Independent degree project – second cycle

Elektroteknik Ingenjör
Electrical Engineering

Wireless Sensor Systems
Development of a wireless sensor system for the characterization of energy harvesting conditions

Felix Hörnschemeyer
Wireless sensor system to characterize energy harvesting conditions

Felix Hörnschemeyer

MID SWEDEN UNIVERSITY
Department of Electronics Design
Examiner: Börje Norlin, borje.norlin@miun.se
Supervisor: Sebastian Bader, Sebastian.Bader@miun.se
Author: Felix Hörnschemeyer, feho1500@student.miun.se
Degree programme: Bachelor of Science, 180 credits
Main field of study: Electrical Engineering
Semester, year: Spring, 2016

2016-07-12
Abstract

This report deals with the development of a wireless sensor system that measures the environmental energy and predicts if energy harvesting could be possible in different areas. It provides an overview over the hardware used to build this system and gives a detailed description of the software implementation of the system. The hardware part presents the microcontroller and platform that is used, as well as the sensors integrated in the system. The software part explains how the used hardware was put together in a program that controls the different components. It explains the possibility to save captured sensor values on an SD card or send them to a remote receiver with an XBee radio module in real time. Also the inclusion of the mbed software library, which provides a lot of useful applications and functions for the project, is an important part. The final part of the report presents the results, showing how the system works.

Keywords: Sensor System, Energy Harvesting, Prediction, Wireless, XBee radio module, SD card, Luminosity Sensor, RGB Sensor, Accelerometer, C++, mbed library.
Acknowledgements

During this project I was assisted by a number of people who contributed to the completion of the project. I want to thank everyone who helped me during the project and the writing of my thesis. First of all I would like to thank my supervisor Sebastian Bader, who helped and guided me throughout the thesis work. Furthermore I want to thank Mid Sweden University and all the people and lecturers who provided me with new knowledge in lectures and projects during my stay in Sundsvall. Finally I would like to thank my flatmate, study college and friend André bei der Hake who worked together with me on all projects and lab sessions the past year and without whom I would not be graduating today.
# Table of Contents

Abstract ........................................................................................................ iv

Acknowledgements .................................................................................. v

Table of Contents ...................................................................................... vi

Terminology ................................................................................................. viii

Acronyms ..................................................................................................... viii

1 Introduction ............................................................................................. 1

1.1 Background and motivation ................................................................. 1

1.2 Related work ....................................................................................... 2

1.3 Overall aim .......................................................................................... 3

1.4 Outline .................................................................................................. 3

2 Methodology .............................................................................................. 4

2.1 System components ............................................................................ 4

2.1.1 Platform SentioEM3 Rev3.2 ............................................................ 4

2.1.2 Sensors .......................................................................................... 5

2.1.3 Radio module ................................................................................ 6

2.1.4 SD card module ............................................................................. 6

2.2 MBED platform .................................................................................. 6

2.3 Simplicity Studio and XCTU ............................................................... 8

3 Implementation ........................................................................................ 10

3.1 Import the ARM mbed library ............................................................ 10

3.2 Sensor implementation ..................................................................... 14

3.2.1 TSL2561 luminosity sensor ......................................................... 14

3.2.2 ISL29125 RGB light sensor ........................................................ 15

3.2.3 ADXL345 triple axis accelerometer sensor .............................. 16

3.3 XBeeS1 radio module implementation ............................................. 18

3.4 SD card implementation .................................................................. 21

3.5 Main file implementation ................................................................ 24

3.5.1 Timer controlled Interrupt implementation ............................. 24

3.5.2 Main function implementation .................................................... 25

4 Results ..................................................................................................... 28

4.1 Experimental Setup .......................................................................... 28

4.2 Measurements results ...................................................................... 28

4.3 Energy consumption ......................................................................... 31
5 Discussion and Conclusion .................................................................33
  5.1 Discussion part .................................................................33
  5.2 Conclusion part .................................................................33
  5.3 Future work .................................................................34
  5.4 Social and ethical aspects .................................................34

References.........................................................................................35

Appendix A: Documentation of own developed program code ..........36
  Main.cpp ......................................................................................36
  ADXL345.h ..................................................................................39
  ISL29125.h ..................................................................................41
  TSL2561.h ....................................................................................42
  XBee_send.h ..................................................................................44
  SD_Config.h ..................................................................................46
  Timer0_Config.h .............................................................................48
**Terminology**

This section provides a list of terms, abbreviations and variable names with brief explanations.

**Acronyms**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Advanced Risc Machines – most common microprocessor chip architecture</td>
</tr>
<tr>
<td>CMSIS</td>
<td>Cortex Microcontroller Software Interface Standard</td>
</tr>
<tr>
<td>FAT</td>
<td>File Allocation Table</td>
</tr>
<tr>
<td>GPIO</td>
<td>General-Purpose Input/Output</td>
</tr>
<tr>
<td>HAL</td>
<td>Hardware Abstraction Layer</td>
</tr>
<tr>
<td>I2C</td>
<td>Inter Integrated Circuit</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>MBED</td>
<td>Provided C++ library for the microcontroller</td>
</tr>
<tr>
<td>MCU</td>
<td>Micro Control Unit</td>
</tr>
<tr>
<td>MISO</td>
<td>Master Input Slave Output</td>
</tr>
<tr>
<td>MOSI</td>
<td>Master Output Slave Input</td>
</tr>
<tr>
<td>SD-card</td>
<td>Secure Digital Memory Card</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>SCLK</td>
<td>Serial Clock</td>
</tr>
<tr>
<td>SSEL</td>
<td>Slave Select</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
</tr>
<tr>
<td>XBee</td>
<td>Radio module sending data based on the IEEE 802.15.4 standard</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Background and motivation

The aim of this project is to develop an environmental sensor system able to measure the energy in different areas and show if there is enough energy that could be harvested to generate a power supply for other autonomous wireless sensor systems. There are already several wireless sensor networks (WSNs) using the power supply via energy harvesting, the problem is that none of them is able to predict if enough energy will be generated to supply the system for the desired time. What these systems rely on cannot be predicted before the system is in its working environment. This is connected to a high risk nobody would like to take. But the reason why energy harvesting is still used in these systems is obvious: to have nodes that do not need charging its battery every so often is an advantage when it comes to flexibility. The nodes can be placed in the measurement spot and remain there until the value capturing is complete. This problem of not knowing if there will be enough energy to supply the system with energy harvesting could be solved by measuring the energy in the different areas before installing an autonomous sensor system. A mapping of the area could lead to a prediction of where most energy could be harvested, how much energy could be harvested and which source could be the most efficient (vibrations, solar, etc.) These places can be located outside in areas with a lot of sunlight or wind, but also inside in industry halls where also vibration or light could be used to supply a wireless sensor system. What should such a system look like and what components should be integrated? The first step is to implement sensors that measure the values in the environment. The sensors have to send the values to a microcontroller, which processes the data. This processing could either involve storing the values somewhere or sending them in real-time to another remote system that stores all the sensor values coming in from all the nodes in the whole system. Thus the system has to consist of more than one node in order to be able to measure in larger areas (mapping) and make a more detailed statement about energy harvesting possibilities. Also every single node should be able to make a system on its own to have a varying number of nodes in one system and become more flexible. To provide even more flexibility, the nodes should, like mentioned
earlier, be able to save the values on for example an SD card or send them to a remote receiver via a radio module. This provides the possibility to also make measurements over longer periods of time when, for example, a receiving station cannot be online all the time. In this case the values are stored on an SD card and can be processed later. Also because the system itself should work wireless in order to place the nodes randomly in areas and not be limited by cables, a node needs to be energy efficient due to the fact that its lifetime is regulated by a small portable battery. The possibility of supplying the nodes with harvested energy does not make sense because the system should predict if it is possible to have an autonomous wireless sensor system in this place. Combining these facts, a measurement system that is able to predict where to harvest energy can reduce the risk of running out of energy using an energy harvest supplied system and therefore creating more flexibility in terms of the use of these systems.

1.2 Related work
Creating a wireless sensor system that measures the environment using different kind of sensors is nothing new. What is new is that it measures in the field of energy harvesting and is able to save the captured values on an SD card and/or send the data via a radio module. Most Wireless Sensor Networks do not use energy harvesting to measure its values but to use it in order to supply them. Other systems like the one from the University of Linz explained in [1] use the harvested energy for its own energy supply, which makes measurements more flexible, if the environment already provides enough energy sources like sunlight to guarantee a continuous measurement. Also, the system described in [2], created by scientists from the University of Milano, is using energy harvesting in form of solar power to supply the sensor system. Their goal was to make the system as energy efficient as possible, because a prediction of the available energy cannot be made. In [3], the ‘PROfile energy prediction model’ is presented by scientists from the University of Rome. This research focused on building a model or simulation to predict the energy that could be harvested in different areas. Their algorithms were able to increase the prediction accuracy to 60%. However this is also far from an unerring statement about the energy that could be harvested in specific areas, it is necessary to develop a sensor system that is able to make more accurate statements about the prediction of energy harvesting. By that autonomous systems can be placed in
the predicted spots without risking to run out of energy during the measurements.

1.3 Overall aim

The main aim is to develop a running measurement system able to save the data from different sensors on an SD card, or to send data to a remote receiver in real-time. As a first step towards this aim, one node will be developed, with the ability to measure and as a result stand as a small independent system, before combining more nodes into a bigger system. That aim could be divided in several steps. The first step is to develop a running platform with the mbed library included in the project. When the mbed library is included it should be possible to setup the three sensors for RGB light, luminosity and acceleration. After the sensors have been applied to the system, the radio module is included in order to send the captured values to a remote receiver. If data can be sent, an option should be to save the data on an SD card. In other words, the main aim focuses on developing software for the node and going through the steps mentioned should lead up to this aim.

1.4 Outline

Chapter 2 describes the method of the system and the different hardware components included in the project. Also the software tools used during the programing are briefly explained. Chapter 3 explains the software program and how the hardware is implemented to create one system. In Chapter 4 results are presented showing how the system works. Chapter 5 presents the discussion and conclusion of the project.
2  Methodology

2.1  System components

This chapter will provide an overview of the hardware and software used in the project. The sections on hardware describe single system components such as the microcontroller used, the sensors and radio as well as SD card modules, while the software section deals with the IDE and software tools used, which enable the configuration of the ‘XBee’ radio module.

2.1.1  Platform SentioEM3 Rev3.2

For this project, the ‘SentioEM3’, developed by the Department of Electronics Design at Mid Sweden University, was selected. The platform works with an EFM32 microcontroller based on the ARM-Cortex M3 architecture. The name of the microcontroller is EFM32GG380F1024. The following figure shows the hardware architecture:

![Figure 1 Block diagram of Sentio-em hardware components](image)
As the diagram shows, the platform provides a RF transceiver and a micro SD card module. The ability to use this two communication modules were decisive for its choice for the project. Also the low power consumption of the Cortex M3 processor supports the choice.

2.1.2 Sensors

To measure light and vibrations in the environment, three different sensors are used in this project. All sensors come from the manufacturer ‘SparkFun’. The first sensor is the ‘ISL29125 RGB’ light sensor, which can measure the strength of the three RGB colors green, blue and red of the incoming visible light. The output is returned in a 16 bit value for each of the three colors.

Second, the ‘TSL 2561’ luminosity sensor is used to measure the intensity of the light. Its output returns two 16-bit numbers that contain the values for visible light and visible light plus infrared light. Because the ‘TSL2651’ is an integrating sensor, it is possible to measure different amounts of light by adjusting it.

The reason it is important to have RGB values in addition to the visible light and the infrared light is to be able to tell how much energy is in the light coming in. For example, blue light has more energy than the red light. So, to calculate the solar energy in the environment it is also important to include the color of the light.

The third sensor is the ‘ADXL345’ triple axis accelerometer, which is used to measure the acceleration in the x, y and z direction. It has a 13-bit resolution making it possible to send up to +16g. Furthermore it is possible to measure both static and dynamic acceleration because of its 4mg/LBS resolution. The output is the same as for the other sensors, a 16-bit value for x, y and z.

All three sensors communicate via I2C and have their own static hardware addresses. Figure 2 below provides further overview of the sensors used and their characteristics, such as their operating voltage, operating current and ADC resolution.
The reason why these sensors were used for the project is that they have been selected for other projects at Mid Sweden University, which indicates that there are already comparative values.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>ISL29125</th>
<th>TSL2561</th>
<th>ADXL345</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation Voltage</td>
<td>3.3V</td>
<td>3V</td>
<td>2.0-3.6V</td>
</tr>
<tr>
<td>Operation Current</td>
<td>56μA</td>
<td>0.6mA</td>
<td>40μA(0.1μA)</td>
</tr>
<tr>
<td>Communication</td>
<td>I2C</td>
<td>I2C</td>
<td>I2C / SPI</td>
</tr>
<tr>
<td>ADC Resolution</td>
<td>16 bits</td>
<td>16 bit</td>
<td>16 bit</td>
</tr>
</tbody>
</table>

Figure 2 Sensor overview

2.1.3 Radio module
For the radio interface the platform provides, an ‘XBeeS1’ module is used to communicate the captured values to a nearby computer or a second measurement system. The module uses the UART protocol to communicate with the microcontroller. Because the mbed platform provides a library for this communication module, the choice to use the ‘XBeeS1’ module for communication was obvious.

2.1.4 SD card module
The integrated micro SD card module provides the possibility to save captured data onto a micro SD card to process the data later on. The module is set up with the help of another mbed library for SD card modules (Description in section 3.4) and communicates via SPI.

2.2 MBED platform
The ARM mbed platform provides several software libraries that are developed by ARM and could be used by the microcontrollers which are based on the ARM architecture. Its programming language is based on C/C++. The platform itself supports several microcontrollers and provides the ability to write and build programs in an online development
compiler. Because the ‘SentioEM3’ is a microcontroller platform developed by the Department of Electronics Design at Mid Sweden University, there is no direct support. However, to use mbed for the microcontroller, the mbed basic library has to be ported into Simplicity Studio IDE (section 2.3).

To understand the structure of the platform, Figure 3 shows the different layers of the basic mbed library:

![Mbed library layers]

The figure shows that the first three layers are MCU independent which means, that they do not have to be changed. The lowest layer which contains the MCU registers is hardware dependent. To connect the upper three layers to the hardware layer, the two MCU dependent layers ‘CMSIS-CORE’ and ‘HAL’ have to be customized in software to adapt to the SentioEM3. This customization is described in detail in section 3.1.

The mbed platform was chosen for the implementation of this project, because the libraries needed for this measurement system are available in this platform, for example the I2C libraries for the conversation with the sensors, a library for the ‘XBee’ radio module and a library for the SD card.

For a better understanding of the mbed platform the following is a short description of the software libraries used and how they work. First there
is the basic mbed source library, which structure was described earlier. It is necessary to implement in the project because it builds the base for all other libraries used. Already implemented in this source library is the communication via I2C. In theory, you call a function and give it the SDA and SCL pins as its parameters.

The following us an example for the sensor ‘ISL29125’:

```c
I2C ISL29125(PC6, PC7);//SDA,SCL
```

This C++ constructor function creates an object that provides an I2C bus condition and enabling communication between microcontroller and sensor. The SD card library works in a similar way. A function is called which sets up a file system and the communication between the microcontroller and the micro SD card module via SPI:

```c
SDFileSystem sd(PB3, PB4, PB5, PB6, "sd"); //MOSI,MISO,SCLK,SSEL
```

As input, the function obtains the microcontroller pins for MOSI, MISO, SCLK, SSEL as well as the name for the file system created. In this way it is possible to create new folders and files on the SD card with further functions and with the use of the FAT File System, which is also included in the library. The ‘XBee’ library, whose constructor function works in the same way, creates a new ‘XBee’ object as follows:

```c
XBee802 xbee = XBee802(RADIO_TX,RADIO_RX,RADIO_RESET,RADIO_RTS, RADIO_CTS, 38400);
```

The values for the function parameters are set in the ‘XBee’ library configuration file and have to be adapted to the pins of the microcontroller before use. A detailed description of the implementation of the libraries used can be found in Chapter 3.

### 2.3 Simplicity Studio and XCTU

Simplicity Studio and XCTU are the two most used programs in this project. To program the software, the program Simplicity Studio IDE by Silicon Labs is used. Because SentioEM3 runs with an ARM based EFM32GG processor by Silicon Labs this IDE is the obvious first choice because it provides setup ready empty projects for the EFM32GG processor type. The program is based on the open-source IDE ‘Eclipse’.
XCTU is the configuration platform used to set up the ‘XBee’ remote radio module on the computer side and receive the sent messages with the sensor values from the measurement system. The program allows the configuration of all parameters in the ‘XBee’ module, such as the address parameters.
3 Implementation

This section describes how the program was implemented. First the mbed library had to be adjusted to the EFM32GG 380F1024 Microcontroller. The second step is the implementation of the sensors followed by the ‘XBe’ radio module and the SD card module. Further details on the software developed can be found in the appendix.

3.1 Import the ARM mbed library

This section will describe how the ARM mbed library is implemented for a target EFM32 GG380F1024 which is not supported by the mbed website. First, the mbed library is fetched from the ARM developer website[6] as well as information about how the library is built and what files have to be replaced. Because mbed does not supply the used microcontroller it is recommended to download the library for a architecture similar to the supported EFM32GG STK3700. The following description of the implementation of the mbed library is therefore made with the source code for the EFM32GG STK3700.

The mbed source library itself consists of four parts that are divided into the following folders: ‘api’, ‘common’, ‘hal’ and ‘targets’ as can be seen in Figure 4. After downloading, some files have to be changed in order to make the source library work for the new target. As mentioned in Chapter 2, the ‘HAL’ and ‘CMSIS’ related files have to be replaced or adjusted. These files are located in the folder ‘target’ in the mbed library.

- mbed/targets/hal: The HAL implementations
- mbed/targets/cmsis: CMSIS-CORE sources

Figure 4 Target dependent directories[5]
For the ‘CMSIS’ Layer the following files have to be replaced by the Silicon Vendors files for the specific microcontroller, in this case the EFM32GG380F1024:

- startup DEVICE.s --> replaced by startup_efm32gg.S
- system DEVICE.c --> replaced by system_efm32.c
- system DEVICE.h --> replaced by system_efm32.h
- DEVICE.h --> replaced by efm32GG280F1024.h
- em_device.h --> replaced by em_device.h (for EFM32GG)
- DEVICE.ld --> replaced by efm32gg.ld

It is important for the startup file (startup_efm32gg.s), that its content is customized by the following. Because the mbed library uses a different name for ‘_vectors’ this section in the startup file has to be customized to ‘.isr_vector’:

```
.section .isr_vector
.align 2
.globl __isr_vector
__isr_vector:
.long __StackTop /* Top of Stack */
.long Reset_Handler /* Reset Handler */
...  
.size __isr_vector, . - __isr_vector
```

Figure 5 Startup file after customization

The reason for this is that mbed needs to be able to dynamically set the interrupt vector table [5]. It is also the reason why the new linker script efm32gg.ld has to be edited. In the beginning of the linker script an additional ‘room’ is made at the very beginning of RAM at 0x20000000. An additional 220 bytes is needed for that so the vector size is 0xE0. Also the data part in the linker file has to be added for this reason. Finally the name ‘.vectors’ has to be replaced by ‘.isr_vector’ in one part.
After customization the parts in the linker script should look like this:

```c
__vector_size = 0xE0;

KEEP(*(.isr_vector))
    *(.text*)

.data : AT (__etext)
{
    __data_start__ = .;
    *("dma")
    PROVIDE( __start_vector_table__ = .);
    += __vector_size;
    PROVIDE( __end_vector_table__ = .);
    *(vtable)
    *(.data*)
    . = ALIGN (4);
    *(.ram)
```

The last change in the ‘CMSIS’ layer is the replacement of the device specific header files. All ‘efm32gg_xxxxx.h’ files have to be replaced by the ‘efm32g3_xxxxx.h’ files, which are provided by Silicon Labs for the EFM32GG microcontroller.

In the Hardware Abstraction Layer (HAL) only some changes in the device specific hardware files have to be made.

In the file ‘device_peripherals.h’ a change of the ‘HFXO_FREQUENZY’ has to be done. It has to be set to 32MHz in order to get the microcontroller running at a maximum performance:

```
#define HFXO_FREQUENCY 32000000
```

Also the file ‘PinNames.h’ has to be edited to be able to use the LED’s and Push Button by just calling their names. The following pin numbers have to be edited for the correct usability of the EFM32GG380F1024:

```c
LED0 = PC10,
LED1 = PC9,
LED2 = PC8,
SW0 = PC11,
SERIAL_TX = PF6,
SERIAL_RX = PF7,
```
The mbed library is now adjusted for the EFM32GG380F1024 microcontroller. To include it in the running program a new empty project in simplicity studio has to be created. As the used kit the EFM32GG380F1024 has to be chosen. The created project comes with standard library files that have to be replaced by the adjusted mbed library. After the inclusion of the mbed library the files have to be included manually in the project properties. Under project properties --> C/C++Build --> Settings --> Includes all folders have to be added to tell the GNU ARM C and the GNU ARM C++ Compiler where the files are saved:

![Figure 6 Program properties and includes](image)

Finally, before the program is ready to run the Linker Script has to be linked. In the settings menu as shown above, the index tab ‘Memory Layout’ provides the opportunity to choose a manual Linker Script. The Linker Script EFM32gg.ld mentioned previously has to be placed here.

13
3.2 Sensor implementation

This section deals with the implementation of the three sensors used. All header files containing the functions of the sensors are also included in the appendix.

3.2.1 TSL2561 luminosity sensor

First the implementation of the luminosity sensor, which functions are stored in the header file ‘TSL2561.h’, will be described. This file is included in the main file to be used in the main program. The header file for the sensor contains a general setup for the I2C communication as well as commands that configure the sensor. To be able to call these tasks from the main program later on, they are put in the function ‘getTSL2561()’ which returns the values for the visible light, the infrared light and the lux value in the main program in order to save them into a file on the SD card or send them via ‘XBee’ to a remote receiver. The function starts with the setup of an I2C object called ‘TSL2561’:

```c
I2C TSL2561(PC6, PC7);
```

Pin PC6 and PC7 represent the microcontroller pins for SDA and SCL. Because the I2C read and write functions are provided by the mbed library, the address value for the sensor has to be stored in an integer variable. The hexadecimal address for this sensor is ‘0x39’ but because mbed works with 8 bit addresses instead of the for I2C typical 7 bit addresses, ‘0x39’ has to be shifted to the left by one:

```c
int address = (0x39<<1);
```

By calling the function ‘TSL2561.frequency()’, the frequency of the I2C bus can be adjusted. In order to communicate with the sensor and to be able to access its registers, a value has to be sent with the write function mbed provides. This value can be stored into a char array ‘data0’ that has the length [2]. The first element [0] gets the address of the register and the second element [1] gets the value that should be written into the register. For the ‘TSL2561’ the power control register has to be set to ‘on’ in order to start up the sensor. In addition the mbed write function needs to be told how many bytes or elements of the array it should deliver:

```c
data0[0] = 0x80;
data0[1] = 0x03;
TSL2561.write(address, data0, 2);
```
After the call, the sensor is ready to sense. The sensed values get stored in two channel registers for the visible light and infrared light respectively. CHANNEL 0 represents the values for the visible light. This register can be read by using the mbed write function to set the address of the channel register first, followed by the read function to read out the sensor value. The sensor values get stored in the two elements of the array.

```c
    data0[0] = 0x8C;
    data0[1] = 0x00;
    TSL2561.write(address, data0, 1);
    TSL2561.read(address, data0, 2);
```

Because the sensor values are stored in 8 bit registers on the sensor (CHANNEL 0 lower byte and CHANNEL 0 higher byte) the value has to be set together in software afterwards. This works out by putting the two values that are stored in the array into an integer. The first value of the first element is shifted to the left by eight and the other value becomes logical or connected to result in the measured value, which is represented by a 16-bit integer:

```c
    dataCH0 = ( (uint16_t)data0[1] << 8 ) | data0[0];
```

The same procedure of reading out the sensor values is carried out for CHANNEL 1, which contains the values for the infrared light. Finally, both channel values are additionally processed in the function ‘getLux()’ which calculates and returns a lux value. This function is taken from the example program of the developer webpage [7], where it is described in detail.

### 3.2.2 ISL29125 RGB light sensor

Because the ‘ISL29125’ sensor works in a way similar to the ‘TSL2561’, only the differences of this sensor will be described, rather than going into full detail. The main difference is that the sensor itself senses three different values, which means that three channel registers have to be read out. The mechanism is the same as for the ‘TSL2561’. The data is read out and then stored into an integer that is returned to the main program by the function. However, in the beginning of the function the sensor needs to be set up a bit different. Instead of powering the sensor up it needs to be set up to a specific measurement mode in the configuration register (0x01). For this example, the sensor gets set up to the mode that measures all colors green, blue, red and additionally puts the lux intensity to 10k which offers a higher resolution:
3.2.3 **ADXL345 triple axis accelerometer sensor**

The setup for the ‘ADXL345’ sensor works similar to the other two sensors. A channel for each direction (x, y and z) stores the measurements on the sensor. The difference compared to the other sensors is the way of measuring. While the programs for the first two sensors sense one value every time the sensor is attached, the accelerometer sensor should measure 1000 values in one second to find the dynamic change of the acceleration in the sensor’s environment. In order to set up the device, some values have to be written into the associated registers. These set ups are carried out within the function ‘getADXL345()’ which is used in the main file to return the sensor values. To define the range in which the sensor measures, the DATA_FORMAT (0x31) register has to be adjusted. By writing a 0x01 to this register, the sensor works within a range of +/- 4G. The second register that has to be adjusted is the BW_RATE (0x2C, bandwidth rate) register in order to set the sensing frequency to its maximum of 3200Hz. This is necessary to fulfill the task of measuring 1000 values every second (default value for this register is 100Hz). The final register that has to be set is the POWER_CTL (0x2D) register in order to set the sensor to measurement mode (0x08). After the setup is complete, the sensor values have to be read from the channel registers with the mbed read function in the same way as for the other sensors. Because it is about a huge amount of data that comes in from the sensor, it is useful to complete this task with a time controlled interrupt that reads out the values every millisecond and stores the captured data into an integer array. Mbed provides a timer interrupt called ‘ticker’ which is the perfect function to use for this work. The ticker function creates an object that calls and repeats a defined function after a defined time until it is detached. So when the ticker gets attached, it calls the function ‘get_v_data()’ that reads out the sensor values from the channel registers of the sensor and stores them into buffer arrays.
The enabling of the interrupt looks like this:

```c
Ticker inter;
READ_READY = false;
inter.attach(&get_v_data, 0.001);
while(!READ_READY);
```

With that, the interrupt calls the function every millisecond to get a new value. Boolean READ_READY forces the program to wait until a specific value is reached (1000 in this case) to carry on with the program. The function ‘get_v_data()’ looks like this:

```c
dat[0] = (0x32);
ADXL345.write(address, dat, 1);
ADXL345.read(address, values, 6);

//The value for x is stored in values[0] and values[1]
xbuffer[n][0] = (int)values[0];
xbuffer[n][1] = (int)values[1];

if(n>=anzahl)
{
  //disable the interrupt
  inter.detach();
  //set to true to continue with the code
  READ_READY = true;
}
```

The function sets up the register address for the channel values to read them and store them one after the other into a two-dimensional integer array buffer for every sensing direction x, y and z. To be able to tell when 1000 measurements are done, an integer variable ‘n’ counts up every time the interrupt occurred and when the defined number has been reached the interrupt is detached and READ_READY is set to ‘true’ to continue with the code. Back in the ‘getADXL345()’ function the variables in the buffers need to be read out and put into a string, separated by a comma in order to be able to return the large number of measurements to the main program and save them into a file. This is done by two position pointers that read out the values in the buffer and store them in the string. The example for the x direction looks like this:

```c
for (k=1;k<anzahl;k++)
{
  x=((int)xbuffer[k][1]<<8)|(int)xbuffer[k][0];
  posx += sprintf(&vibx[posx],", %d",x);
}
```
The values of the buffer are first combined in an integer variable \( x \), similar to the process in the TSL 2561 sensor mentioned previously. This variable is stored at the end of the string whose position pointer counts up to the last element. By the end this string is returned to the main program for further processing.

3.3 XBeeS1 radio module implementation

To use the radio communication interface of the platform, mbed provides a library for the ‘XBee’ module as mentioned before. To be able to use the functions from the library the file ‘config.h’ in the library has to be customized by defining the hardware platform pins to the software:

```c
#define RADIO_TX                PE10
#define RADIO_RX                PE11
#define RADIO_RTS               PB0
#define RADIO_CTS               PA15
#define RADIO_RESET             PE9
#define RADIO_SLEEP_REQ         PE12
#define RADIO_ON_SLEEP          PE13
#define DEBUG_TX                PF6
#define DEBUG_RX                PF7
```

The functions that are necessary to send and configure the ‘XBee’ module for the use of the program are stored in the header file ‘XBee_Send.h’. First in this file the addresses of the used ‘XBee’ modules have to be set, because every module has a unique 64-bit address. The mbed library also provides the possibility to send data to a 16-bit address. This address is not unique and can be changed using the tool XCTU as mentioned in section 2.3.

To make the program run faster and more efficiently, the initialization and the sending of the data with the module is divided into two parts. An ‘XBee_startup()’ function initializes the module at the first run of the program when the module is enabled in the main program (described in section 3.5). The startup function contains the initialization and also checks if it was a successful. After the call of the function the ‘XBee’ is enabled and ready to send data. To send the data another function of the mbed library is used. The function is called ‘xbee.send_data()’ and is used in two functions that set up the captured data and sends it to the remote receiver. These two functions, called ‘XBee_send_RGB()’ and ‘XBee_send_light()’ retrieve the sensor values from the main program and copy them into a string that is sent to the receiver. The receiver can convert the string into the required form.
and process it. The example shows the send for one color of the ‘ISL29125’ sensor:

```c
void XBee_send_RGB(unsigned int blue, unsigned int green, unsigned int red)
{
    memset(data,0,32);
    memcpy(data, (char*)&blue,2);
    data_len = strlen(data);
    xbee.send_data(remoteDevice16b, (const uint8_t)data, data_len);
}
```

First the string is emptied and then the value of the measured color blue is copied into the string. Next, the string length is determined to, later on, tell the send function how long the string is. In addition, the send function needs the address of the remote node and the data itself to execute the command. The commands for the other colors green and red as well as the send function for the values of the ‘TSL2561’ sensor work in the same way. Before each value is sent, a string clarifies in which order the values arrive. For example the string for the ‘ISL29125’ sensor looks like this: “Sensor data RGB blue/green/red”. This can also be seen in the pictures below, which show the incoming frames from one measurement.

![Picture 1 - Incoming XBee data recorded with XCTU](image)
### Frame details

**Length**

00 24 (36)

**Frame type**

B1 (RX (Receive) Packet 16-bit Address)

**16-bit source address**

AA AA

**RSSI**

27

**Options**

00

**RF data**

<table>
<thead>
<tr>
<th>ASCII</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor Data RGB blue/green/red:</td>
<td></td>
</tr>
</tbody>
</table>

**Checksum**

48

---

Picture 2 - Incoming frame details
3.4 SD card implementation

Like for the radio module, the implementation of the SD card module to save the captured sensor values onto a micro SD card is also done with a library provided by mbed. The functions for setup and for saving the values on the SD card are placed in the header file ‘SD_Config.h’. Like for the radio module implementation, the setup and the saving of values for the SD card implementation is separated into two parts. The created file is a ‘.txt’ file (text file).

To setup the file system it is necessary to create a new SDFileSystem object with the associated function:

```
SDFileSystem sd(PB3, PB4, PB5, PB6, "sd");
```

The object enables communication between the microcontroller and the micro SD card slot via SPI. Like for the other mbed functions it is necessary to provide the hardware pins, in this case for MOSI, MISO, SCLK and SSEL (SPI). The name for the file system on the SD card in this example is ‘sd’. Next, the set up creates the directory where the file should be placed:

```
mkdir("/sd/LightSensor", 0777);
```

In order to be able to make more than one measurement without removing the SD card first, to save the file with the captured values, the files have to be named differently for each new starting measurement. Otherwise they would be erased at the start of a new measurement. The implementation of a small function, which counts the files already in the directory of the created file system, therefore returns the value as an integer. This number is copied into a string that is used to create the new file in the directory:

```
file_number = read_file_number(dir_light);
sprintf(filename_light,"/sd/LightSensor/Measurement_%d.txt",file_number);
light = fopen((const char*)filename_light, "w");
```

By applying this method, the old files in the directory will never be erased or replaced by the new file with the same name. The ‘read_file_number()’ function works as follows: it searches the created folder and sets up a counter variable for each file in the directory. Then the value is returned as an integer. At the end of the setup, the file is opened and a header for the files is added, which describes the values
that will follow and in what way they are sorted. Also from which sensor the data comes is included. To be able to have different folders, files and headers in these files for the two light sensors and the accelerometer sensor, a set up function is implemented for each of them. These functions are called ‘Setup_SDcard_Light()’ and ‘Setup_SDcard_Vibrate()’.

To access the files and to write data into them, a function is implemented that fetches the sensor values from the main program, opens the previously created files and stores the data in these files. This applies to both files (accelerometer and light sensor) in two different functions that work in the same way. The function for the light sensors is called ‘write_light_to_file()’:

```c
void write_light_to_file(...) {
    SDFileSystem sd(PB3, PB4, PB5, PB6, "sd");
    light = fopen((const char*)filename_light,"a");
    fprintf(light, "%s,%d,%d,%d,%d,%d,%d\n", zeit, blue, green, red, vis, ir, lux);
    fclose(light);
}
```

The function not only writes the data into the file, it also formats the file. First the time is written into the file, followed by the sensor values that are divided by a comma. This makes it easier to process the data later on, to import into Matlab, Excel etc. Figure 7 provides an example of a file of the light sensors that was created with the program. The values for both light sensors are written into the same file, because both sensors are used together, and the values for the acceleration are saved in another file that follows the same rules of creation. Figure 8 shows the arrangement of different files that are created for each new measurement in the directory.
Wireless sensor system to characterize energy harvesting conditions

Felix Hörnschemeyer

3 Implementation

2016-07-12

Figure 7 Several values stored in one file during one measurement

Figure 8 File structure in the folder LightSensor
### 3.5 Main file implementation

All the header files with the described functions for the different setups for sensor, radio module and SD card are accessed from the main file. The file’s name is main.cpp and it contains the ‘main()’ function that is called when the program is running on the microcontroller. In order to have an energy and work time efficient program it is necessary to set up and call only those functions that are necessary for the running program. What is meant by this is that single sensors or the method of how the data is processed can be selected or deselected. This means that it would be also possible to measure the light sensors only and save them onto the SD card if required. These options are implemented by a ‘define’ for each module similar to:

```c
#define SENSOR_RGB 1
```

It is only when the value is not zero that the module is active and is setup and processed. In this example the ‘ISL29125’ RGB sensor would be activated. Next there are two Booleans defined that are used to check if the SD card or the ‘XBee’ initialization has already been done. These Booleans are set to false at the start and after the program had been running for the first time they were set to true, in order to save energy and time by not initializing the modules each time the program runs. The next thing defined is an integer called ‘measure_intervall’. This integer can be set to the interval or period duration the program should wait during two measurements.

#### 3.5.1 Timer controlled interrupt implementation

To provide the possibility to wait after the program has completed its task until the next capturing, it is necessary to set the microcontroller and all used modules to a sleep mode in order to make the program more efficient and to save energy. The EFM32 microcontroller provides different sleep modes. For this program the EM2 energy mode was selected, because it is the deepest (most efficient) energy mode where it is still possible to wake up the modules with a timer controlled interrupt. This timer controlled interrupt is set up in the header file ‘Timer0_Config.h’. To set up the interrupt the low frequency clock ‘LFRCO’ has to be enabled and attached to the low energy timer (LETIMER0) to be able to count during the sleep mode. After that the interrupt parameters for the timer have to be initialized and the time the interrupt has to be called has to be calculated.
The calculation works as follows:

```c
period = (uint16_t) 1 * (letimer_clock/1) +0.5;
uint32_t timertop = period * s;
```

By calculating the variable ‘period’, the program calculates the steps the clock counts during one second. The normal ‘timer0’ speed running at the ‘LFRCO’ clock is 32768Hz. Due to the fact that the used counter is a 16 bit counter and therefore only able to count to 65535, a prescaler of 4096 is used here to divide the clock speed to 8Hz. By so doing, much higher waiting intervals are possible during two measurements. (65535/8Hz equals more than two hours). The variable ‘timertop’ in the calculation above calculates the value that the timer is set to by multiplying the previously calculated value ‘period’ with the over-handed variable ‘s’ from the main program, which equals the time in seconds. At the end of the setup the timer is set to count down until it reaches an underflow and the interrupt occurs. Every time the program is running the timer is set to the variable ‘timerhop’ and counts down till it reaches 0. Then an interrupt occurs which wakes the microcontroller from sleep to process the measurement for the new period.

### 3.5.2 Main function implementation

The ‘main()’ function in the main file accesses and processes all the defined and described functions that were mentioned previously. At the beginning of the function two LEDs, LED0 and LED2 of the platform are turned on to show that the program is running, also when the system is not connected to a PC and running on a battery. LED0 shines green to show that the microcontroller is working and LED2 shines red when the program is running. This means that LED2 is turned off when the microcontroller is in sleep mode and lights up when the program is actively running. Important to make the mbed libraries work correctly is to call the function mbed_sdk_init() at the very beginning to match and adjust the clocks of the microcontroller to the libraries. Next, the timer function is called which sets up the interrupt as described in section 3.5.1.

Because the program should be able to run several times until the microcontroller is shut down, it is necessary to create a loop that the program runs through every time the microcontroller wakes up from sleep mode. This loop is implemented with a ‘while(1)’ loop which repeats the code within until the command becomes ‘false’ (which cannot happen
because 1 always remains 1 = ‘true’ here) or the microcontroller is shut down as mentioned previously. The code in the loop starts by measuring the time. Because time grows linearly and the measurement period is based on this time, the variable ‘zeit’ is added by one ‘measurement_intervall’ every time the loop runs. To display the time clearly it is formatted as: ‘hh:mm:ss’, and packed into a string for better processing while writing it into the files on the SD card:

zei += measure_intervall;
sprintf(zeit,"%d:%d:%d",zeit/3600,(zeit/60)%60,zeit%60);

After that, the actual measurements start. Because it makes more sense to have the same values read out from the sensors to write on the SD card and to send via ‘XBee’ it is necessary to first read out the sensor and then process the data. Because it is possible to deactivate single sensors or methods, the program asks if the ‘defines’ from above are true to proceed the function. The example for the ‘ISL29125’ looks like this:

```c
if(SENSOR_RGB)
{
    //get RGB sensor values
    getISL29125(blue,green,red);
}
```

The previously mentioned function used to retrieve the sensor values is called and returns the results into the variables blue, green and red. These variables can then be written into a file on the SD card or be sent with the ‘XBee’ module. This happens after all three sensors, if they are activated, are read out. First, when the SD card mode is activated and the program enters the segment, the program asks if the module was already set up (Boolean SD_CARD_INIT = ‘true’ or ‘false’). In case the function runs for the first time, the Boolean is still defined as ‘false’ and the SD card is set up. Next, the Boolean is set to ‘true’ and the next time the program will only write the data to the file. This ‘write to file’ follows the set up and can be illustrated by this example for the light sensors:

```c
if(SENSOR_LIGHT||SENSOR_RGB)
    write_light_to_file(zeit,blue,green,red,visib,ir,lux);
```
The measured variables are handed over to the function and written into the file that was created during the setup. The same exact procedure is repeated for the ‘XBee’ module to send the data to a remote receiver.

After all the sensors have been read out and the data processed, the microcontroller goes to sleep mode until the interrupt wakes it again to perform the loop again. In addition to the sleep mode the microcontroller is set to, it is important to also set the SD card module and the ‘XBee’ module to a sleep state to save energy. Both modules have an enable/sleep function that can be controlled using a GPIO pin. The mbed library provides a function facilitating the setting of GPIO pins. ‘DigitalOut’ creates an object that refers to a GPIO pin output. The XBee sleep pin for example is on PE13. By assigning the created object to this pin, it can be controlled by writing a 1 or a 0 to it.

\texttt{DigitalOut XBee\_sleep(PE13);} 

By writing ‘XBee\_sleep = 1;’ for example, the Xbee is set to sleep. The same is required for the SD card, whose related pin is PA10. Both modules are set to sleep at the end of the loop and woken up when used. After that the microcontroller is set to EM2 sleep mode and wakes up when the timer interrupt occurs.
4 Results

The results are divided into three sections. In the first part the experimental setup is described. The second section shows the result of two measurements taken with the system and the third part consists of an energy consumption measurement for the system.

4.1 Experimental Setup

The following measurements for the RGB Light and Luminosity Light sensors were recorded in a student’s room without artificial lightning, and the only source of light was the sunlight coming in through the window. This renunciation of other light sources was done to see a change in the light conditions during the day. Therefore the system was placed on a desk close to the window, facing south. To measure values for the accelerometer sensor, the system was placed on top of a vibrating plate that provided acceleration in a sinus form.

The first measurement for the light sensors was taken on 28 May 2016 from 14:30 till 18:30. The second measurement was taken 18:30 till 22:30. Both measurements were recorded for a duration of four hours with a measuring period set to every ten minutes. Because of how the sun travels varies over the year, it is important to know at which latitude the measurement was taken. The location for the measurement was Sundsvall, latitude N 62° 23.329084’. For the recording of the acceleration sensor, measurements over one second were made, providing 1000 values each. The sensor was strapped onto a vibration and the sinus curve had a frequency of 30 Hz.

4.2 Measurement results

The diagrams below show the sensor values as the 16-bit number the sensors provide as an output over time. There is an additional calculation of the lux value showed for the luminosity light sensor in the diagram of the recorded light. The result from the acceleration sensor shows the deviation from the zero position as a 16-bit number the sensor provides over time. The diagrams were created using Excel and the captured values that were stored on the micro SD card.
Wireless sensor system to characterize energy harvesting conditions

Felix Hörnschemeyer

4 Results
2016-07-12

Diagram 1 RGB sensor values - Measurement 1

Diagram 2 Light sensor values - Measurement 1
Wireless sensor system to characterize energy harvesting conditions

Felix Hönschemeyer

2016-07-12

Diagram 3 RGB sensor values - Measurement 2

Diagram 4 Light sensor values - Measurement 2
An important part of a microcontroller in a measurement system is its energy consumption. Because we have a measurements system which is made to measure energy in the environment, the energy consumption plays an even more interesting role. Two values are salient for this measurement. First the energy consumption during the active, the measuring time of the microcontroller, and second the consumption during the period of sleep. Combining these values provides us with an overview of the complete energy consumption of the measurement system for the time of the operation. Figure 9 shows the energy consumption of one measurement period. The measurement was taken with a frequency of 500Hz and was taken with Matlab. Also all sensors and modules were enabled for this measurement. The energy consumption during sleep mode is approximately 900µA. The active part of the
sensor can be divided into three parts: The first part from 6400 to 7050, which has a consumption of 35mA, and it is where the measurements for the sensors are made. The second part is from 7050 to 7100, which consumes between 40mA at the lowest and 160mA at the highest and it is the part (average of 100mA) of the program where the data is written onto the SD card. The third part is from 7150 to 7400, with an energy consumption of between 40mA and 160mA, and it is the activation of the ‘XBee’ module, the sending of the data to the remote receiver. As can be seen, the active time where the program is running is about 2 seconds. (7400-6400 = 1000; 1000/ 500Hz=2s).

Figure 9 - One period of measurement
5 Discussion and Conclusion

5.1 Discussion

As can be seen in the diagrams for the light sensors in Chapter 4 the first measurement starts with low values. These values climb higher until they reach a first maximum. After that they fall and then reach a new climax. That its value increases, which means an increase of brightness, can be explained by the moving sun, which was rising and shining more from the south during the measurement. During the short period where the light values get lower, clouds disrupted some of the light from getting to the sensor. In measurement 2 it is clear that the sun moves further away and the light slowly disappears until all values are close to zero by 22:30. The diagrams show measurements that are expected because it could be assumed that would rise in daytime and set in the evening. The measurement for the accelerometer sensor, which was taken on a vibration plate, shows the desired form of a sinus curve that was given as the input for the plate. Also the measured frequency of 30Hz matches the input of the plate.

For both types of sensors, light and accelerometer, the results show that the system works as it should. Also saving on the SD card and sending via ‘XBee’ to a remote receiver works as desired and as presented in Chapter 3 previously. The results show that the system works and the project’s goals have been achieved. The results do not provide any details on the environment or other parameters measured; what they do is clarify the way the system works and that it works.

5.2 Conclusion

To conclude the project, it can be said that all steps leading to the overall aim were achieved as the results show. The system provides the possibility to measure energy in different environmental areas and indicate if there is energy that could be harvested. Therefore the project can be considered as a success for the beginning of building a measurement system with several nodes. The results and measurements have the desired values and as such provide a platform to work with. Building one node was the first step towards a whole energy measurement sys-
System including several nodes, which is able to provide concrete and flexible statements about energy harvesting.

5.3 Future work
After completing the construction of a platform making its own system, the next step would be, as mentioned in the introduction already, to include more nodes to expand the system. This system could consist of several nodes that are able to communicate with each other (sending measured values to receiver nodes) and therefore be able to cover a great area for environmental measurements. It could be applied in areas where there is a high energy occurrence on a daily basis like by the ocean (wind), in the desert (solar), or in industrial halls. The whole system would be able to measure the environmental energy in these areas (mapping) and indicate whether there is a sufficient amount of energy that could be harvested.

5.4 Social and ethical aspects
Society and people can benefit from this type of system in the future as it can show where energy can be harvested, collected and in this way save energy for batteries that would otherwise have to be charged to power a sensor system. This is and aid in building a more efficient environment where little energy can be saved. Using a large number of autonomous sensor systems, the small amount of energy saved by each system becomes a quite large amount of energy. When it comes to the green revolution that starts in our continent and other parts of the world it is also necessary to not just look at how we can produce more energy in a renewable and green way, but also how we can save energy. This is a factor that will play an important role in the future, how we can use energy in a more efficient way and save energy rather than waste it.
References


Appendix: Documentation program code developed

Main.cpp

#include "mbed.h"
#include "mbed_debug.h"
#include "TSL2561.h"
#include "ISL29125.h"
#include "ADXL345.h"
#include "XBee802_Send.h"
#include "XBeeLib.h"
#include "mbed_assert.h"
#include "RTC_Config.h"
#include "SDFileSystem.h"
#include "mbed_overrides.c"
#include "SD_Config.h"
#include "Timer0_Config.h"

DigitalOut myled0(LED0);
DigitalOut myled1(LED1);
DigitalOut myled2(LED2);

// Set the used Sensors
#define SENSOR_RGB 1
#define SENSOR_LIGHT 1
#define SENSOR_VIBRATION 1

// Set the used Data Methods
#define MODE_SD_CARD 1
#define MODE_SEND_XBEE 1

// Bool to control if the initialization is done
bool SD_CARD_INIT = false;
bool XBEE_INIT = false;

// Measurement interval defined in seconds
int measure_intervall = 10;

// Variables were the sensor measurements get stored in
unsigned int blue = 0, red = 0, green = 0;
unsigned int visib = 0, ir = 0, lux = 0;
char vibx[8000] = {0};
char viby[8000] = {0};
char vibz[8000] = {0};

// Variables that store the time while measuring.
unsigned int zeit = -measure_intervall;
char azeit[12] = {0};

DigitalOut XBee_sleep(PE13);

int main()
{
    myled0 = 1;
    myled2 = 1;

    // Initialize the microcontrollers clocks with the mbed library's
    mbed_sdk_init();
    wait[1];

    // Set the LR TIMER0 to an interrupt every (measure_intervall) seconds
    timer0_setup(measure_intervall);
Wireless sensor system to characterize energy harvesting conditions

Felix Hörnschemeyer

Appendix: Documentation program code developed

2016-07-12

//Enter the loop that the microcontroller processes during one measurement period
while(1){

//Calculate the time and convert it in a string hh:mm:ss
zeit += measure_intervall;
sprintf(azeit,sizeof(zei
20230111\n
if(SENSOR_RGB)
{ //get RGB sensor values
getISL29125(blue,green,red);
}

if(SENSOR_LIGHT)
{ //get Light sensor values
getTSL2561(visib, ir, lux);
}

if(SENSOR_VIBRATION)
{ //get vibration sensor values
getADXL345(vibx, viby, vibz);
}

if(MODE_SD_CARD)
{ SD_enable = 1;
if(!SD_CARD_INIT)
{
if(SENSOR_LIGHT || SENSOR_RGB)
{ //Set up the SD card file system for the light sensors
Setup_SDcard_Light();
}
if(SENSOR_VIBRATION)
{ //Set up the SD card file system for the vibration sensor
Setup_SDcard_Vibrate();
SD_CARD_INIT = true;
debug("\nSD Card Mode enabled\n");
}

//Write the values into the files
if(SENSOR_LIGHT || SENSOR_RGB)
write_light_to_file(azeit,blue,green,red,visib,ir,lux);
if(SENSOR_VIBRATION)
{ write_vibration_to_file(azeit,vibx,viby,vibz);
}

//Send the sensor values to a remote module
if(MODE_SEND_XBEE)
{ //set the sleep pin to zero to wake the XBee up
XBee_sleep=0;
wait_ms(10);
if(!XBEE_INIT)
{
XBee_startup();
XBEE_INIT = true;
debug("\nXBEE Mode enabled\n");
}
Wireless sensor system to characterize energy harvesting conditions

Felix Hönschemeyer

Appendix: Documentation program code developed
2016-07-12

```
//Send the RGB sensor values
if(SENSOR_RGB)
{
    XBee_send_RGB(blue,green,red);
}

//Send the light sensor values
if(SENSOR_LIGHT)
{
    XBee_send_light(visib,ir,lux);
}

myled2 = !myled2;
XBee_sleep = 1;
SD_enable = 0;
deepsleep();
myled2 = !myled2;
```
Wireless sensor system to characterize energy harvesting conditions

Felix Hörnschemeyer

Appendix: Documentation program code developed 2016-07-12

ADXL345.h

/*
 * ADXL345.h
 * Created on: 21.05.2016
 * Author: Felix
 */

#ifndef ADXL345_H_
#define ADXL345_H_

// This buffer will hold values read from the ADXL345 registers.
char values[10];

// These variables will be used to hold the x, y and z axis accelerometer values.
int x, y, z;

// Values that are measured in one measurement are stored in int arrays
int xbuffer[2000][2]={0};
int ybuffer[2000][2]={0};
int zbuffer[2000][2]={0};

// Counting variables
int n = 0;
int k = 0;

// Position pointer
int posx = 0;
int posy = 0;
int posz = 0;

// Number of measurements
const int anzahl = 1000;

// Create the I2C Module
I2C ADXL345(PC6, PC7);
int address = (0x53<<1);
char dat[2] = {0};

// Create the Interrupt using Ticker function by mbed
Ticker inter;
bool READ_READY = false;
bool ADXL_INIT = false;

/*
 * get_v_data reads out the values from the Sensor using I2C
 * The data for x, y and z is stored in two bytes respectively
 * The data gets stored in the int array buffers
 * This function is proceeded during the interrupt
 */

void get_v_data()
{
    dat[0] = (0x32);
    ADXL345.write(address, dat, 1);
    ADXL345.read(address, values, 6);

    // The value for x is stored in values[0] and values[1]
    xbuffer[n][0] = (int)values[0];
    xbuffer[n][1] = (int)values[1];

    // The value for y is stored in values[2] and values[3]
    ybuffer[n][0] = (int)values[2];
    ybuffer[n][1] = (int)values[3];

    // The value for z is stored in values[4] and values[5]
    zbuffer[n][0] = (int)values[4];
    zbuffer[n][1] = (int)values[5];

    if(n>=anzahl)
    {
        // Disable the interrupt
        inter.detach();

        // Set to true to continue with the code after the read from the sensor done.
        READ_READY = true;
        // debug("Interrupt detached\n");
    }
    n++;
}
/* getADXL345 performs the setup for the sensor, enables the Interrupt to read out the sensor values and gives the sensor data back in form of three strings for x y z */

bool getADXL345(char (&vibx)[8000], char (&viby)[8000], char (&vibz)[8000])
{
    //Set the frequency on the I2C
    ADXL345.frequency(400000);

    //Put the ADXL345 into +/- 4G range by writing the value 0x01 to the DATA_FORMAT (0x31) register.
    dat[0] = (0x31);
    dat[1] = (0x01);
    ADXL345.write(address, dat, 2);

    //Put output rate to 3200 Hz by writing 0xFF into the the BW_RATE register (0x2C)
    dat[0] = (0x2C);
    dat[1] = (0xFF);
    ADXL345.write(address, dat, 2);

    //Put the ADXL345 into Measurement Mode by writing 0x08 to the POWER_CTL (0x2D) register.
    dat[0] = (0x2D);
    dat[1] = (0x08);
    ADXL345.write(address, dat, 2);

    //wait till ADXL sensor is started and ready to reply
    if (ADXL_INIT)
    {
        wait_ms(10);
        ADXL_INIT = true;
    }

    //set the buffers content to zero to ensure no old data within
    memset(xbuffer,0,2000);
    memset(ybuffer,0,2000);
    memset(zbuffer,0,2000);

    //read out the sensor values activating Interrupt 'inter', that calls function get_v_data() and wait till READ_READY is set to true
    n = 1;
    READ_READY = false;
    inter.attach(get_v_data, 0.001);
    while (!READ_READY);

    //set the string arrays content to zero to ensure no old data within
    memset(vibx,0,8000);
    memset(viby,0,8000);
    memset(vibz,0,8000);

    //set the counter values to zero
    posx = 0;
    posy = 0;
    posz = 0;

    //converting the int array into a string
    //Therefore both values that make one sensor value (8Bit + 8Bit) are combined in one int and then written at the end of the string, separated through a comma
    for (k=1;k<anzahl;k++)
    {
        x=((int)xbuffer[k][1]<<8)|(int)xbuffer[k][0];
        posx += sprintf(&vibx[posx],", %d",x);
        y=((int)ybuffer[k][1]<<8)|(int)ybuffer[k][0];
        posy += sprintf(&viby[posy],", %d",y);
        z=((int)zbuffer[k][1]<<8)|(int)zbuffer[k][0];
        posz += sprintf(&vibz[posz],", %d",z);
    }

    return(true);
}
#endif /* ADXL345_H_ */
Appendix: Documentation
program code developed
2016-07-12

Felix Hörnschemeyer

Wireless sensor system to characterize energy harvesting conditions

ISL29125.h

/*
 * ISL29125.h
 *  Header File for the RGB Sensor ISL29125
 * Author: Felix
 */

#ifndef ISL29125_H_
#define ISL29125_H_

bool ISL_INIT = false;

/*
 * Return the RGB values of the ISL29125 sensor in the variables:
 *  * dataBlue, dataGreen, dataRed
 */

bool getISL29125(unsigned int &dataBlue, unsigned int &dataGreen, unsigned int &dataRed)
{
    I2C ISL29125(PC6, PC7);
    int address = (0x44<<1);
    ISL29125.frequency(400000);
    char dblue[2]={0};
    char dgreen[2]={0};
    char dred[2]={0};
    //unsigned int dataBlue=0, dataGreen=0, dataRed=0;

    //Set the configuration register one (0x01) to Mode Green/Red/Blue (0x05) + the Lux intensity to 10k (0x0B)=(0x0D)
    dblue[0] = (0x01);
    dblue[1] = (0x0D);
    ISL29125.write(address, dblue, 2);

    //wait till the sensor is started and ready to reply
    if(!ISL_INIT)
    { wait_ms(400);
      ISL_INIT=true;
    }

    //read out value blue from sensor
    dblue[0] = (0x0D);
    ISL29125.write(address, dblue, 1);
    ISL29125.read(address, dblue,2);

    //read out value red from sensor
    dred[0] = (0x0B);
    ISL29125.write(address, dred, 1);
    ISL29125.read(address, dred,2);

    //read out value green from sensor
    dgreen[0] = (0x09);
    ISL29125.write(address, dgreen, 1);
    ISL29125.read(address, dgreen,2);

    //store both 8Bit Data register into one 16Bit value for every color
    dataBlue = ( (uint16_t)dblu[1] << 8 ) | dblue[0];
    dataRed = ( (uint16_t)dred[1] << 8 ) | dred[0];
    dataGreen = ( (uint16_t)dgreen[1] << 8 ) | dgreen[0];

    //Give out the Sensor Values
    //debug("Data value blue:\t%dn",dataBlue);
    //debug("Data value red:\t%dn",dataRed);
    //debug("Data value green:\t%dn",dataGreen);
    return(true);
}

#endif /* ISL29125_H_ */
TSL2561.h

/*
 * I2C_Program.h
 * Header File for the Luminosity Sensor TSL2561
 * Author: Felix
 */

#ifndef TSL2561_H
#define TSL2561_H

bool TSL_INIT = false;

/* Convert raw data to lux
 * gain: 0 (1X) or 1 (16X), see setTiming()
 * ms: integration time in ms, from setTiming() or from manual
 * integration
 * CH0, CH1: results from getData()
 * lux will be set to resulting lux calculation
 * returns true (1) if calculation was successful
 * RETURNS false (0) AND lux = 0.0 IF EITHER SENSOR WAS SATURATED
 * (0xFFFF)
 */

bool getLux(unsigned int CH0, unsigned int CH1, unsigned int &lux)
{
    unsigned int ms = 402;
    unsigned char gain = 0;
    double ratio, d0, d1;

    // Determine if either sensor saturated (0xFFFF)
    // If so, abandon ship (calculation will not be accurate)
    if ((CH0 == 0xFFFF) || (CH1 == 0xFFFF))
    {
        lux = 0.0;
        return(false);
    }

    // Convert from unsigned integer to floating point
    d0 = CH0; d1 = CH1;

    // We will need the ratio for subsequent calculations
    ratio = d1 / d0;

    // Normalize for integration time
    d0 *= (402.0/ms);
    d1 *= (402.0/ms);

    // Normalize for gain
    if (!gain)
    {
        d0 *= 16;
        d1 *= 16;
    }

    // Determine lux per datasheet equations:
    if (ratio < 0.5)
    {
        lux = 0.0304 * d0 - 0.062 * d0 * pow(ratio,1.4);
        return(true);
    }

    if (ratio < 0.61)
    {
        lux = 0.0224 * d0 - 0.031 * d1;
        return(true);
    }

    if (ratio < 0.80)
    {
        lux = 0.0128 * d0 - 0.0153 * d1;
        return(true);
    }

    //...
Appendix: Documentation
program code developed
2016-07-12

Felix Hönschemeyer

Wireless sensor system to characterize energy harvesting conditions

if (ratio < 1.30)
{
    lux = 0.00146 * d0 - 0.00112 * d1;
    return(true);
}

// if (ratio > 1.30)
lux = 0.0;
return(true);

/*
 * Setup the sensor, read the values and store them in dataCH0 and dataCH1
 * Additionally the values are calculated into lux and stored into the var. lux
 *
*/

bool getTSL2561(unsigned int &dataCH0, unsigned int &dataCH1, unsigned int &lux) {
    I2C TSL2561(PC6, PC7);
    int address = (0x39<<1);
    TSL2561.frequency(400000);
    char data0[2]={0};
    char data1[2]={0};
    //Setup the sensor by writing 0x03 (power on) to power control register (0x80)
    data0[0] = (0x80);
    data0[1] = 0x03;
    TSL2561.write(address, data0, 2);
    //wait till TSL is started and ready to reply
    if(!TSL_INIT)
    {
        wait_ms(500);
        TSL_INIT = true;
    }
    //Read out data from CH0
    data0[0] = (0x8C);
    data0[1] = 0x00;
    TSL2561.write(address, data0, 1);
    TSL2561.read(address, data0, 2);
    //debug("CHANNEL0: %d and %d\n",data0[0],data0[1]);
    //Read out data from CH1
    data1[0] = (0x8E);
    data1[1] = 0x00;
    TSL2561.write(address, data1, 1);
    TSL2561.read(address, data1, 2);
    //debug("CHANNEL1: %d and %d\n",data1[0],data1[1]);
    //Store both 8Bit Data register into one 16Bit value for CH0 and CH1
    dataCH0 = ( (uint16_t)data0[1] << 8 ) | data0[0];
    dataCH1 = ( (uint16_t)data1[1] << 8 ) | data1[0];
    //Calculate the lux value out of the two channel values
    getLux(dataCH0, dataCH1, lux);
    /*
    //lux = (float(lux));
    debug("Lux: %d\n", lux);
    */
    return(true);
}

#endif /* TSL2561_H */
XBee_send.h

/*
* XBee802_Send.h
* Created on: 03.05.2016
*      Author: Felix
*/

#ifndef XBEE802_SEND_H_
#define XBEE802_SEND_H_
#include "XBeeLib.h"
#include "mbed_assert.h"
#if defined(ENABLE_LOGGING)
#include "DigiLoggerMbedSerial.h"
using namespace DigiLog;
#endif

// Addresses may have to be changed having different radio modules.
#define REMOTE_NODE_ADDR64_MSB ((uint32_t)0x0013A200)
//#error "Replace next define with the LSB of the remote module's 64-bit address (SL parameter)"
#define REMOTE_NODE_ADDR64_LSB ((uint32_t)0x40693762)

// Replace next define with the remote module's 16-bit address (MY parameter)
#define REMOTE_NODE_ADDR16 ((uint16_t)0x1111)
#define REMOTE_NODE_ADDR64 UINT64(REMOTE_NODE_ADDR64_MSB, REMOTE_NODE_ADDR64_LSB)

using namespace XBeeLib;

//assembling the addresses to the used devices
const RemoteXBee802 remoteDevice64b = RemoteXBee802(REMOTE_NODE_ADDR64);
const RemoteXBee802 remoteDevice16b = RemoteXBee802(REMOTE_NODE_ADDR16);

//Define the radio module
XBee802 xbee = XBee802(RADIO_TX, RADIO_RX, RADIO_RESET, RADIO_RTS, RADIO_CTS, 38400);

void XBee_startup() {
  RadioStatus radioStatus = xbee.init();
  MBED_ASSERT(radioStatus == Success);
}

/*
* mbed example functions to send data
*/

void XBee_send_RGB(unsigned int blue, unsigned int green, unsigned int red) {
  char data[] = "Sensor Data RGB blue/green/red:");
  uint16_t data_len = strlen(data);
  xbee.send_data(remoteDevice16b, (const uint8_t *)data, data_len);
  memset(data,0,32);
  memcpy(data, (char *)&blue,2);
  data_len = strlen(data);
  xbee.send_data(remoteDevice16b, (const uint8_t *)data, data_len);
  memset(data,0,32);
  memcpy(data, (char *)&green,2);
  data_len = strlen(data);
  xbee.send_data(remoteDevice16b, (const uint8_t *)data, data_len);
  memset(data,0,32);
  memcpy(data, (char *)&red,2);
  data_len = strlen(data);
  xbee.send_data(remoteDevice16b, (const uint8_t *)data, data_len);
}
void XBee_send_light(unsigned int visib, unsigned int ir, unsigned int lux) {
    char data[] = "Sensor Data Light visible/IR/Lux:";
    uint16_t data_len = strlen(data);
    xbee.send_data(remoteDevice16b, (const uint8_t *)data, data_len);
    memset(data, 0, 32);
    memcpy(data, (char *)&visib, 2);
    data_len = strlen(data);
    xbee.send_data(remoteDevice16b, (const uint8_t *)data, data_len);
    memset(data, 0, 32);
    memcpy(data, (char *)&ir, 2);
    data_len = strlen(data);
    xbee.send_data(remoteDevice16b, (const uint8_t *)data, data_len);
    memset(data, 0, 32);
    memcpy(data, (char *)&lux, 2);
    data_len = strlen(data);
    xbee.send_data(remoteDevice16b, (const uint8_t *)data, data_len);
}

#endif /* XBEE802_SEND_H_ */
SD_Config.h

/*
 * SD_Config.h
 * Created on: 12.05.2016
 * Author: Felix
 */

#ifndef SD_CONFIG_H_
#define SD_CONFIG_H_

#include "mbed.h"

DigitalOut SD_enable(PA10);
FILE *light;
FILE *vibrate;

bool FILE_CREATED = false;

char dir_light[22] = "/sd/LightSensor";
char dir_vibration[22] = "/sd/VibrationSensor";
char filename_light[50];
char filename_vibration[50];
int file_number=0;

int read_file_number(char *dir) {
    int file_count = 0;
    DIR * dirp;
    struct dirent * entry;
    dirp = opendir(dir);
    while ((entry = readdir(dirp)) != NULL) {
        file_count++;
    }
    closedir(dirp);
    return file_count;
}

/*
 * Setup_SDcard enables the SD Card enable pin (PA10) on the µC and create a
 * file system for the SD card
 * A folder and the text file were the measurements are saved in is created. A
 * individual header vibration / light is added to the file.
 */

void Setup_SDcard_Light() {
    SD_enable = 1;
    SDFFileSystem sd(PB3, PB4, PB5, PB6, "sd"); // MOSI, MISO, SCLK, SSEL
    sd.disk_initialize();
    mkdir("/sd/LightSensor", 0777);
    file_number = read_file_number(dir_light);
    sprintf(filename_light, "/sd/LightSensor/Measurement_%d.txt", file_number);
    //debug("Filename: %s\n",filename_light);
    light = fopen("/sd/LightSensor/Measurement.txt", "w");
    light = fopen((const char*)filename_light, "w");
    fclose(light);
    if(light == NULL)
{ debug("Could not open file for write\n"); }

light = fopen((const char*)filename_light, "a");
fprintf(light, "Measurement Energy Harvesting - Light F.H. 2016 \r\n\r\n");
fprintf(light, "Time(h:m:s),RGB blue,RGB green,RGB red,Light Visible and IR,Light IR,Light Lux\r\n\r\n");
fclose(light);

void Setup_SDcard_Vibrate()
{
    SD_enable = 1;

    SDFileSystem sd(PB3, PB4, PB5, PB6, "sd"); // MOSI, MISO, SCLK, SSEL
    mkdir("/sd/VibrationSensor", 0777);
    file_number = read_file_number(dir_vibration);
    sprintf(filename_vibration, "/sd/VibrationSensor/Measurement_%d.txt", file_number);
    vibrate = fopen((const char*)filename_vibration, "w");
    fclose(vibrate);
    if(vibrate == NULL)
    { debug("Could not open file for write\n"); }
    vibrate = fopen((const char*)filename_vibration, "a");
    fprintf(vibrate, "Measurement Energy Harvesting F.H. 2016 \r\n\r\n");
    fprintf(vibrate, "Time(h:m:s),Vibration values \r\nTime,y \r\nTime,z \r\n\r\n");
    fclose(vibrate);
}

void write_light_to_file(char* zeit, unsigned int blue, unsigned int green, unsigned int red, unsigned int vis, unsigned int ir, unsigned int lux)
{
    SDFileSystem sd(PB3, PB4, PB5, PB6, "sd");
    light = fopen((const char*)filename_light,"a");
    fprintf(light, "%s,%d,%d,%d,%d,%d,%d \r\n", zeit, blue, green, red, vis, ir, lux);
    fclose(light);
}

void write_vibration_to_file(char* zeit, char* vibx, char* viby, char* vibz)
{
    SDFileSystem sd(PB3, PB4, PB5, PB6, "sd");
    vibrate = fopen((const char*)filename_vibration,"a");
    fprintf(vibrate, "%r\n%s\n%r\n%s\n%r\n%s\n\r\n", zeit, vibx, zeit, viby, zeit, vibz);
    fclose(vibrate);
}

#endif /* SD_CONFIG_H_ */
Timer0_Config.h

/**
 * Timer0_Config.h
 * Created on: 24.05.2016
 * Author: Felix
 */

#ifndef TIMER0_CONFIG_H_
#define TIMER0_CONFIG_H_
#include "mbed.h"
#include "em_rtc.h"
#include "em_emu.h"
#include "em_cmu.h"
#include "em_chip.h"
#include "em_letimer.h"

void systemClocksInit(void)
{
    // Enable LFRCO and wait for it to stabilize
    CMU_OscillatorEnable(cmuOsc_LFRCO, true, true);
    // Select LFRCO as clock source for LFACLK
    CMU_ClockSelectSet(cmuClock_LFA, cmuSelect_LFRCO);
    //Sets a clock divider to have the ability of long time periods
    CMU_ClockDivSet(cmuClock_LETIMER0, cmuClkDiv_4096);
    CMU_ClockEnable(cmuClock_HFPER, true);
    CMU_ClockEnable(cmuClock_CORELE, true);
    //ENABLE CLOCKS OF PERIPHERIA
    CMU_ClockEnable(cmuClock_GPIO, true);
    CMU_ClockEnable(cmuClock_USART0, true);
    CMU_ClockEnable(cmuClock_LEUART0, true);
    CMU_ClockEnable(cmuClock_LETIMER0, true);
}

void timer0_setup(int s)
{
    uint16_t period = 0;
    uint32_t letimer_clock = 0;
    LETIMER_Init_TypeDef init_letimer;
    systemClocksInit();
    //enable interrupts for LETIMER0
    NVIC_EnableIRQ(LETIMER0_IRQn);
    letimer_clock = CMU_ClockFreqGet(cmuClock_LETIMER0);

    /* Period calculation: t * (clk/presc)+0.5 (must be less or equal to 16 bit), where:
       t - required time in seconds
       clk - timer clock
       presc - timer clock prescaler, 1 - in LETIMER
       0.5 - rounding period to up */
    period = (uint16_t) 1 * (letimer_clock/1) +0.5; //we will get 32768/4096
    uint32_t timertop = period * s;
    init_letimer.enable = true; /* Start counting when init completed. */
    init_letimer.debugRun = false; /* Counter shall not keep running during debug halt. */
    init_letimer.rtcComp0Enable = false; /* Don't start counting on RTC COMP0 match. */
    init_letimer.rtcComp1Enable = false; /* Don't start counting on RTC COMP1 match. */
init_letimer.comp0Top = true; /* Load COMP0 register into CNT when counter underflows. COMP0 is used as TOP */
init_letimer.bufTop = false; /* Don't load COMP1 into COMP0 when REP0 reaches 0 */
init_letimer.out0Pol = 0; /* Idle value for output 0 */
init_letimer.out1Pol = 0; /* Idle value for output 1 */
init_letimer.ufos0 = letimerUF0ANone; /* PWM output on output 0 */
init_letimer.ufos1 = letimerUF0ANone; /* Pulse output on output 1 */
init_letimer.repMode = letimerRepeatFree; /* Count until stopped */
LETIMER_IntEnable(LETIMER0, LETIMER_IF_UF); /* LETIMER counts down, so we set underfl. irq*/
LETIMER_CompareSet(LETIMER0, 0, timertop); /* Setting up the timer TOP value, to count from it */

LETIMER_Init(LETIMER0, &init_letimer); /* LETIMER initialization and starting */

void LETIMER0_IRQHandler()
{
  uint32_t flags;
  // all interrupt flags
  flags = LETIMER_IntGet(LETIMER0);

  // if interrupt is underflow interrupt
  if((flags & LETIMER_IF_UF))
  {
    // clear it
    LETIMER_IntClear(LETIMER0, LETIMER_IF_UF);
  }
}
#endif /* TIMER0_CONFIG_H_ */