Laboratory Project

EL024A Sensor Networks

Group 7

Petter Haugen
Jørgen Ryther Hoem
Erlend Røed Myklebust

Spring 2016

Mittuniversitetet
MID SWEDEN UNIVERSITY

NTNU – Trondheim
Norwegian University of
Science and Technology

Introduction

Sensors are surrounding us in our everyday life, at work, leisure and a variety of other activities.

Implementation of wireless technology in sensor systems expands the scope of sensor usage in new applications, making it possible to perform measurements in locations that were previously considered inaccessible due to communication and power restrictions. These elements are still considered challenges, but can be partly handled by using wireless sensor networks working on embedded platforms.

In this project, we will develop a wireless sensor system, using embedded platform toolkits to perform event driven measurements of light and temperature. The toolkits in question are the Waspmote platform by Libelium and the XBee wireless module. With these, we will be able to implement a small-scale application whereby utilizing the wireless network capabilities of the toolkits, various sensor data can be communicated between different units, or nodes.

Table of Contents

Chapter 1 Initial work

1.1 Task

The main goal of this project is to develop a small wireless sensor system consisting of three nodes using the IEEE 802.15.4 wireless standard in conjunction with the XBee protocol. The equipment used in this project, is the Waspmote system from Libelium, which is a sensor and network development toolkit based around the Arduino environment.

The tasks for this project are divided into four levels, starting from an intermediate level to a more advanced level.

The first level begins with transmitting light sensor data from one node to a second node. While the second level introduces event detection to the light sensor, using interrupts.

The third level is based upon the previous levels, but amends the network with a third node. The first node in the network serves as the main sensor node, which is triggered by a light sensor event, and then transmits the sensor data to the second node. The second node is then adding temperature data to the payload and forwards this to the third node.

The fourth and last level uses the light sensor event to make the second node initiate an average temperature measurement. When the event occurs a second time, the measurement stops and the data is sent to the third and last node in the network.

Although we have been developing and testing solutions for all four levels, level three and four have been subject to most of our attention and efforts. We are justifying this because the higher levels include elements from the first two levels, and additionally serve to prove a greater understanding of the course material. Furthermore, for the sake of the length of this rapport and because it summarizes the project as a whole, we will only focus on explaining our level four implementation in this rapport.

1.2 Solution overview

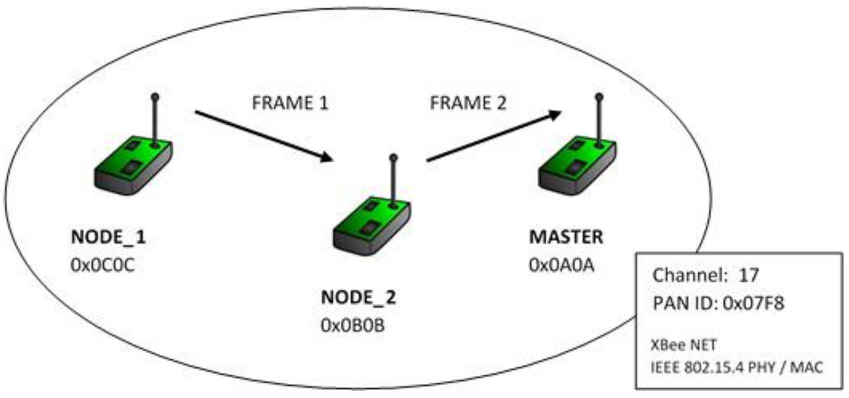


Figure 1. Network diagram for our level 4 implementation

Figure 1 illustrates the network in our level 4 implementation of the wireless sensor system. In short, it describes a small, confined, point-to-point topology, where communication is configured to channel 17 (0x11) of the IEEE 802.15.4 2.4 GHz band, with network ID (PAN ID) 0x07F8 and no encryption. All in accordance with the laboratory project task description.

The transmission function provided by the Waspmote API requires that the payload be sent either as a single 2-byte integer or as a character array. The method using character array is defining the payload size by the first zero byte in the array. The data transferred between the nodes are therefore not sent as binary data but are instead converted and concatenated into one string before transmission.

1.2.1 Node 1

Node 1 is the first node in the chain. This node consists of two light sensors and has the network address “0C0C” and the string identifier “NODE_1”.

The main task for Node 1 is to send a notification packet to Node 2 when an event is detected. This event is triggered when one of the light sensors reach a certain threshold level. The packet payload is a 2-byte long string containing an integer value followed by a terminating zero.

Name	Trigger	End
Frame	1	2
Data Type	char	char
Length	1	1

Table 1. Paylaod frame format 1 (FRAME 1)

1.2.2 Node 2

The second node in the network is Node 2 and has the network address “0B0B” and the string identifier “NODE_2”. This node performs multiple operations where the main task is to measure the temperature whenever the node is told to do so.

When Node 2 receives a packet where the first byte of the payload is equal to the character “1” it starts measuring the temperature periodically using the built-in temperature sensor. This continues until the node receives another identical packet containing “1”. The averaged temperature data is then transmitted to the Master node. The packet payload also contains information such as the duration and the timestamp of the measurement in seconds followed by a terminating zero.

Name	Temperature					Duration					Timestamp				End
Frame	1	2	3	4	5	6	7	8	9	10	11	12	…	42	43
Data Type	string					string					string				char
Length	5					5					32				1

Table 2. Payload frame format 2 (FRAME 2)

1.2.3 Master node

The Master node is the third and last node in the sensor network. This node has the address “0A0A” and the string identifier “MASTER”. The main task for this node is to receive packets from Node 2 and present the payload in a user-friendly format.

The received data from Node 2 is parsed, converted and displayed in such a manner that the user can see when the temperature measurement started, the duration of the measurement and the average temperature in Celsius.

Chapter 2 Implementation in detail

2.1 Node 1 – Sensor node

Node 1 is the first link in the sensor system and consists of a Waspmote v.1.1 board, an XBee wireless module, an Events sensor board and two PDV-P9203 photoconductive sensors. As mentioned before, its purpose is to communicate to Node 2 whenever the sensors detect luminosity either above or below a set of thresholds.

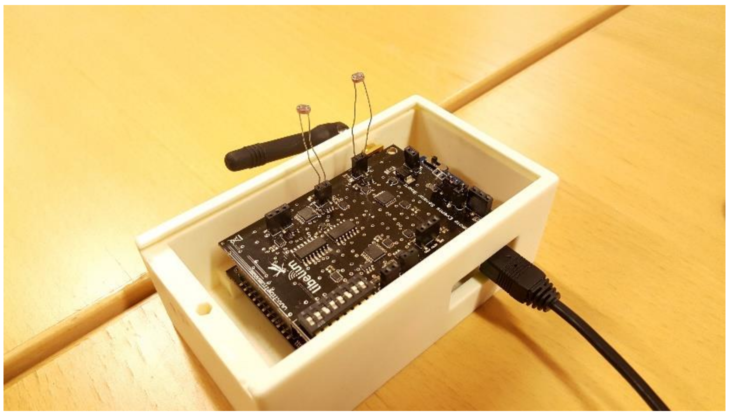


Figure 2. Picture of Node 1 and all its hardware

Figure 3 illustrates Node 1’s firmware in form of a flow diagram.

The program begins with a series of defines and global variable declarations. Most of which are used for setting network related parameters later on in the code, but some are related to other configurations like sensor thresholds and sensor socket selection. The XBee packet payload, named data, is also declared and given a value in this section of the code. Note that this is the only evaluation of data in the entire code!

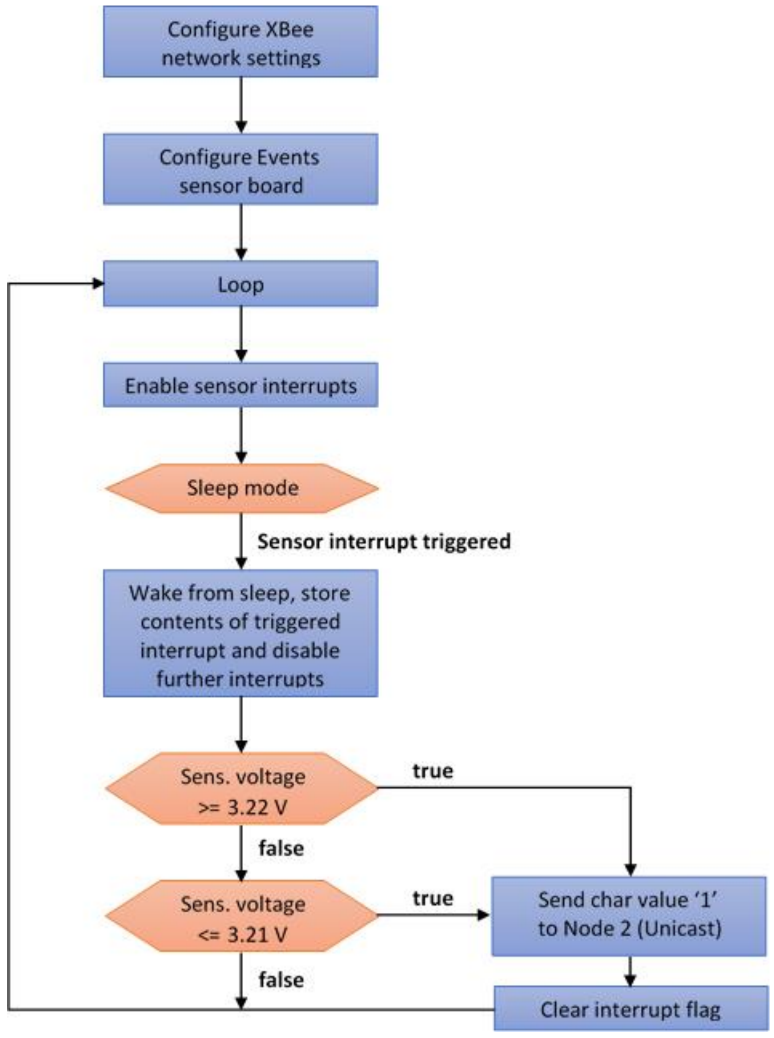

Configure XBee network settings
Configure Events sensor board
Loop
Enable sensor interrupts
Sleep mode
Sensor interrupt triggered
Wake from sleep, store contents of triggered interrupt and disable further interrupts
Sens. voltage >= 3.22 V
true
false
Sens. voltage <= 3.21 V
true
false
Send char value '1' to Node 2 (Unicast)
Clear interrupt flag

Figure 3. Flow diagram describing Node 1's firmware

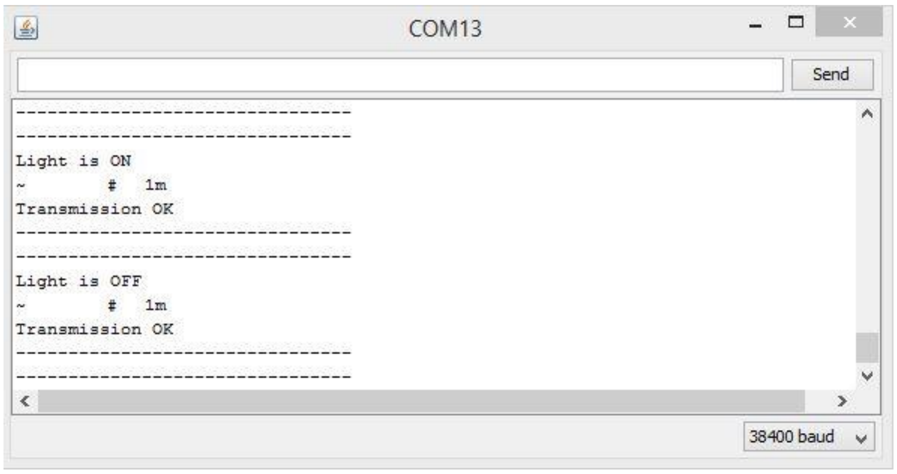
Next, the program commences the setup()-function, in which a series of function calls are made to configure and activate the connected XBee module. Parameters such as the network PAN identifier, IEEE 802.15.4 channel, the node's own 16-bit short address (referred to as source address) and network encryption level are all configured and stored in non-volatile memory. In the second part of the setup()-function, the Events sensor board is activated and the two sensor sockets are enabled and set to generate interrupts whenever their respective threshold values are exceeded. Sensor socket 3 on the Events sensor board is hardware configured to generate an interrupt when the measured sensor voltage rises above the threshold from a lower value (rising edge). The opposite is true for sensor socket 6, i.e. an interrupt is generated when the measured sensor voltage falls below the threshold from a higher value (falling edge). In other words, the sensor in socket 3 detects when the light switches from off to on, while the sensor in socket 6 detects when the light switches from on to off, in our application.

Succeeding the setup()-function, is the loop()-function, otherwise known as the main loop. It is the meat of the program, in the sense that this is where the program execution spend most of its time. Whenever the program execution reach the end of this function, it simply jumps back to the first line of the function and starts all over again, hence the name loop. For every loop-through, a start string is written to the UART terminal, followed by a general enabling of sensor interrupts from the Events sensor board, and then a low power mode, called sleep mode, is activated on the Waspmote board. In sleep mode, the Waspmote's microcontroller powers down and seizes all processing, only keeping power to its volatile memory. This has the effect of halting the code execution. Note that the Events sensor board is not affected by the sleep mode, and remain capable of generating sensor interrupts. When a sensor interrupt occurs, the Waspmote wakes up and resumes code progression from where it left off.

The first thing the program does after waking up, is re-enabling the XBee module in order to reestablish UART communication with the Waspmote. Next, sensor interrupts are disabled to prevent untimely interrupts from breaking the code progression. Then, the contents of the interrupt register is stored in an integer variable called SensorEvent.intFlag. This variable can now be used to determine which sensor caused the current interrupt. If the socket 3 sensor caused the interrupt, the message "Light is ON" is printed to the terminal window, while if socket 6 caused the interrupt, the message "Light is OFF" is printed to the terminal window. In either case, the function transmit_package() is called and the intFlag is cleared for future use.

The function transmit_package() builds an XBee-packet with ID 0x07, that is sent UNICAST from Node 1's source address (0x0C0C) to the destination address 0x0B0B (Node 2's source address) with payload data. As mentioned before, data is set to a fixed value (char '1') at the start of the program and is never changed after this. It is also worth mentioning that the packet payload is always data, regardless of which sensor interrupt occurred. We have chosen this approach because it simplifies the control structure needed in Node 2.

Figure 4 depicts the terminal window output for an example run-through where a sensor socket 3 interrupt is triggered and subsequently followed by a sensor socket 6 interrupt a few seconds later.



```
--------------------------------
--------------------------------
Light is ON
~        #   1m
Transmission OK
--------------------------------
--------------------------------
Light is OFF
~        #   1m
Transmission OK
--------------------------------
--------------------------------
```

Figure 4. Node 1's terminal output for the example run-through

2.2 Node 2 – Intermediary node

Node 2 is the middle link in the chain and consists simply of a Waspmote v.1.1 board and an XBee module. As with Node 1, Node 2 is also a sensor node, but unlike Node 1, its sensor readings will actually be processed and communicated for further use.

Figure 5 illustrates Node 2’s firmware in form of a flow diagram. Please note that this is a simplified flow diagram, where some steps and control conditions have been omitted for the sake of readability.

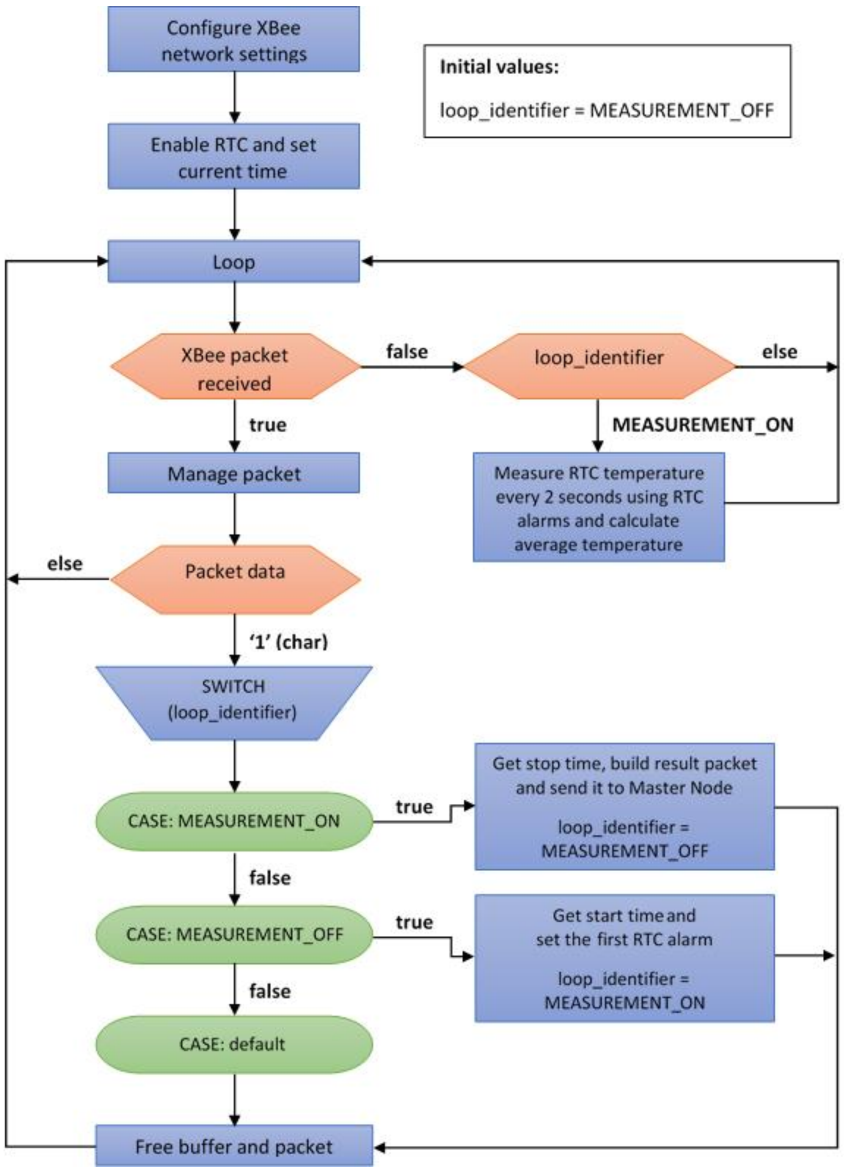

Configure XBee network settings
Initial values:
loop_identifier = MEASUREMENT_OFF
Enable RTC and set current time
Loop
XBee packet received
false
loop_identifier
else
true
MEASUREMENT_ON
Manage packet
Measure RTC temperature every 2 seconds using RTC alarms and calculate average temperature
else
Packet data
‘1’ (char)
SWITCH (loop_identifier)
CASE: MEASUREMENT_ON
true
Get stop time, build result packet and send it to Master Node
loop_identifier = MEASUREMENT_OFF
false
CASE: MEASUREMENT_OFF
true
Get start time and set the first RTC alarm
loop_identifier = MEASUREMENT_ON
false
CASE: default
Free buffer and packet

Figure 5. Flow diagram describing Node 2's firmware

The program begins with a series of defines and global variable declarations. Some are identical to those in Node 1’s firmware, but most of them are different. Node 2’s operation involves several data processing and control elements; hence, it requires more global variables than Node 1. One of the said variables is especially important, loop_identifier. It governs the majority of the code flow by altering between its two valid states, MEASUREMENT_ON and MEASUREMENT_OFF. Initially it is set to MEASUREMENT_OFF.

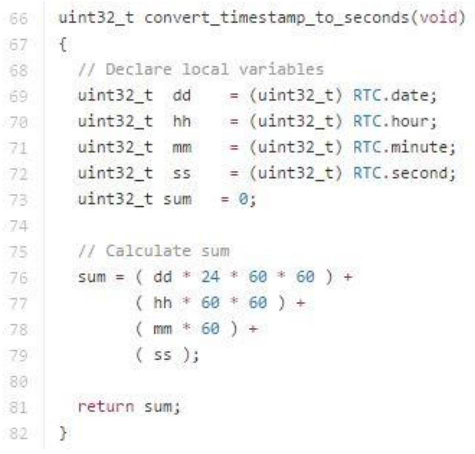
Following the defines section, comes the setup()-function. The first half of which is identical to that of Node 1, as described in the previous sub-chapter, the only exception being the configuration of a different source address (0x0B0B for Node 2). The other half however, is very different. Here the Real Time Clock (RTC) on the Waspmote board is activated and set to start at a specific date and time.

Finally, the main loop commences. Explaining this part of the code on a line-by-line basis is pointless because the flow is entirely dependent on the current state of the control structure. Figure 5 provides a much clearer view on what is going on. However, a condensed overview of the main loop's functionality is both possible and necessary to present in from of text.

Concisely, whenever an XBee packet with payload '1' (char) is received from Node 1, one of two things can happen:

- If an average temperature measurement is not currently running, the program will get a start time from the RTC and begin measuring the RTC temperature every 2 seconds, using RTC alarms. For each measurement, an average temperature is re-calculated based on the current and previous temperatures.
- If an average temperature measurement is currently running, the program will get a stop time from the RTC, stop all measurements, build an XBee packet containing the resulting data and send the packet to the Master Node.

As mentioned in subchapter 1.2.2, we have chosen a very specific payload format for the results packet Node 2 sends to the Master Node. 5 bytes for the average temperature (4 digits and a decimal point), 5 bytes for the measurement duration in seconds and finally up to 32 bytes for the start time timestamp. We have chosen to send the measurement duration in seconds plus the start time timestamp instead of sending both start time and stop time timestamps. This results in a reduction of the XBee packet payload size, thus conserving energy. At least slightly. It does however require some software trickery. When an average temperature measurement is started and the program gets the start time from the RTC, the function convert_timestamp_to_seconds() is called. See Figure 6.

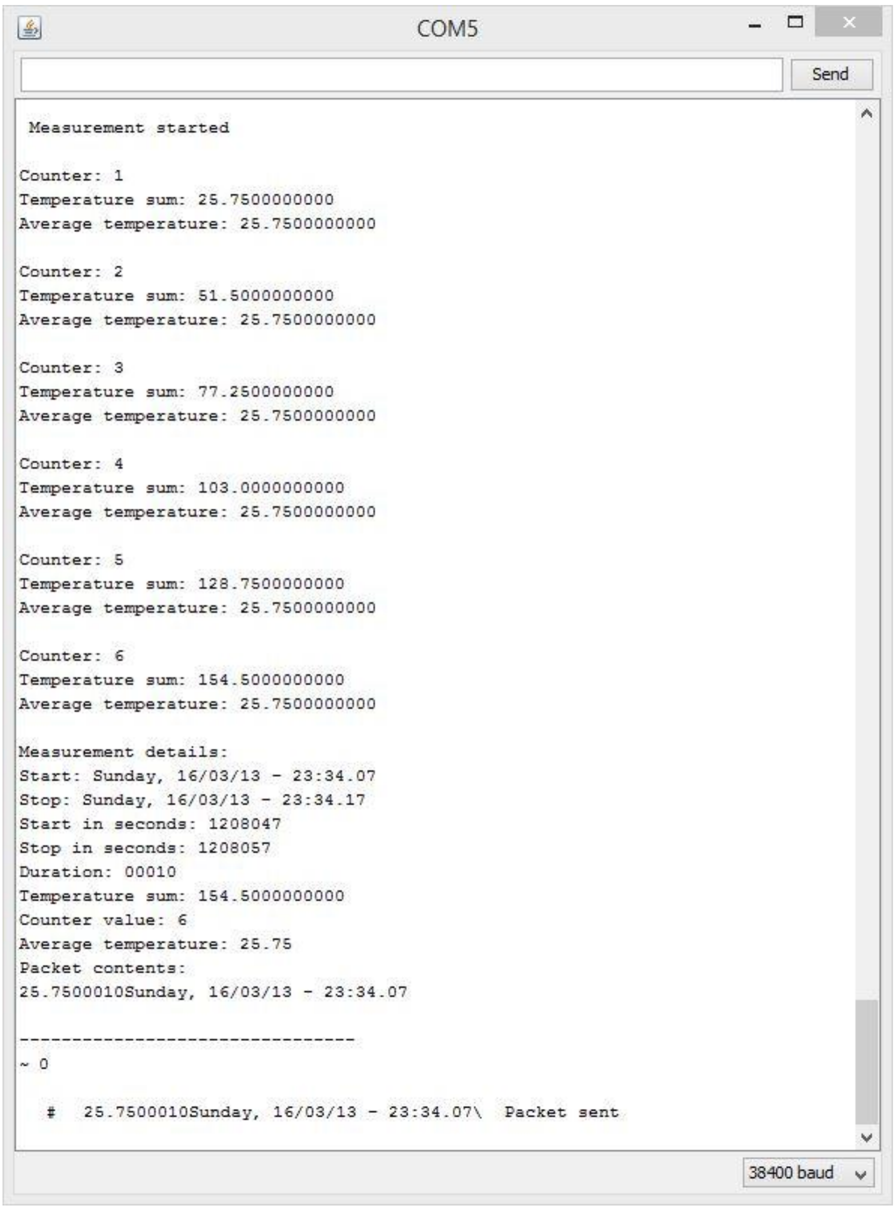


```
uint32_t convert_timestamp_to_seconds(void)
{
  // Declare local variables
  uint32_t  dd    = (uint32_t) RTC.date;
  uint32_t  hh    = (uint32_t) RTC.hour;
  uint32_t  mm    = (uint32_t) RTC.minute;
  uint32_t  ss    = (uint32_t) RTC.second;
  uint32_t sum   = 0;

  // Calculate sum
  sum = ( dd * 24 * 60 * 60 ) +
        ( hh * 60 * 60 ) +
        ( mm * 60 ) +
        ( ss );

  return sum;
}
```

Figure 6. Code function for converting timestamp to sum of seconds

This function simply imports the various RTC-fields generated when the start time was fetched from the RTC and converts them to a sum of seconds. Another conversion is then performed at the end of an average temperature measurement, and the difference between the two sums are calculated, yielding a measurement duration in seconds.

All the while throughout an average temperature measurement, the terminal window is updated for each new RTC temperature measurement. Figure 7 depicts Node 2's terminal window output for the same example run-through present in the Node 1 sub-chapter. Here we can see all the relevant data for the entire measurement, including start and stop time converted to number of seconds, duration in seconds, average temperature and more.



```
 Measurement started

Counter: 1
Temperature sum: 25.7500000000
Average temperature: 25.7500000000

Counter: 2
Temperature sum: 51.5000000000
Average temperature: 25.7500000000

Counter: 3
Temperature sum: 77.2500000000
Average temperature: 25.7500000000

Counter: 4
Temperature sum: 103.0000000000
Average temperature: 25.7500000000

Counter: 5
Temperature sum: 128.7500000000
Average temperature: 25.7500000000

Counter: 6
Temperature sum: 154.5000000000
Average temperature: 25.7500000000

Measurement details:
Start: Sunday, 16/03/13 - 23:34.07
Stop: Sunday, 16/03/13 - 23:34.17
Start in seconds: 1208047
Stop in seconds: 1208057
Duration: 00010
Temperature sum: 154.5000000000
Counter value: 6
Average temperature: 25.75
Packet contents:
25.7500010Sunday, 16/03/13 - 23:34.07

--------------------------------
~ 0

   #  25.7500010Sunday, 16/03/13 - 23:34.07\  Packet sent
```

Figure 7. Node 2's terminal output for the example run-through

2.3 Master node

The Master Node is the end point of our sensor system and is the one responsible for presenting the measurement results to the user. As with Node 2, it simply consists of a Waspmote v.1.1 board and an XBee module.

Figure 8 illustrates the Master Node's firmware in form of a flow diagram.

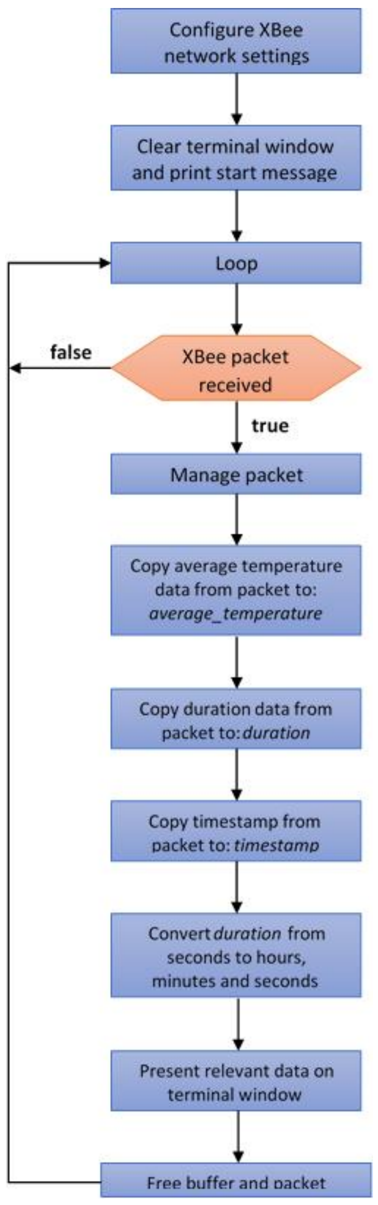
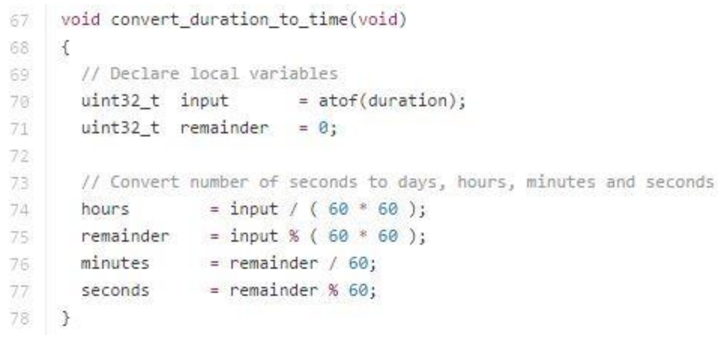

Configure XBee network settings
Clear terminal window and print start message
Loop
false
XBee packet received
true
Manage packet
Copy average temperature data from packet to: average_temperature
Copy duration data from packet to: duration
Copy timestamp from packet to: timestamp
Convert duration from seconds to hours, minutes and seconds
Present relevant data on terminal window
Free buffer and packet

Figure 8. Flow diagram describing the Master Node's firmware

Compared to the firmware of Node 1 and 2, the Master Node's firmware is deceptively simple.

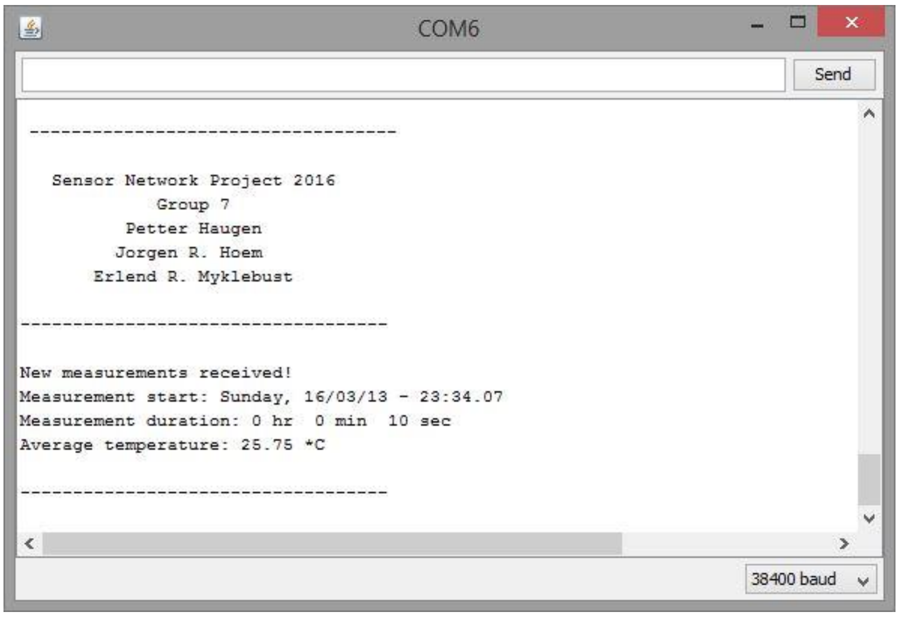
The program starts with the usual defines and global variable declarations, and the first half of the setup()-function is again identical to the other ones, except for configuring a different source address (0X0A0A). The second half of the setup()-function however, and indeed most of the main loop, is all about the presentation of data.

The main loop starts by checking if an XBee packet has been received and if yes, then treats the packet. From the payload, the average temperature, measurement duration in seconds and measurement start time timestamp is extracted and placed in separate variables and strings. In order to make sense of the measurement duration, more software trickery is needed. This is done by calling the function convert_duration_to_time(). See Figure 9. Here, the duration in seconds is split into hours, minutes and seconds. Note that there is no days value, nor is there need for one. This is because the duration in seconds value that is received in the packet payload is only represented by 5 decimal digits, and hence will overflow if the duration exceeds approximately 28 hours (99999 seconds ≈ 27 hours and 46 minutes).



```
void convert_duration_to_time(void)
{
  // Declare local variables
  uint32_t  input      = atof(duration);
  uint32_t  remainder  = 0;

  // Convert number of seconds to days, hours, minutes and seconds
  hours       = input / ( 60 * 60 );
  remainder   = input % ( 60 * 60 );
  minutes     = remainder / 60;
  seconds     = remainder % 60;
}
```

Figure 9. Code function for converting seconds to hours, minutes and seconds



```
-----------------------------------

  Sensor Network Project 2016
            Group 7
         Petter Haugen
        Jorgen R. Hoem
      Erlend R. Myklebust

-----------------------------------

New measurements received!
Measurement start: Sunday, 16/03/13 - 23:34.07
Measurement duration: 0 hr  0 min  10 sec
Average temperature: 25.75 *C

-----------------------------------
```

Figure 10. The Master Node's terminal output for the example run-through

With the duration conversion completed, all that is left is to present the results to the terminal window. Unlike the other nodes, the Master Node is the only one that “require” a PC connection for presenting data to the terminal window. While it certainly is possible, and sometimes beneficial to see the terminal output for the other nodes, it is not a requirement for their operation. Node 1 and 2 will happily perform their duties without a USB connection. However, if the Master Node is also disconnected, all data will be lost, since it does not incorporate any form of data logging outside of the terminal window memory.

Figure 10 depicts the Master Node’s terminal window output for the same example run-through present in the two previous sub-chapters. Here we can clearly see that the duration has been communicated and converted successfully, since it is the same as in Figure 7.
The same is also true for the average temperature value and the measurement start time timestamp.

Chapter 3 Discussion and conclusion

Integers are known to be more bandwidth- and power efficient than ASCII strings in wireless transmission schemes, because they are generally shorter in length (fewer bytes needed to represent them). The payload data should preferably have been transmitted as an array of integers, but due to limitations expressed in chapter 1.2, we were not able to execute such an action. Because of the payload size definition, any packet containing the value zero would have been terminated at the zero point value, resulting in a possible loss of critical packet data. A possible solution to this problem could have been to format the data to a given integer value instead of the value zero, prohibiting pre-termination of the packet size. This method would not be flawless, since the data representation on the receiver side would not be equal to the measured integer value. The nodes would also use more power to analyze and manipulate the data.

The timestamps supported by the Waspmote API are defined as up to 32 character length arrays. During the implementation, we experienced that the timestamp length varied according to the day of the week set in the RTC clock. With a varying bit length, interpreting and handling the data needed for event calculation proved to be hard. The solution was to convert the date- and time information from the RTC clock directly into seconds, and execute the event calculation locally on node 2. A Unix based time stamp could have solved this issue, providing integers with set length, easier to interpret and convert at the data reception.

To improve the power efficiency in the wireless sensor system, we would have preferred to implement sleep modes in all of the sensor nodes. This proved to be impossible for our specific application due to restrictions within the Waspmote hardware and API. None of the sleep modes available in the Waspmote energy system supports idle listening for data traffic, unlike other embedded platforms such as the Atmel ATmega256RFR2. Since the event detection and data transmission is executed at non-deterministic times, this would have been the only recommended sleep mode for node 2 and the master node in our network.

Wireless networks holds a large volume of traffic, and being able to discriminate specific traffic is important for the sake of robustness and power management within each sensor node. The final code contains a limited control structure for node 2, but no further control of the received data-structure is implemented in the master node. Corrupted data from Node 2 or possibly any other data addressed to the master node could corrupt or disrupt the measured results. Letting the master check the data content and control if there is a decimal point in payload byte 3 could be implemented to solve this issue.

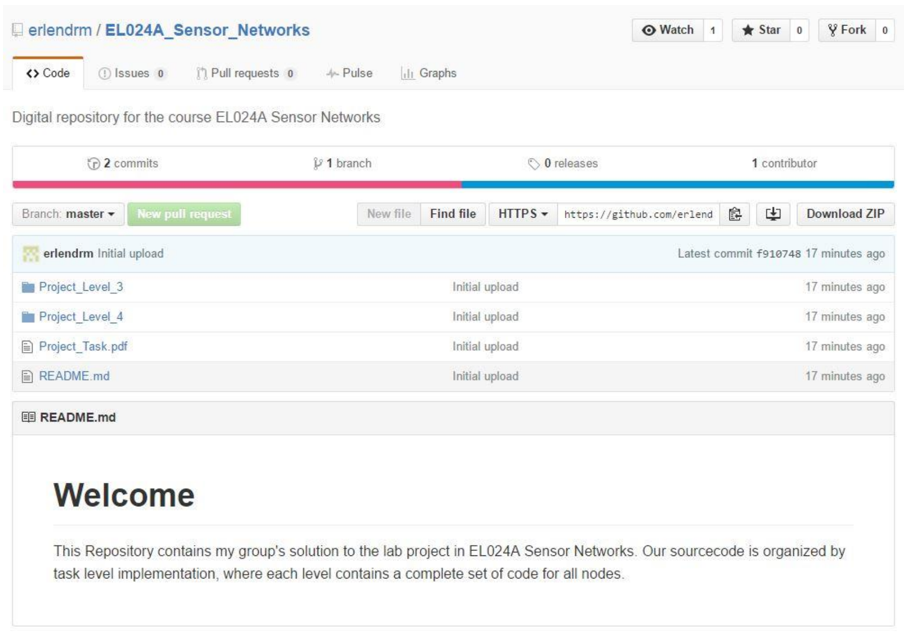
An effective method for power management in wireless sensor networks is to allocate only critical tasks to nodes who have limited power, and let nodes with a constant power supply perform heavier calculations. The specification of level 4 explains that computing should take place locally in Node 2, meaning that the battery powered node will perform most of the heavy calculations.

In this project, we have constructed a fully functioning wireless sensor system, using the Waspmote embedded platform, consisting of three nodes and a sensor board. The system is optimized within the limits of the Waspmote platform, and serves its purpose easily and elegantly: Event-based detection of light values, wireless control and transmission of data from sensor nodes and power management in a limited scale.

END

Appendix

A1: Digital repository (GitHub)



Digital repository for the course EL024A Sensor Networks

erlendrm Initial upload — Latest commit f910748 17 minutes ago

Project_Level_3	Initial upload	17 minutes ago
Project_Level_4	Initial upload	17 minutes ago
Project_Task.pdf	Initial upload	17 minutes ago
README.md	Initial upload	17 minutes ago

README.md

Welcome

This Repository contains my group's solution to the lab project in EL024A Sensor Networks. Our sourcecode is organized by task level implementation, where each level contains a complete set of code for all nodes.

All our source code for this project, in addition to the project task and a .pdf-version of this report has been uploaded to the GitHub repository "EL024A_Sensor_Networks". Here, the files are organized by task level implementation, where each level folder contains a complete set of code for all nodes.

If you wish to read any of our code, simply navigate to a desired .pde-file and click it to read, using the integrated GitHub code viewer. In our opinion, the code is easier to read this way, as opposed to reading it in the Waspmote IDE. You can also one-click-download a zip-version of the entire repository, in case you want to test our code for yourself.

Please use the following static link to reach the repository:

github.com/erlendrm/EL024A_Sensor_Networks