (pulls it high).
extern void mist_spi_deselect(void);


/* Clock control */

// Enables and initializes the 1 MHz clock.
extern void mist_clock_enable(void);

// Powers down the 1 MHz clock.
extern void mist_clock_disable(void);

```

```

/* Reset control */

// Asserts /RESET (pulls it low).
extern void mist_reset_assert(void);

// Releases /RESET (pulls it high).
extern void mist_reset_release(void);


/* SINT */

// Returns the state of the SINT pin (nonzero if high, 0 if low)
extern uint8_t mist_get_sint(void);


/* DOUT */

// Returns the state of the DOUT pin (nonzero if high/stable, 0 if low)
extern uint8_t mist_is_reg_stable(void);


#endif
```

**Listing 2.21:** mist_lowlevel.c

```
#include "mist_lowlevel.h"

#include "LPC11xx.h"
#include <stdint.h>


void mist_ll_init() {
    // CLK pin (pin 0.11)
    LPC_IOCON->R_PIO0_11 = 0xC1; // set GPIO, no pullup
    LPC_GPIO0->DIR |= 0x800; // set pin as output

    // /CS pin (pin 0.2)
    LPC_IOCON->PIO0_2 = 0xC0; // set GPIO, no pullup
    LPC_GPIO0->DIR |= 0x004; // set output
    LPC_GPIO0->DATA |= 0x004; // set default high

    // MISO pin (pin 0.8)
    LPC_IOCON->PIO0_8 = 0xC1; // set MISO, no pullup

    // MOSI pin (pin 0.9)
    LPC_IOCON->PIO0_9 = 0xC1; // set MOSI, no pullup
    LPC_GPIO0->DIR |= 0x200; // set output

    // SCLK pin (pin 2.11)
    LPC_IOCON->SCK_LOC = 0x01; // set SCK on pin 2.11
    LPC_IOCON->PIO2_11 = 0xC1; // set SCLK, no pullup
    LPC_GPIO2->DIR |= 0x800; // set output

    // /RESET pin (pin 1.1)
    LPC_IOCON->R_PIO1_1 = 0xC1; // set GPIO, no pullup
    LPC_GPIO1->DIR |= 0x002; // set output
```

```

    // SINT pin (pin 1.0)
    LPC_IOCON->R_PIO1_0 = 0xC1; // set GPIO, no pullup

    // DOUT pin (pin 1.2)
    LPC_IOCON->R_PIO1_2 = 0xC1; // set GPIO, no pullup
}


void mist_spi_enable() {
    // enable SPI clock (set to 1 MHz input clock)
    LPC_SYSCON->SYSAHBCLKCTRL |= 0x800;
    LPC_SYSCON->SSP0CLKDIV = 12;

    // release SPI reset
    LPC_SYSCON->PRESETCTRL |= 0x01;

    // setup SPI
    LPC_SSP0->CR0 = 0x000F; // 16 bit, SPI mode 0, no further clock division
    LPC_SSP0->CR1 = 0x0002; // enable SPI
    LPC_SSP0->CPSR = 0x0010; // set prescaler to lowest acceptable value

}

void mist_spi_disable() {
    // assert SPI reset
    LPC_SYSCON->PRESETCTRL |= 0x01;

    // disable SPI clock
    LPC_SYSCON->SYSAHBCLKCTRL &= ~0x800;
}

uint16_t mist_spi_transfer(uint16_t data) {
    // ensure the SPI is turned on (otherwise this will block forever)
    if (!(LPC_SSP0->CR1 & 0x0002)) {
        return 0;
    }

    // start the transfer
    LPC_SSP0->DR = data;

    // wait for the SPI to finish the transfer (wait for busy and RX empty
    // to be released)
    while ((LPC_SSP0->SR & 0x10) && !(LPC_SSP0->SR & 0x04));

    // return received data
    return LPC_SSP0->DR;
}

extern void mist_spi_select() {
    // Set /CS pin low
    LPC_GPIO0->DATA &= ~0x004;
}

extern void mist_spi_deselect() {
    // Set /CS pin high
    LPC_GPIO0->DATA |= 0x004;
```

```
}


void mist_clock_enable() {
    // enable sensor clock timer
    LPC_SYSCON->SYSAHBCLKCTRL |= 0x200;

    // reset timer before setup
    LPC_TMR32B0->TCR = 0x02;

    // setup sensor clock timer
    LPC_TMR32B0->PR = 0; // no prescaler
    LPC_TMR32B0->MR0 = 11; // count to 12 for 1 MHz
    LPC_TMR32B0->MR3 = 6; // 50% duty cycle
    LPC_TMR32B0->MCR = 0x02; // reset on MR0
    LPC_TMR32B0->PWMC = 0x08; // enable PWM for MR3

    // sensor clock pin
    LPC_IOCON->R_PIO0_11 = 0xC3; // select timer output

    // start sensor clock timer
    LPC_TMR32B0->TCR = 0x01;
}

void mist_clock_disable() {
    // disable sensor clock timer
    LPC_SYSCON->SYSAHBCLKCTRL &= ~0x200;

    // set pin to GPIO mode    to ensure the output is defined
    LPC_IOCON->R_PIO0_11 = 0xC0;
}


void mist_reset_assert() {
    // assert reset (pull it low)
    LPC_GPIO1->DATA &= ~0x002;
}

void mist_reset_release() {
    // release reset (pull it high)
    LPC_GPIO1->DATA |= 0x002;
}


uint8_t mist_get_sint() {
    return LPC_GPIO1->DATA & 0x001;
}


uint8_t mist_is_reg_stable() {
    return LPC_GPIO1->DATA & 0x004;
}
```

## 2-10 Main loop

**Listing 2.22:** main.c

```c
/*
===============================================================================
 Name        : main.c
 Author      : Ingmar Jager & Jan Angevare
 Version     : 1.0.0
 Copyright   : $(copyright)
 Description : main definition
===============================================================================
*/


#include "LPC11xx.h"

#include "ZigBee.h"
#include "mist.h"
#include "power_modes.h"

__packed struct WICS_data {
  char  SequenceID;
  char  MeasurementInterval;
  __packed struct measurement Measurements[10];
};

int associate(void);
void wait_for_associated(void);

void SystemInit(void) {
  int i;

  // Set running mode for PDRUNCFG and SYSAHBCLKCTRL
  LPC_SYSCON->PDRUNCFG = 0xED88;
  LPC_SYSCON->SYSAHBCLKCTRL = 0x1305F;

  // Switch to XTAL
  for (i = 0; i < 200; i++) __NOP();

  LPC_SYSCON->SYSPLLCLKSEL = 0x01;
  LPC_SYSCON->SYSPLLCLKUEN = 0x00;
  LPC_SYSCON->SYSPLLCLKUEN = 0x01;

  LPC_SYSCON->MAINCLKSEL = 0x01;
  LPC_SYSCON->MAINCLKUEN = 0x00;
  LPC_SYSCON->MAINCLKUEN = 0x01;

  // Disable IRC
  LPC_SYSCON->PDRUNCFG |= 0x003;

  // Set complete pin config here
  LPC_IOCON->RESET_PIO0_0  = 0x0C0;
  LPC_IOCON->PIO0_1        = 0x0D0;
  LPC_IOCON->PIO0_2        = 0x0C1;
  LPC_IOCON->PIO0_3        = 0x0C0;
  LPC_IOCON->PIO0_4        = 0x1C0;
```

```
  LPC_IOCON->PIO0_5         = 0x1C0;
  LPC_IOCON->PIO0_6         = 0x0C0;
  LPC_IOCON->PIO0_7         = 0x0C1;
  LPC_IOCON->PIO0_8         = 0x0C1;
  LPC_IOCON->PIO0_9         = 0x0C1;
  LPC_IOCON->SWCLK_PIO0_10  = 0x0C0;
  LPC_IOCON->R_PIO0_11      = 0x0C3;

  LPC_IOCON->R_PIO1_0       = 0x0C1;
  LPC_IOCON->R_PIO1_1       = 0x0C1;
  LPC_IOCON->R_PIO1_2       = 0x0C1;
  LPC_IOCON->SWDIO_PIO1_3   = 0x0C0;
  LPC_IOCON->PIO1_4         = 0x041;
  LPC_IOCON->PIO1_5         = 0x0C1;
  LPC_IOCON->PIO1_6         = 0x0C1;
  LPC_IOCON->PIO1_7         = 0x0C1;
  LPC_IOCON->PIO1_8         = 0x0C0;
  LPC_IOCON->PIO1_9         = 0x0C0;
  LPC_IOCON->PIO1_10        = 0x0C0;
  LPC_IOCON->PIO1_11        = 0x041;

  LPC_IOCON->PIO2_0         = 0x0C0;
  LPC_IOCON->PIO2_1         = 0x0C0;
  LPC_IOCON->PIO2_2         = 0x0C0;
  LPC_IOCON->PIO2_3         = 0x0C0;
  LPC_IOCON->PIO2_4         = 0x0C0;
  LPC_IOCON->PIO2_5         = 0x0C0;
  LPC_IOCON->PIO2_6         = 0x0C0;
  LPC_IOCON->PIO2_7         = 0x0C0;
  LPC_IOCON->PIO2_8         = 0x0C0;
  LPC_IOCON->PIO2_9         = 0x0C0;
  LPC_IOCON->PIO2_10        = 0x0C0;
  LPC_IOCON->PIO2_11        = 0x0C1;

  LPC_IOCON->PIO3_0         = 0x0C0;
  LPC_IOCON->PIO3_1         = 0x0C0;
  LPC_IOCON->PIO3_2         = 0x0C0;
  LPC_IOCON->PIO3_3         = 0x0C0;
  LPC_IOCON->PIO3_4         = 0x0C0;
  LPC_IOCON->PIO3_5         = 0x0C0;

  LPC_GPIO0->DIR            = 0xA3C;
  LPC_GPIO0->DATA           = 0x004;

  LPC_GPIO1->DIR            = 0x7A2;
  LPC_GPIO1->DATA           = 0x0A0;

  LPC_GPIO2->DIR            = 0xFFF;
  LPC_GPIO2->DATA           = 0x000;

  LPC_GPIO3->DIR            = 0x03F;
  LPC_GPIO3->DATA           = 0x000;
}

int associate() {
  if (ZigBee_get_state()) {
    ZigBee_request_connection_status();
```

```
    power_modes_sleep(10000, 1);

    while (ZigBee_get_state())
      if (!power_modes_sleep(0, 1))
        break;

  }

  return !ZigBee_get_state();
}

void wait_for_associated() {
  if (ZigBee_get_state()) {
    ZigBee_wake_up();

    while (!associate()) {
      // Association is taking way too long, the coordinator is probably off.
      // Enter deep sleep for five minutes to save power while waiting for it
      // to turn on.
      ZigBee_set_sleep();
      //power_modes_deep_sleep(300);
      ZigBee_wake_up();
    }
  }
}

int main(void) {
    // Buffer handle
  char* data;
  int size;

  // Header data + sample count
  struct WICS_data measure_data;
  int i = 0;
  int sample_amount = 1;

  // Initialize
  LPC_GPIO3->DATA |= 0x008;
    LPC_GPIO2->DATA &= ~0x040;
  ZigBee_init();
  initializeSensor();
  power_modes_init();

  /* vOOR MEETING*/
  //LPC_GPIO2->DATA |= 0x040;
  //while(1) power_modes_deep_sleep(300);
/**/

  measure_data.MeasurementInterval = 1;
  measure_data.SequenceID = 0;

  power_modes_sleep(10, 0);

  i = 0;

  // Enter main-loop
  while (1) {
```

```

  // Enter out of order Loop (measurments are taken while ZigBee sends)
  while (1) {
    // Get measurement, this takes a while
    getMeasurement(&measure_data.Measurements[i]);

    // Ensure that we are still associated
    wait_for_associated();

    // Wait till time has transpired, this takes even longer
    ZigBee_set_sleep();
    power_modes_deep_sleep(measure_data.MeasurementInterval);

    // Check for new messages, if so reconfigure everything
    // Send the data we already got and reset count
    ZigBee_wake_up();
    if (ZigBee_check_for_new_message(&data, &size)) {

      // Send everything we've got
      //ZigBee_send((char*)&measure_data, 2 + 7 * (i + 1));

      ZigBee_send(data, size);

      // Reset count
      i = 0;
      measure_data.SequenceID++;

      // Reconfiguring
      //measure_data.MeasurementInterval = data[0];
      //sample_amount = (((data[1]>0)? data[1]: 1)<11)?data[1]:10;

      // Release buffer
      ZigBee_done_reading_new_message();

    // Only if no message was received will we update the count
    // and on enough samples send, otherwise the count will be
    // reset and no measurements will be in the buffer
    } else {
      i = (i + 1) % sample_amount;
      if (i == 0) {
            ZigBee_send((char*)&measure_data, 2 + 7 * sample_amount);
        measure_data.SequenceID++;
          } else {
            power_modes_sleep(3, 0);
            ZigBee_set_sleep();
          }
    }
  }

  //Enter in order Loop (first measurement then send)
  while (1) {
    // Get measurement, this takes a while
    getMeasurement(&measure_data.Measurements[i]);

    // Check for new messages, if so reconfigure everything
    // Send the data we already got and reset count
    ZigBee_wake_up();
```

```
      if (ZigBee_check_for_new_message(&data, &size)) {

        // Send everything we've got
        //ZigBee_send((char*)&measure_data, 2 + 7 * (i + 1));
        ZigBee_send(data, size);

        // Reset count
        i = 0;
        measure_data.SequenceID++;

        // Reconfiguring
        measure_data.MeasurementInterval = data[0];
        sample_amount = (((data[1]>0)? data[1]: 1)<11)?data[1]:10;

        // Release buffer
        ZigBee_done_reading_new_message();

      // Only if no message was received will we update the count
      // and on enough samples send, otherwise the count will be
      // reset and no measurements will be in the buffer
      } else {
        i = (i + 1) % sample_amount;
        measure_data.SequenceID++;
        if (i == 0) ZigBee_send((char*)&measure_data, 2 + 7 * sample_amount);
      }
    }

  }

  // Without this stuff won't compile
  //return 0 ;
}
```