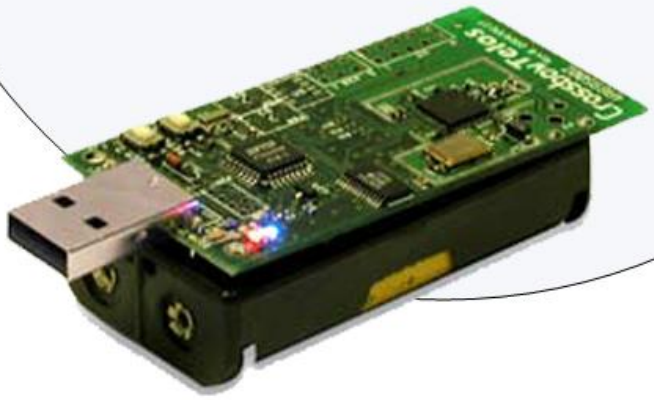
Building and adapting a multihop Wireless Sensor Network

Peter De Cauwer
Tim Van Overtveldt
3EA-ICT



CONTENTS

multihoposcilloscope 5

What? 5

Tinyos 6

Operating System 6

Installation 7

program multihop oscilloscope 8

header file 8

multihoposcilloscopeappc file 9

multihoposcilloscopec 10

module part of the file 10

implementation part of the file 11

Multi Hop/Sens oscilloscope (peter) 17

Introduction 17

MultiHopOscilloscopeApp.nc 17

MultiHopOscilloscopeC.NC 18

Interfaces 18

Preprocessor 18

Declaratie Variabelen 18

Event Timer.fired 20

event Send.SendDone 21

Conclusion 22

Bibliography 23

multi hop/sens oscilloscope 23

introduction 23

how does it work 23

implementation file 23

declarion 23
event boot.booted 23
event timer.fired 24
event readL.readdone 25
event readl.readdone 25
RSSI Testing 27
Introduction 27
Anchor node 27
Functional description 27
Anchornode.H 27
AnchornodeAppC 28
AnchorNodeC 28
Preprocessor 28
interfaces 28
conclusion 31
BlindNode 31
Functional description 31
Blindnode.h 31
BlindnodeAppC 31
BlindNodeC 32
RSSI readings 34
INtroduction 34
Method 34
octopus synchronous 35
Application octopus 35
bools 35
Timer.fired 35

collection send 36
visualization of nodes with rssi 37
Introduction 37
header file 37
Configuration file 37
module file 38
module part 38
implementation part 39
Visualization and direction with the use of RSSI 44
introduction 44
what does it do? 44
module file 45
implementatin file 45
module part 45
implementation part 45

MULTIHOPOSCILLOSCOPE

WHAT?

This application collects data and is installed on the TelosB node. It takes samples from the chosen sensor with a certain period and broadcasts a message every 5 readings. The default sampling rate is 4Hz (can be altered in the java application).

The advantage of the application is that it uses multihop to expand the range of the network.

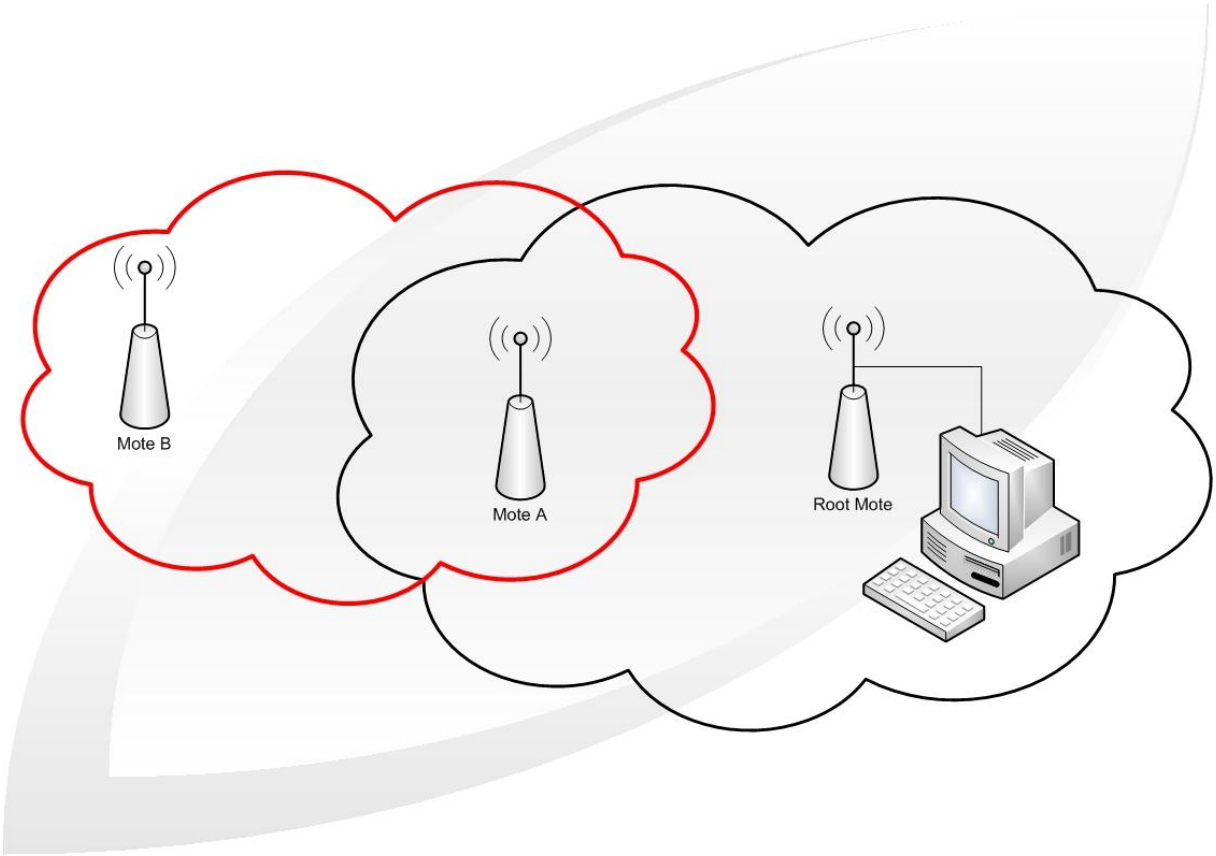


Figure 1: A multihop network

A node is connected to the computer through the serial USB port. The node is configured as the root of the network. The network makes use of Collection, the network is symbolized as een tree with the root at the base, where the data is sent to. Node A is in range of node B and the root node. Node B is out of range from the root node. The solution is that node A acts as a bridge between the root and node B.

Figure 2: TelosB node

TINYOS

TinyOS is a operating system which is used for embedded systems for wireless sensor networks. De language for programming nodes is NesC. NesC describes a row of events (tasks and processes) and is a variant off C.
An application exist out of 1 or more components which are linked to each other to come to the actual program. A component declares and uses interfaces. These interfaces are the only way to approach the component. An interface contains functions and commands which needs to be declared. Events are also functions with an interface, but these need to be declared by the user of the interface. It is possible to declare/use multiple interfaces or to use multiple instantiations of the same interface.

OPERATING SYSTEM

Het programmeren van de TelosB motes doen we via Linux: Xubuntos. XubunTOS simplifies the installation of TinyOS by using a Linux live CD. The bootable live CD contains a working TinyOS environment and offers the option to perform a full installation. XubunTOS is built from Xubuntu and TinyOS 2.x Debian packages (plus the TinyOS 1.x CVS repository). After installation, Debian's APT package manager can keep your software up-to-date.

xubuntos

xubuntu + TinyOS

INSTALLATION

In the directory off ur application (/apps/MultihopOscilloscope) we open a terminal and type:

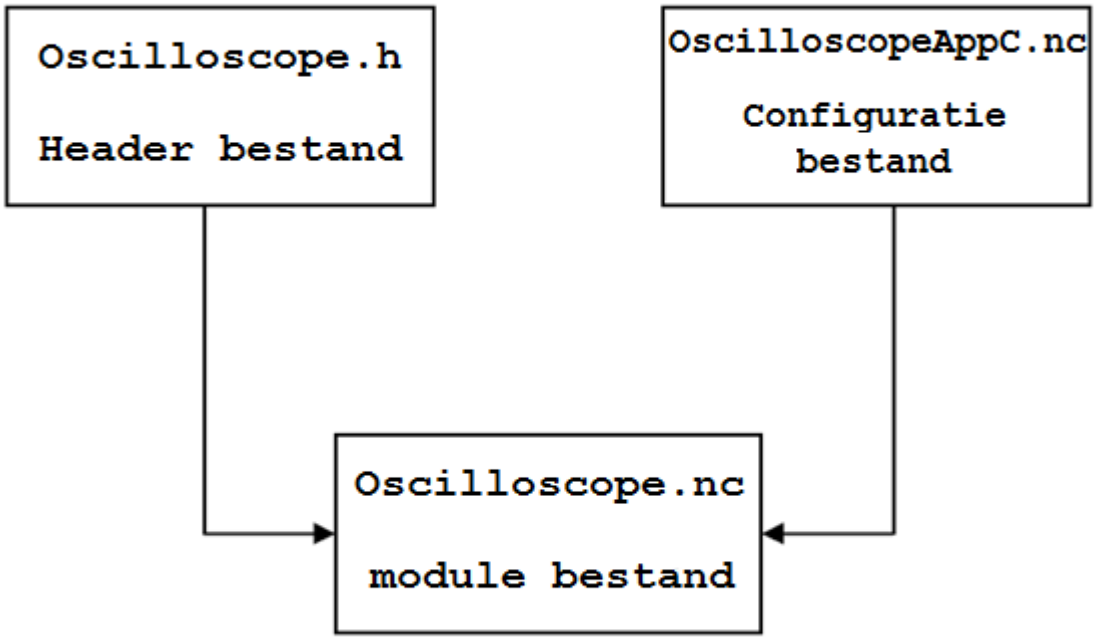
- *Make telosb:* De TinyOS application is compiled from his directory
- *Make telosb reinstall,500:* This compiles an image from the application with the ID 500, which is comatible with the telosb platform

In the subdirectory (/apps/MultihopOscilloscope/java) we start the serialforwarder tool. The serialforwarder is a packet source, thus a communication medium which can be used by the application to send and receive packts to and from a node.

We type:

1. make
2. java net.tinyos.sf.SerialForwarder -comm serial@/dev/ttyUSB0:telosb
3. ./run

PROGRAM MULTIHOP OSCILLOSCOPE



The application MultihopOscilloscope contains mainly 3 files:

- Oscilloscope.h: the header file has a struct for the storage of data
- OscilloscopeAppc.nc: the configuration file is used as a source file for the nesC compiler to generate a executable file. This file can use and supply interfaces. nesC uses arrows to connect interfaces with each other
- Oscilloscope.nc: the module file contains the implementation of the application. It may use every command from the interface, which it implements.

HEADER FILE

```
#ifndef MULTIHOP_OSCILLOSCOPE_H
#define MULTIHOP_OSCILLOSCOPE_H
enum {
  NREADINGS = 5,
  DEFAULT_INTERVAL = 1024,
  AM_OSCILLOSCOPE = 0x93
};

typedef nx_struct oscilloscope {
  nx_uint16_t version;
  nx_uint16_t interval;
  nx_uint16_t id;
  nx_uint16_t count;
  nx_uint16_t readings[NREADINGS];
} oscilloscope_t;
#endif
```

The “define” command is used when we import this file in other files. The name of this header file is added to a list which the compiler uses to make relations between the classes in the different.
The “ifndef” line prevents that the header file will appear multiple times in that list.
Next we declared a couple of variables in the program:

- NREADINGS is the number of sensor reading per sent message.
- DEFAULT_INTERVAL contain the interval time between the transmitted and received packets.
- AM_OSCILLOSCOPE contains an ID number for the visual java program “oscilloscope” on the PC which receives the packets.

De struct oscilloscope is used in the program to store the data and it contains some variables:

- VERSION is a version number, this can be used when the root sends a assignment to a node
- INTERVAL is used to configure a timer in the main program
- ID servers as identification for each node, to know which reading belongs to which
- COUNT contains the sample rate
- READINGS[NREADINGS] is an array , which stores the readings off the sensors

MULTIHOPOSCILLOSCOPEAPPC FILE

```
configuration MultihopOscilloscopeAppC {}
implementation
{
  components MainC, MultihopOscilloscopeC, LedsC, new TimerMilliC(),
    new HamamatsuS1087ParC() as Sensor;

  //MainC.SoftwareInit -> Sensor;

  MultihopOscilloscopeC.Boot -> MainC;
  MultihopOscilloscopeC.Timer -> TimerMilliC;
  MultihopOscilloscopeC.Read -> Sensor;
  MultihopOscilloscopeC.Leds -> LedsC;


  components CollectionC as Collector,
    ActiveMessageC,
    new CollectionSenderC(AM_OSCILLOSCOPE),
    SerialActiveMessageC,
    new SerialAMSenderC(AM_OSCILLOSCOPE);

  MultihopOscilloscopeC.RadioControl -> ActiveMessageC;
  MultihopOscilloscopeC.SerialControl -> SerialActiveMessageC;
  MultihopOscilloscopeC.RoutingControl -> Collector;

  MultihopOscilloscopeC.Send -> CollectionSenderC;
  MultihopOscilloscopeC.SerialSend -> SerialAMSenderC.AMSend;
  MultihopOscilloscopeC.Snoop -> Collector.Snoop[AM_OSCILLOSCOPE];
  MultihopOscilloscopeC.Receive -> Collector.Receive[AM_OSCILLOSCOPE];
  MultihopOscilloscopeC.RootControl -> Collector;

  components new PoolC(message_t, 10) as UARTMessagePoolP,
    new QueueC(message_t*, 10) as UARTQueueP;

  MultihopOscilloscopeC.UARTMessagePool -> UARTMessagePoolP;
```

```
  MultihopOscilloscopeC.UARTQueue -> UARTQueueP;

  components new PoolC(message_t, 20) as DebugMessagePool,
    new QueueC(message_t*, 20) as DebugSendQueue,
    new SerialAMSenderC(AM_CTP_DEBUG) as DebugSerialSender,
    UARTDebugSenderP as DebugSender;

  DebugSender.Boot -> MainC;
  DebugSender.UARTSend -> DebugSerialSender;
  DebugSender.MessagePool -> DebugMessagePool;
  DebugSender.SendQueue -> DebugSendQueue;
  Collector.CollectionDebug -> DebugSender;
}
```

In this file we declare a couple components and interfaces:

- HamamatsuS1087ParC is the lightsensor.
- CollectionC is a data collecting service which uses the tree routing protrocol to deliver data to the root.
- ActiveMessageC is the 'naming' wrapper around the CC2420 active message layer.
- CollectionsenderC is the virtual collection send abstraction (sends multihop RF)
- SerialActiveMessageC are serial messages for the communication between the mote and the computer
- SerialAmSenderC sends to the serial port
- PoolC is a component that supplies the dynamic memory pool
- QueueC is a general FIFO queue component, with a certain sizee

In the remaining part of the file connections are made with these components to make them usable in the main program.

MULTIHOPOSCILLOSCOPEC

MODULE PART OF THE FILE

```
module MultihopOscilloscopeC {
  uses  {
    // Interfaces for initialization:
    interface Boot;
    interface SplitControl as RadioControl;
    interface SplitControl as SerialControl;
    interface StdControl as RoutingControl;
```

In this file we declare the interface which we are going to use. The above 4 interfaces are used for the initialization:

- Splitcontrol is used to switch the control between the radio and serial connection
- Boot is used for booting
- StdControl is used switch between the on and off power status of the

```
    interface Send;
    interface Receive as Snoop;
    interface Receive;
    interface AMSend as SerialSend;
    interface CollectionPacket; /*
```

```
    interface RootControl;

    interface Queue<message_t *> as UARTQueue;
    interface Pool<message_t> as UARTMessagePool;
```

The above interfaces are in charge of the communication, serial and multihop:

- Send is the interface to send (uart or RF)
- Receive as Snoop listen to the packet that are sent with RF
- Rootcontrol is the interface which controls if the node is the root or not
- Queue & Pool is used as memeory storage like with the function memcpy

```
    interface Timer<TMilli>;
    interface Read<uint16_t>;
    interface Leds;
```

The other interfaces have a simple function:

- Timer is used with the interval (Timer.fired)
- Read is being used to make readings
- Leds to use the leds

IMPLEMENTATION PART OF THE FILE

```
implementation {
  task void uartSendTask();
  static void startTimer();
  static void fatal_problem();
  static void report_problem();
  static void report_sent();
  static void report_received();

  uint8_t uartlen;
  message_t sendbuf;
  message_t uartbuf;
  bool sendbusy=FALSE, uartbusy=FALSE;

  oscilloscope_t local;

  uint8_t reading;

  bool suppress_count_change;
  bool ROOT = FALSE;
```

- uartSendTask() is a task which is used to send data over the uart to a computer and back. A task activates a component to do background processing in an application. So a task is a function which tells TinyOS to execute a certain action later in time. The other function are used to trigger a certain event to help to debug.
- Sendbuf & uartbuf are of the type message_t. Message_t is the standard message buffer in TinyOS 2.x. This type keeps the data at an offset, this is important when you pass a message buffer between 2 different layers. Sendbuf & uartbuf are used with the transmission off messages over RF and the uart.

- sendbusy en uartbusy are bools and are used in the program to signal when it is possible to send something with RF of uart.
- Local is linked to your struct from the header file, so it contains interval, version, ID…
- Reading is a variable which contains the number of readings that have past
- Suppress_count_change is a bool: when we come across an Oscilloscope message, then we check his sample count. If it is further then our, then we jump ahead (we put our count equal to the count of the received message). If this situation occurs then we have to suppress our next count++. This is a simple form of time synchronization.
- ROOT is also a bool: we use it to see if the node is configured as the root or not.

```
  event void Boot.booted()   {
  local.interval = DEFAULT_INTERVAL;
  local.id = TOS_NODE_ID;
  local.version = 0;

          if (local.id % 500 == 0){
           call RootControl.setRoot();
            ROOT = TRUE;
          }

    if (call RadioControl.start() != SUCCESS)
    fatal_problem();

  if (call RoutingControl.start() != SUCCESS)
    fatal_problem();
}
```

The application is booted with Boot.booted(). In this event the ID, interval and version off the node is initialized. We check if the configured ID is equal to the root ID, if it is then we put the bool ROOT = TRUE.
The radiocontrol & routingcontrol is started in this event, because of this, some events are activated.

```
  event void RadioControl.startDone(error_t error)
  {
    if (error != SUCCESS)
      fatal_problem();

    if (sizeof(local) > call Send.maxPayloadLength())
      fatal_problem();

    if (ROOT == TRUE)
          {
                  if (call SerialControl.start() != SUCCESS)
                  fatal_problem();
          }

          startTimer();
  }

  event void SerialControl.startDone(error_t error)
  {
    if (error != SUCCESS)
```

```
    fatal_problem();
  }
```

The above events are coupled to the start function, these events are used to signal errors. If no errors occur , then we check if the configured node has the ID of the root. If it is the root ID then we start the SerialControl because it is only the root who uses this interface. After this phase we start the timer de timer (startTimer()).

```
 static void startTimer()
 {
  if (call Timer.isRunning())
        call Timer.stop();

  call Timer.startPeriodic(local.interval);
  reading = 0;
 }

 event void RadioControl.stopDone(error_t error) { }
 event void SerialControl.stopDone(error_t error) { }
```

The timer is started with a period equal to the 'local.interval'. So the timer fires with that period, which is enough time to receive and transmit messages.

```
 event message_t*
 Receive.receive(message_t* msg, void *payload, uint8_t len)   {
  oscilloscope_t* in = (oscilloscope_t*)payload;
  oscilloscope_t* out;
  if (uartbusy == FALSE)
        {
              out = (oscilloscope_t*)call SerialSend.getPayload(&uartbuf);
              if (len != sizeof(oscilloscope_t))
              {
                    return msg;
              }
              else
              {
                    memcpy(out, in, sizeof(oscilloscope_t));
              }

              uartlen = sizeof(oscilloscope_t);
              post uartSendTask();
  }
        else
        {
   message_t *newmsg = call UARTMessagePool.get();
   if (newmsg == NULL)
         {
    report_problem();
    return msg;
   }

   out = (oscilloscope_t*)call SerialSend.getPayload(newmsg);
   memcpy(out, in, sizeof(oscilloscope_t));
```

```
    if (call UARTQueue.enqueue(newmsg) != SUCCESS)
            {
      call UARTMessagePool.put(newmsg);
      fatal_problem();
      return msg;
    }
  }
  return msg;  }
```

Only the root will receive the message from this interface. It is his job to send the received messages to the PC through the serial uart of the telosB.

the event message_t ***receive**(message_t *msg, void *payload, uint8_t len) has 3 parameters:

- Msg is a pointer to the buffer where the incomming AM message is
- Payload is a pointer to the payload off the packet
- Len is the length of the data region

De function memcpy(out, in, sizeof(oscilloscope_t) copies multiple fields of the struct to another and has 3 parameters:

- Out is the destination
- In is the source
- Sizeof(oscilloscope_t) is the size of the data that needs to be copied

If the uart is not busy, then we activate uartsSendTask() and the data will be send over the uart bus to the computer.

If the uart is busy, then we store the message in a queue, when the uart is free, then we will send the messages in the. If a new message ==null then we drop it.
If we can't add a new message to the tail of the queue, then we drop the message and we wait until the queue space starts to get full without being full.

```
 task void uartSendTask()
 {
   if (call SerialSend.send(0xffff, &uartbuf, uartlen) != SUCCESS) {
     report_problem();
   }
            else
            {
     uartbusy = TRUE;
   }
 }

 event void SerialSend.sendDone(message_t *msg, error_t error)
 {
   uartbusy = FALSE;
   if (call UARTQueue.empty() == FALSE)
            {
     message_t *queuemsg = call UARTQueue.dequeue();
     if (queuemsg == NULL)
              {
```

```
      fatal_problem();
      return;
    }
    memcpy(&uartbuf, queuemsg, sizeof(message_t));
    if (call UARTMessagePool.put(queuemsg) != SUCCESS)
            {
      fatal_problem();
      return;
    }
    post uartSendTask();
  }
}
```

SerialSend is bound to SerialAMSenderC.AMSend (configuratie bestand) and sends the received data to the computer through the serial port. So SerialSend.send(0xffff, &uartbuf, uartlen) sends a packet with a data payload of a certain length to a certain address. It has 3 parameters:

- 0xfff is the address of the destination het adres naar waar het pakket wordt gestuurd
- &uartbuf is the message with the data
- Uartlen is the length of the payload in the packet

The next event checks if the transmission is completed.Het volgende event controleert of de verzending afgehandeld is.
The transmission over the uart is finished, so it isn't busy anymore. Then we check if the queue is empty, if this is not the case, then the data is extracted from the Messagepool (queue with the data that needs to be send over the uart) and sent.

```
 event message_t*
 Snoop.receive(message_t* msg, void* payload, uint8_t len)
 {
   oscilloscope_t *omsg = payload;

   report_received();

   if (omsg->version > local.version)
           {
     local.version = omsg->version;
     local.interval = omsg->interval;
     startTimer();
   }
   if (omsg->count > local.count)
           {
     local.count = omsg->count;
     suppress_count_change = TRUE;
   }
   return msg;
 }
```

We listen to the traffic in the sky.
The code above is only activated when the root sends a packet to a node. We receive a packet from the root. De payload contains a value which is compared to the "local.version" value. If the value of the packet is higher than we take over this value (version & interval). In this way the timer is adjusted and the timer is started again.

If the number of samples is smaller than we take over the value of the count and make the bool suppress_count_change TRUE. This way the teller won't increment.

```
event void Timer.fired()   {
  if (reading == NREADINGS) {
    if (!sendbusy)
            {
                    oscilloscope_t *o = (oscilloscope_t *)call Send.getPayload(&sendbuf);
                    memcpy(o, &local, sizeof(local));

                    if (call Send.send(&sendbuf, sizeof(local)) == SUCCESS)
                            sendbusy = TRUE;
     else
                            report_problem();
    }
    reading = 0;
    if (!suppress_count_change)
      local.count++;
    suppress_count_change = FALSE;
  }

  if (call Read.read() != SUCCESS)
    fatal_problem();
}
```

With every sample period we take a reading of the sensor. In the event Read.readDone, the variable "reading" increments until it reaches the value of "Nreadings". When the array is full, then it will bes end with the send interface, which is coupled to the CollectionSenderC. Further we increment our count if we did not jump forward.

```
event void Send.sendDone(message_t* msg, error_t error)
{
  if (error == SUCCESS)
    report_sent();
  else
    report_problem();

  sendbusy = FALSE;
}
```

If the transmission is complete then the bool sendbusy is set to FALSE en reading can again be transmitted.

```
event void Read.readDone(error_t result, uint16_t data)
{
  if (result != SUCCESS)
          {
    data = 0xffff;
    report_problem();
  }
  local.readings[reading++] = data;
}
```

The readings from the sensor are stored in the array local.readings.

MULTI HOP/SENS OSCILLOSCOPE (PETER)

INTRODUCTION

We figured out some solutions to adapt the original MultiHopOscilloscope application:

- We make an extra field in local for each new sensor we add. This is a rather simple solution for our application and requires little new code. The reason why we didn't choose for this is, because we would have to adapt our Java GUI. Considering we are not accustomed to Java, this choice fell out of favor.
- Put the sensor data alternatively in local.readings and make local.id an array with length NREADINGS. In this field we alternatively put a different id to distinct the sensors from each other. This solution is less elegant than the previous and has the same the disadvantages, which is why we didn't opt for it.
- The third solution is to keep our local intact, so that we don't have to adapt our Java GUI. When our sensor is ready we transfer the data to local as we did in the original application. To put it simply: we copy the data into local. Another thing we should we aware of is our count .We shouldn't increment this for each packet we send, instead we should have a count variable for each sensor we use. To make a distinction between different sensors we make sure that the source field is different for each packet we send containing the data of a particular sensor

MULTIHOPOSCILLOSCOPEAPP.NC

```
configuration MultihopOscilloscopeAppC { }
implementation {
  components MainC, MultihopOscilloscopeC, LedsC, new TimerMilliC();
  components new HamamatsuS1087ParC() as Sensor;
  components new DemoSensorC() as Sensor2;
  components PrintfC;
```

We define some new components ,namely PrintfC for the printf service and DemoSensorC() which serves as the second sensor.Notice that the DemoSensorC component is generec (new keyword) and that we rename it to Sensor2 with the *as* keyword. If we wish to change this sensor, we only have to replace DemoSensorC() with the name of a different sensor, with the advantage that we can leave the rest of our code as it is.

```
  MultihopOscilloscopeC.Boot -> MainC;
  MultihopOscilloscopeC.Timer -> TimerMilliC;
  MultihopOscilloscopeC.ReadA -> Sensor;
  MultihopOscilloscopeC.ReadB -> Sensor2;
  MultihopOscilloscopeC.Leds -> LedsC;


  MultihopOscilloscopeC.PrintfControl -> PrintfC;
  MultihopOscilloscopeC.PrintfFlush -> PrintfC;
```

We wire the correct interfaces to each other. This is pretty clear.

MULTIHOPOSCILLOSCOPEC.NC

We will only place here that has changed since the original code. This code will be marked in red. When there is any rational doubt or ambiguity, the code will be explained further.

INTERFACES

```
// Miscalleny:
  interface Timer<TMilli>;
  interface Read<uint16_t> as ReadA;
  interface Read<uint16_t> as ReadB;
  interface Leds;

  interface SplitControl as PrintfControl;
  interface PrintfFlush;
```

We add the Read interface for the 2nd sensor. Notice that this interface has a parameter. The last 2 interface are for the printf-library.

PREPROCESSOR

```
#include "printf.h"
#include "Timer.h"
#include "MultihopOscilloscope.h"
```

With this line we add the functionality of the printf-library. Since the 3 bits of information we get from toggling the leds are more than insufficient for debugging a complex application, we needed a better way to get run-time information. The printf function provides us this. It has certainly proven to be a handy tool in testing & debugging an application.

DECLARATIE VARIABELEN

```
//index for readingsa&b
  uint8_t readinga; /* 0 to NREADINGS */
  uint8_t readingb; /* 0 to NREADINGS */
  //array to save the sensordata before they are copied into local.readings
  nx_uint16_t readingsa[NREADINGS];
  nx_uint16_t readingsb[NREADINGS];
  //nessecary for a correct count
  uint16_t counta = 0;
  uint16_t countb = 0;
```

The purpose of the variables is to temporarily save the sensor data we collect. Considering we have 2 or more sensors working and we have only one readings field in local, we must have a way to save them in another location. The attentive reader may wonder why I simply did not add another local instead of adding these variables. This choice was made so that as little code as possible should be adapted. The memory cost of these extra variables does not justify the extra code and potential errors we would make.

```
  //TRUE if the readings are ready to send but the radio is busy
 bool awaiting = FALSE;
 bool bwaiting = FALSE;
```

These variable are there to make sure that all sensor data gets transmitted. One of the problems we encountered during the writing of our application was that the data of the 2nd sensor never got transmitted. More explanation in the definition of the event timer.fired.

```
event void Boot.booted() {
   local.interval = DEFAULT_INTERVAL;
   local.id = TOS_NODE_ID;
   local.version = 0;

   // Beginning our initialization phases:
   if (call RadioControl.start() != SUCCESS)
    fatal_problem();

   if (call RoutingControl.start() != SUCCESS)
    fatal_problem();

   if (call PrintfControl.start() != SUCCESS)
          fatal_problem();

 }
```

Once the mote has booted we initialize the necessary services. Here we initialize the Printf service by calling the command PrintfControl.start().

```
 event void PrintfControl.startDone(error_t error) {
          printf("Hi I am writing to you from my TinyOS application!!\n");
          call PrintfFlush.flush();
}
```

When the printf service has started, this event will be signaled to our application. We printf a line of text here to make sure that the function is correctly working. There is no need any more to check error_t. We use the command PrintfFlush.flush() we want to send a rule of information to our terminal. A problem I encountered with this function is that should be used only once in a block of code. Otherwise the lines after this statement will not be send to the terminal by repeating this command.

```
//printf service has been stopped
  event void PrintfControl.stopDone(error_t error) {
            printf("This should not be printed...");
            call PrintfFlush.flush();
  }

  //printf lines have been sent
  event void PrintfFlush.flushDone(error_t error) {}
```

The other events for the printf service.

```
static void startTimer() {
    if (call Timer.isRunning()) call Timer.stop();
    call Timer.startPeriodic(local.interval);
            //indices worden op nul geplaatst
    readinga = 0;
    readingb = 0;
   }
```

The indices need to be set to zero to make sure the first packet is correct.

EVENT TIMER.FIRED

This block of code serves to copy the values from readingsa to local. First we check if there is enough data to transmit, we check if there are as many as NREADINGS readings. NREADINGS is five in our case, it is not advised to raise this number. This would cause the radio to corrupt.

The local.count fields makes sure that the data is displayed in the correct order in our GUI. We have declared a counter for both our sensors.

The node_id variable is also changed, if we use the 2nd sensor we increment it with 1, and so on…

The printf lines are there for debugging purposes.

```
event void Timer.fired() {
   if (readinga == NREADINGS) {
            printf("Aantal metingen bereikt\n");
    i = 0;
    local.count = counta;
    local.id = TOS_NODE_ID;
    printf("id = %u\n",local.id);
   //call PrintfFlush.flush();
   while(i<NREADINGS){
            local.readings[i] = readingsa[i];
            i++;
    }
...
```

In the while loop the values of the array readingsa are assigned to local.readings

```
....
....
 if (!suppress_count_change)
          counta++;
 suppress_count_change = FALSE;
 readinga = 0;
 }
  else
          awaiting = TRUE;
```

The lines of code after this block are the same as in the original application.

We increment the count so that the data is displayed correctly.

Our index is placed at zero again so that new sensordata can be collected.

If we couldn't send for some unknown reason, then awaiting is set TRUE, this variable will be used in the send.SendDone event.

The code for the other sensors is analogous to the previous!

```
//call naar de sensoren
   if (call ReadA.read() != SUCCESS)
    fatal_problem();
   if (call ReadB.read() != SUCCESS)
    fatal_problem();
```

We change the name of the interfaces and add one for the 2nd sensor.

EVENT SEND.SENDDONE

```
event void Send.sendDone(message_t* msg, error_t error) {
  if (error == SUCCESS)
    report_sent();
  else
    report_problem();
  sendbusy = FALSE;

  if ( awaiting ){
            //code for sending packet a
  printf("sendbusy = %u\n",sendbusy);
    if (!sendbusy) {
          oscilloscope_t *o = (oscilloscope_t *)call Send.getPayload(&sendbuf);
          memcpy(o, &local, sizeof(local));
          if (call Send.send(&sendbuf, sizeof(local)) == SUCCESS){
            sendbusy = TRUE;
            printf("The packet of sensor A has been transmitted!\n");

          }
     else{
       report_problem();
            printf("Packet of sensor A has not been transmitted!\n");
          }
    if (!suppress_count_change)
          counta++;
    suppress_count_change = FALSE;
    readinga = 0;
  }
  }
  awaiting = FALSE;
```

Why have we added so much code here? The problem with our first implementation of a MultiHopOscilloscope with multiple sensors was that the 2 sensor finished together (their read.ReadDone event was signaled at the same time) In timer.fired() we then send the data. The problem is that the radio needs to be checked if it's busy or not. This is absolutely necessary, otherwise 2 functions could access the radio at the same time, corrupting each other's

messages. When the application wants to send the first packet it calls Send and sets sendbusy to TRUE. This function is split-phase and as a result immediately returns, while the radio is still busy. This is a form of non-blocking code. The application can then execute the next piece of code, which is the code for sending the packet of the 2nd sensor. This means that while the radio is still busy, the processor will execute the code to start sending the second packet. This will not work considering sendBusy is checked.

In practice this would result in the first packet getting sent, but the second would always be skipped. The radio van only send a packet when its ready with the previous packet. That's why we add a variable to each of pieces of code where the send function is called. This variable checks whether or not the data has been sent or not! In the Send.sendDone event this variable is checked and the data that has not been sent yet, will be transmitted.

The rest of the code as the same in the event timer.fired(), it is the code for transmitting the data. At the bottom of this block, its guardvariable, awaiting, is changed back to FALSE. Notice that the first block of code from timer.fired() has not been copied. This is not necessary because the sensor collects its data less fast than the radio can transmit a packet. Should the sensor sample faster than the radio, the values in local.readings will not be valid anymore. But then again a lot of packets will be dropped then. This is undesirable.

```
event void ReadA.readDone(error_t result, uint16_t data) {
   if (result != SUCCESS) {
    data = 0xffff;
    report_problem();
   }
   readingsa[readinga++] = data;
   }

  event void ReadB.readDone(error_t result, uint16_t data) {
   if (result != SUCCESS) {
    data = 0xffff;
    report_problem();
   }
   readingsb[readingb++] = data;
    }
```

De data has to be put in the correct variable, and there is also a block of code for the 2nd sensor.

CONCLUSION

Writing this application was not a walk in the park. One badly put accolade can cause the application not to reach 5 readings. A small mistake is quickly made and this can result in disastrous effect for your application. The fact that there is not an IDE at hand for TinyOS and that there are no well debugging options doesn't make programming for this platform any easier. Because of this is made some stupid errors which I didn't see at first. During the making of this application I learned to work with the printf-library, which is essential in my opinion for solving programming errors. Unfortunately this library has some errors. These errors cost me about a day of work.

Furthermore it may be noticeable that this is not so streamlined. If we had used the first solution, the code would be nicer and more compact. The best solution in this could case would be a queue where we can stack up the packets.

The only limitation to this program is that the TOD_NODE_IT have to be separated by 2 units for each mote. So the motes should not be given successive ID's.

BIBLIOGRAPHY

Björn, S., & David, J. (2006-2007). Bachelorproef Zigbee.

MULTI HOP/SENS OSCILLOSCOPE

INTRODUCTION

The multi hop/sens oscilloscope application is an extension to the multihoposcilloscope application. It implements 2 readings from 2 different sensors (light & voltage). This application is a backup plan for implementing multiple sensors.

HOW DOES IT WORK

We need to send 2 different sensor values, so we specify an extra struct variable (oscilloscope_t localD) to store the readings and to transmit.
We have 2 different sensors, so we need 2 read interfaces.
The idea is rather simple, when the Timer fires, then we look if 5 reading have been stored. When this is done, then we block the light sensor and activate the reading of the voltage sensor. So every 5 reading it switches from sensor and transmits it through RF.

IMPLEMENTATION FILE

DECLARION

```
bool NEXTREADINGREADY=FALSE;

 oscilloscope_t localD;
 oscilloscope_t localL;

 uint8_t readingL; /* 0 tot NREADINGS */
 uint8_t readingD;
```

We declare the bool NEXTREADINGREADY to be able to switch between the different sensors. We declare 2 variables localD and localL of the type struct. We also declare 2 integers readingL and readingD to keep track of the number of readings that have been taken.

EVENT BOOT.BOOTED

```
 localL.id = TOS_NODE_ID;
```

```
localD.id = TOS_NODE_ID + 1;
```

We specify 2 different ID for each node to make it compatible with the Java GUI supplied by TinyOS.

EVENT TIMER.FIRED

```
{
  if (readingL == NREADINGS)
        {
    if (!sendbusy)
         {

                oscilloscope_t *o = (oscilloscope_t *)call Send.getPayload(&sendbuf);
                memcpy(o, &localL, sizeof(localL));

                if (call Send.send(&sendbuf, sizeof(localL)) == SUCCESS)
                {
                        sendbusy = TRUE;
                }
     else
                        report_problem();
    }
    readingL = 0;
    /* Part 2 of cheap "time sync": increment our count if we didn't
       jump ahead. */
    if (!suppress_count_change)
      localL.count++;

    suppress_count_change = FALSE;
  }

        if (readingD == NREADINGS)
        {
      if (!sendbusy)
         {

                oscilloscope_t *o = (oscilloscope_t *)call Send.getPayload(&sendbuf);
                memcpy(o, &localD, sizeof(localL));

                if (call Send.send(&sendbuf, sizeof(localD)) == SUCCESS)
                {
                        sendbusy = TRUE;
                }
     else
                        report_problem();
    }
    readingD = 0;
    /* Part 3 of cheap "time sync": increment our count if we didn't
       jump ahead. */
    if (!suppress_count_change)
      localD.count++;
```

```
    suppress_count_change = FALSE;
  }

        if ( NEXTREADINGREADY == FALSE )
        {
                printf("start van readL!\n");
                        if (call ReadL.read() != SUCCESS)
                        fatal_problem();
        }
        else
        {
                printf("start van readD!\n");
                if (call ReadD.read() != SUCCESS)
                        fatal_problem();
        }
}
```

Basically in this event we check if 5 reading have been taken from the light sensor, if so, then we transmit the data with the ID of localL. After the 5 readings we switch from sensor and take 5 readings from the voltage sensor. After the transmission of these 5 readings we switch again to the light sensor.

EVENT READL.READDONE

```
{
  if (result != SUCCESS)
        {
    data = 0xffff;
    report_problem();
  }

  localL.readings[readingL++] = data;

        if (readingL == 5)
        {
        NEXTREADINGREADY = TRUE;
        printf("NEXTREADINGREADY is TRUE\n");
        }
}
```

In this event we take readings from the light sensor, if we reach 5 readings then we give the bool NEXTREADINGREADY a certain value, so that we switch to the other sensor.

EVENT READL.READDONE

```
{
        if (result != SUCCESS)
        {
    data = 0xffff;
    report_problem();
          }
        data = data/100;
```

```
  localD.readings[readingD++] = data;

        if ( readingD == 5)
        NEXTREADINGREADY = FALSE;
}
```

In this event we take readings from the voltage sensor, if we reach 5 readings then we give the bool NEXTREADINGREADY a certain value, so that we switch to the other sensor.

RSSI TESTING

INTRODUCTION

The final step of our thesis was to integrate our work with that of the master students. The subject of their research was dynamic event positioning. This means that when an event occurs at a node (the sensors exceed a certain value), a *mobile node* should be sent to it, to investigate the situation, for example by taking a picture of the environment. To be able to direct this *mobile node* to the correct coordinates, its position should be known. The best solution according to the masters is to use RSSI readings and then triangulate these.

To know the distance between nodes, we need to know the relation between distance and RSS. The purpose of this work was to write an application so that we could gather some data about this relationship. If we want to know the distance to a certain node, we need to know the standard RSSI value for that distance.

The second piece of functionality this application will demonstrate is that we can put the intelligence in the mobile node. In the application of our master students, all the data would get routed to a base station which processes this data, calculates the correct direction for the mobile node to follow from it, and then sends these directions back to the mobile node.

The application we built is merely for testing purpose only, it has too little functionality to be of any practical value, but is has been a helpful tool in helping us collect the RSSI data.

The application actually consists of 2 applications, namely one for the anchor node which detects the event and broadcasts a warning signal, and one for the mobile node which detects this warning and tries to find the anchor node.

ANCHOR NODE

FUNCTIONAL DESCRIPTION

This mode checks it sensors for a threshold, when one of the sensor values exceeds this threshold, it signals a warning to the other nodes.

ANCHORNODE.H

In this header file we declare some constants and the packet format.

```
enum {
  /* Default sampling period. */
  DEFAULT_INTERVAL = 512,

  AM_OSCILLOSCOPE = 0x93,

  MAX_SENS_VAL = 1000
};
```

DEFAULT_INTERVAL is our sampling interval, MAX_SENS_VAL is the threshold for our sensors. If it exceeds this, we start broadcasting.

This is our format of our payload. It contains the id the mote needing sending, and a bool warning which signals whether there is a problem or not.

```
typedef nx_struct oscilloscope {
  nx_uint16_t id; /* Mote id of sending mote. */
  nx_bool     warning;  /* Warning or Not */
} oscilloscope_t;
```

of

ANCHORNODEAPPC

```
#include "printf.h"

configuration AnchorNodeAppC { }
implementation
{
  components AnchorNodeC, MainC, ActiveMessageC, LedsC,
    new TimerMilliC(), new DemoSensorC() as Sensor,
    new AMSenderC(AM_OSCILLOSCOPE), PrintfC;

  AnchorNodeC.Boot -> MainC;
  AnchorNodeC.RadioControl -> ActiveMessageC;
  AnchorNodeC.AMSend -> AMSenderC;
  AnchorNodeC.Timer -> TimerMilliC;
  AnchorNodeC.Read -> Sensor;
  AnchorNodeC.Leds -> LedsC;
  AnchorNodeC.PrintfControl -> PrintfC;
  AnchorNodeC.PrintfFlush -> PrintfC;
}
```

This is al pretty easy, we add the interfaces for the radio, boot, timers, sensors, leds and the printf library.

Note that we do an include for the printf library.

ANCHORNODEC

Here we will discuss the actual implementation of the program.

PREPROCESSOR

```
#include "Timer.h"
#include "AnchorNode.h"
#include "printf.h"
```

The necessary header files of course.

INTERFACES

This is the signature of our applications, here we declare the interfaces we use.

```
module AnchorNodeC
{
  uses {
    interface Boot;
    interface SplitControl as RadioControl;
    interface AMSend;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
    interface Leds;
    interface SplitControl as PrintfControl;
    interface PrintfFlush;
  }
}
```

We provide the same interfaces as the one we wire in our app file.

VARIABLES

```
implementation
{
  message_t sendbuf;
  bool sendbusy;

  /* Current local state - warning, node id */
  oscilloscope_t local;
```

Here we declare the necessary variables. Sendbuf is the buffer of our radio and is standard practice in any application that uses the radio or the uart.

Sendbusy is a guard variable which enforces that only one function has access to the radio at a given time.

Local is where we keep our data, this will be used as the payload for the radio packet

EVENTS & CALLS

```
  // Use LEDs to report various status issues.
  void report_problem() { call Leds.led0Toggle(); }
  void report_sent() { call Leds.led1Toggle(); }
  void report_event() { call Leds.led2Toggle(); }

  event void Boot.booted() {
    local.id = TOS_NODE_ID;
    if (call RadioControl.start() != SUCCESS)
      report_problem();
    if (call PrintfControl.start() != SUCCESS)
      report_problem();
  }
```

Used for debugging and for signaling that the mote has booted

```
void startTimer() {
  call Timer.startPeriodic(DEFAULT_INTERVAL);
}
event void RadioControl.startDone(error_t error) {
  startTimer();
}
event void RadioControl.stopDone(error_t error) {
}
event void PrintfControl.startDone(error_t error) {
          printf("Hi I am writing to you from my TinyOS application!!\n");
          call PrintfFlush.flush();
}
event void PrintfControl.stopDone(error_t error) {
          printf("This should not be printed...");
          call PrintfFlush.flush();
}
event void PrintfFlush.flushDone(error_t error) {
             }
```

Some events to start the radio, timer and the printf service.

We start the timer with the interval defined in our header file, the interval is DEFAULT_INTERVAL.

```
event void Timer.fired() {
          if ( local.warning == 1 )
           {
          if (!sendbusy && sizeof local <= call AMSend.maxPayloadLength())
           {
             memcpy(call AMSend.getPayload(&sendbuf), &local, sizeof local);
             if (call AMSend.send(AM_BROADCAST_ADDR, &sendbuf, sizeof local) == SUCCESS)
               sendbusy = TRUE;
           }
          if (!sendbusy)
            report_problem();
    }
   if (call Read.read() != SUCCESS)
     report_problem();
  }
```

The timer.fired event is fired when a certain amount of time has passed by. If local.warning = TRUE then we start sending our emergency broadcast. But first me must check whether our data will fit in one packet. The memcopy is used to copy local into our sendbuffer.

We send our packet with the AMSend interface, which is single-hop and use the AM identifiers… Our first argument is the address to which we want to send. In this case we used the broadcast address.

```
event void AMSend.sendDone(message_t* msg, error_t error) {
   if (error == SUCCESS)
     report_sent();
   else
     report_problem();

   sendbusy = FALSE;
 }
```

Finally we call Read.read() which makes our sensor start reading its value. It will signal the Read.readDone event when its ready, which is explained below.

This event is signaled when the radio has finished sending its packet. The error status is defined in error. We set the sendbusy variable back to FALSE.

```
event void Read.readDone(error_t result, uint16_t data) {
   if (result != SUCCESS)
    {
          data = 0xffff;
          report_problem();
    }
    if ( data > MAX_SENS_VAL )
          local.warning = 1;
    printf("Sensorvalue = %u\n",data);
    printf("Warning = %u\n",local.warning);
    call PrintfFlush.flush();
  }
```

This event signals when the sensors has finished sampling its data. The data is passed in the data argument.

If the value of data exceeds our threshold we set local.warning to true. We also print some lines for debugging purposes.

CONCLUSION

This should be easy enough to understand, if you however for some reason don't, it might be a good idea to read some of the basic tutorials in the TinyOS documentation.

BLINDNODE

FUNCTIONAL DESCRIPTION

This application listens for the broadcasts sent by the anchor nodes. When it receives one, it calculates the RSSI of the message and stores this data. Since RSSI values are very fluctuating, we need to apply a filter to it. In our case we took the average of a number of readings.

With this data the mobile node should be able to determine where it should go to. Since we only get data from one node we can only determine the distance between the two, but not their angle, or their exact position to each other.

We added a very simple mechanism through which the mobile node can come closer to its anchornode. Note that this routing mechanism is very simple, it was merely intended as a helpful tool for our master students.

BLINDNODE.H

This event is roughly the same as AnchorNode.h

The only thing we add is the constant:

```
NREADINGS = 10
```

This defines the number of RSSI readings we need before we calculate the average.

BLINDNODEAPPC

The app file is also simple and adds few little interfaces. We the CC2420ActiveMessageC which provides the CC2420Packet interface. With this interfaces we can measure our RSSI.

BLINDNODEC

VARIABLES

```
implementation
{
  int8_t RSSIvals[NREADINGS];
  int8_t reading = 0;
  int8_t RSSIval_filter_old;
  int8_t RSSIval_filter_new;
  int8_t direction = 1;
  bool wait = FALSE;
  bool first = TRUE;
```

RSSIvals is the array that stores the RSSI readings. RSSIval_filter_old & new are the results we get from our filter function. Direction defines the direction we should head out.

The bool wait is necessary for when the mobile node is moving to its destination, any RSSI measurements then are pretty useless.

EVENTS

```
  event void Notify.notify( button_state_t state ) {
    if ( state == BUTTON_PRESSED ) {
      wait = FALSE;
      printf("Userbutton is pressed\n");
              call PrintfFlush.flush();
    }
  }
```

This event is fired when the user presses the userbutton (telosb platform only!). This is used to tell the application that the mote has reached its destination. Masterstudent Man Hun Wong will interface a mobile robot to the mote, so this should become the event where the robot signals that the movement is done.

```
  void filter(){
    int16_t temp_total = 0;
    for(i=0;i<NREADINGS;i++){
       temp_total += RSSIvals[i];
    }
    RSSIval_filter_new = temp_total / NREADINGS;
  }
```

A simple averaging filter

```
void movement(){
 if (first){
        RSSIval_filter_old = RSSIval_filter_new;
        first = FALSE;
 }
 printf("RSSIval_filter_new = %i\n", RSSIval_filter_new);

 if (RSSIval_filter_new < RSSIval_filter_old){
        printf("RSSI is verslechterd!\n");
        direction++;
        if (direction >= 5)
                direction = 1;
 }
 else {
        printf("RSSI is verbeterd!\n");
 }
 printf("Druk op de knop als je klaar bent!\n");
 if (direction == 1)
        printf("Vooruit\n");
 else if (direction == 2)
        printf("Achteruit\n");
 else if (direction == 3)
        printf("Links\n");
 else if (direction == 4)
        printf("Rechts\n");
 RSSIval_filter_old = RSSIval_filter_new;
 }
```

This function determines in which direction the mobile node should travel.

Our simple algorithm does the following:
It compares the new RSS readings to the old.
If they improve: maintain same direction
Else: move in a new direction

Our application simply cycles between forward, backward, leftward, rightward.

While this is a very simple inefficient way of working, it does the trick.

```
event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
   oscilloscope_t *omsg = payload;
   report_received();

   if ( !wait ){
    local.warning = omsg->warning;

    if ( local.warning == 1 )
          RSSIvals[reading++] = call CC2420Packet.getRssi(msg);
    if ( reading == NREADINGS ){
          filter();
          reading = 0;
          wait = TRUE;
      if (RSSIval_filter_new > 10){
                printf("Destination reached!\n");
            printf("Proficiat!\n");
      }
          else
              movement();
          call PrintfFlush.flush();
    }
   }
   return msg;
  }
```

This event is signaled whenever the radio receives a message. It turns on a led for indication.

We take a RSSI reading with the CC2420Packet.getRSSI command and put it in the RSSIvals array.
If we have enough readings we filter them.

If our RSSI is good enough we are around our destination, else we need to move.

RSSI READINGS

INTRODUCTION

For our master student Man Hung Wong, it was very important that he had RSSI readings for a given distance so he could, determine where his mobile node was.

METHOD

We used the applications we described in the above sections. We set the NREADINGS constant to 10 which would mean that the mobile node takes 10 RSSI readings before he averages them. We distanced our motes 1 meter separated from each other. We then noted the averaged reading; we repeated this process 5 times. This process was repeated but for different distances, we incremented 1 meter per turn, until we reached 10 meters. For each step then averaged the results and subtracted 45 from these. This is because the CC2420 radio has a 45dBm offset.

These measurements were performed about 50 cm from the ground. We turned off any nearby radio signal, such as Wi-Fi, Bluetooth. These could interfere with our signal.

We then regressed this data to get a mathematical equation which describes the relationship between distance and RSSI.

The equation for this graph is: $y = a(1-e^{-bx})$ with

- a = -0.017352956
- b = 0.087152976

OCTOPUS SYNCHRONOUS

We were working together with the master students Nick & David and they had a problem with sending the data from anchor nodes to the root node. Basically they visualize the nodes in the network by taking the RSSI from the mobile node in reference to the anchor nodes. The anchor nodes send their data with the AMSend interface to the mobile node, which receives these packets with the AMReceive interface. The localization is controlled by the root node, so the mobile node forwards the received packets to the root. We investigated the problem and came to the conclusion that the data from the anchor nodes wasn't being sent by the mobile node to the root node. The local data of the mobile node was being forwarded.

The solution is to synchronize the application with the help of some bools. So, we are going to implement a system that switches between the transmission of the data from the mobile node (local data) and the transmission of the data from the anchor nodes (external data).

APPLICATION OCTOPUS

BOOLS

```
bool SWITCHTOB = FALSE;
bool BSEND = FALSE;
```

We specify 2 bools to switch between the transmission of local & external data:

- SWITCHTOB is a bool which gives the control to the event Blindreceive so that it is able to transmit the data from the anchor nodes
- BSEND is a bool that we used in Timer.fired to make the sending process synchronous with the timer

TIMER.FIRED

```
                        if(voltageIsRead && lightIsRead && tempIsRead && humidityIsRead && rssiIsRead)
                        {
                                localCollectedMsg.count++;
                                if(root)
                                        post serialSendTask();
                                else if(anchor)
                                        post anchorSendTask();
                                else
                                        if (SWITCHTOB == FALSE)
                                                post collectSendTask();
                                        else
                                                BSEND = TRUE;
                        }

                }
        }
```

Timer.fired is the central event in the application. We have added some lines of code to make the program synchronous. If we have collected the data from the sensors, and if the bool SWITCHTOB = FALSE then we transmit the local data. If it is TRUE then we switch to the blindreceive event and transmit the external data.

COLLECTION SEND

```
event void CollectSend.sendDone(message_t* msg, error_t error) {
        if (error != SUCCESS)
                reportProblem();
        sendBusy = FALSE;

        if (BSEND == FALSE)
        {
                SWITCHTOB = TRUE;
                printf("Collectsend.sendDone van BLIND is gedaan\n");
                call PrintfFlush.flush();
        }
        else
        {
                BSEND = FALSE;
                SWITCHTOB = FALSE;
                printf("Collectsend.sendDone van BROADCAST is gedaan\n");
                call PrintfFlush.flush();
        }
        localCollectedMsg.reply = NO_REPLY;
        reportSent();
}
```

We transmit the data with the CollectSend.send as a result we get the event CollectSend.sendDone. In this event we adjust the value of the bools to switch between the transmission of the local and external data. So, if we just transmitted the local data, then the bool BSEND == FALSE and thus we make the bool SWITCH equal to TRUE, so that when the timer fires, the bool BSEND become TRUE and we can transmit the external data.

VISUALIZATION OF NODES WITH RSSI

INTRODUCTION

We are working together with 3 different master students: Nick Verbaendert, David Henderickx en Wong Man Hung. Our task is to implement RSSI to make a mobile node to move into the direction of a node which signals an certain event. So we have one mobile node and 3 anchor nodes to help us to go in the right direction. We also need to make our application compatible with the visualization software of Nick and David. We have implemented the following:

- 4 sensors: voltage, temperature, light intensity and humidity
- RSSI
- AM receiver
- Changed the memory allocation (pool & queue)
- Collection Tree Protocol

HEADER FILE

```
typedef nx_struct oscilloscope {
                nx_am_addr_t moteId;
                nx_uint16_t count;
                nx_uint16_t quality;
                nx_am_addr_t parentId;
                nx_uint8_t reply;

                nx_uint16_t Voltreading;
                nx_uint16_t Lightreading;
                nx_uint16_t Tempreading;
                nx_uint16_t Humidityreading;
                nx_uint16_t RSSI;

} oscilloscope_t;
```

In the header file we added 5 new elements:

- Voltreading: contains the internal voltage
- Lightreading: contains the light intensity
- Tempreading: contains the temperature
- Humidityreading: contains the humidity
- RSSI: Received Signal Strength Indication

CONFIGURATION FILE

```
components new DemoSensorC() as SensorVoltage;                 // internal Voltage
components new HamamatsuS1087ParC() as SensorLight;        // light intensity
components new SensirionSht11C() as SensorTempHumidity;  //  temperature and humidity
```

```
MultihopOscilloscopeC.ReadVoltage -> SensorVoltage;
MultihopOscilloscopeC.ReadLight -> SensorLight;
MultihopOscilloscopeC.ReadTemperature -> SensorTempHumidity.Temperature;
MultihopOscilloscopeC.ReadHumidity -> SensorTempHumidity.Humidity;
```

In this file we add the components for the 4 different sensors. We connect them to the read interfaces so that it is usable in the main program.

```
components new AMReceiverC(AM_OSCILLOSCOPE) as MobileReceiver;

MultihopOscilloscopeC.MobileReceive -> MobileReceiver;
```

To locate a mobile node in a network we take the RSSI from 3 different nodes and with those values we can calculate where the mobile node is situated. Thus, we add the component AMReceiver to listen to the packets sent by the other 3 nodes to get the RSSI.
We connect the AMReceiverC to the Receive interface.

```
components CC2420ActiveMessageC as Radio;

MultihopOscilloscopeC -> Radio.CC2420Packet;
```

We add the component CC2420ActiveMessageC (CC2420 is the radio) to be able to get the RSSI.

```
components CtpP as CollectP;
MultihopOscilloscopeC.CollectInfo -> CollectP;
```

We add the CtpP to get some more information like the parentID and expected transmissions (ETX). CTP uses expected transmissions as its routing gradient.

MODULE FILE

MODULE PART

In the module part we declare the interface which we are going to use and the commands included.

```
interface CC2420Packet;
```

Interface for getting RSSI

```
interface Receive as MobileReceive;
```

Interface voor de AM receive.

```
interface CtpInfo as CollectInfo
```

Interface for getting certain data from CTP like quality and parentID.

```
interface Queue<message_t *> as RadioQueue;
```

```
interface Pool<message_t> as RadioMessagePool;
```

Interface for the use of a queue and pool. This is used for storing received messages for the anchor nodes.

```
interface Read<uint16_t> as ReadVoltage;
interface Read<uint16_t> as ReadLight;
interface Read<uint16_t> as ReadTemperature;
interface Read<uint16_t> as ReadHumidity;
```

Read interface reads out the different sensors

IMPLEMENTATION PART

```
bool voltageIsRead = FALSE;
bool lightIsRead = FALSE;
bool tempIsRead = FALSE;
bool humidityIsRead = FALSE;
bool rssiIsRead = FALSE;
bool root = FALSE;
```

We add some bools to see when a certain reading is finished and to know which node is the root.

```
event void Timer.fired(){
                if (call ReadVoltage.read() == SUCCESS) voltageIsRead = TRUE;

                if (call ReadLight.read() == SUCCESS) lightIsRead = TRUE;

                if (call ReadTemperature.read() == SUCCESS) tempIsRead = TRUE;

                if (call ReadHumidity.read() == SUCCESS) humidityIsRead = TRUE;
                getRssi(&sendbuf);

                if(voltageIsRead && lightIsRead && tempIsRead && humidityIsRead && rssiIsRead){
                        local.count++;
                        if(!root)
                        {
                                oscilloscope_t *out;
                                if (!sendBusy){
                                out = (oscilloscope_t *)call CollectSend.getPayload(&sendbuf);
                                        memcpy(out, &local, sizeof(oscilloscope_t));
                                        post collectSendTask();
                                }
                                else
                                {
                                        message_t *newmsg = call RadioMessagePool.get();
                                        if (newmsg == NULL)
                                        {
                                                report_problem();
                                        }

                                        out = (oscilloscope_t*)call CollectSend.getPayload(newmsg);
                                        memcpy(out, &local, sizeof(oscilloscope_t));
```

```
                                        if (call RadioQueue.enqueue(newmsg) != SUCCESS)
                                        {
                                        call RadioMessagePool.put(newmsg);
                                                fatal_problem();
                                        }
                                }

                        }
                }
        }
```

Timer.fired is the main event of the program. Here we collect the data and transmit it when bool sendbusy = FALSE. If the bool is false, then we store the data in the right format on the queue.

```
        task void collectSendTask() {
                if (!root) {
                        if (call CollectSend.send(&sendbuf, sizeof(oscilloscope_t)) == SUCCESS)
                        {
                                oscilloscope_t *o;
                                sendBusy = TRUE;
                                o = (oscilloscope_t *)call CollectSend.getPayload(&sendbuf);
                                printf("De id = %u\n",o->moteId);
                                call PrintfFlush.flush();
                        }
                        else
                                report_problem();
                }
        }
```

CollectSendTask is a task and is called on to transmit his own data and data received from the anchor nodes, so it is called on in the events 'Timer.fired' and 'Mobilereceive.receive'. The task is adapted in the way that we don't need to copy the data into the message_t sendbuf here.

```
        event void CollectSend.sendDone(message_t* msg, error_t error){
                if (error != SUCCESS)
                        report_problem();

                sendBusy = FALSE;

                if (call RadioQueue.empty() == FALSE)
                {
                        message_t *queuemsg = call RadioQueue.dequeue();
                        if (queuemsg == NULL)
                        {
                              fatal_problem();
                                return;
                        }
                        memcpy(&sendbuf, queuemsg, sizeof(message_t));
                        if (call RadioMessagePool.put(queuemsg) != SUCCESS)
                        {
                                fatal_problem();
                                return;
```

```
                        }
                        post collectSendTask();

                }

                local.reply = NO_REPLY;

                report_sent();
}
```

In the Collect.sendDone we put the bool sendBusy to FALSE and check if there is still data on the queue. If so, then we transmit them.

```
        event void ReadVoltage.readDone(error_t ok, uint16_t data) {
        if (ok == SUCCESS)
        {
                fillPacket();
                local.Voltreading = data;
        }
        else
        {
                data = 0xffff;
                report_problem();
 }
}

        event void ReadLight.readDone(error_t ok, uint16_t data) {
        if (ok == SUCCESS)
        {
                fillPacket();
                local.Lightreading = data;
        }
        else
        {
   data = 0xffff;
   report_problem();
 }
}

        event void ReadTemperature.readDone(error_t ok, uint16_t data) {
        if (ok == SUCCESS)
        {
                fillPacket();
                local.Tempreading = data;
        }
        else
        {
   data = 0xffff;
   report_problem();
 }

}

        event void ReadHumidity.readDone(error_t ok, uint16_t data) {
```

```
        if (ok == SUCCESS)
        {
                fillPacket();
                local.Humidityreading = data;
        }
        else
        {
   data = 0xffff;
   report_problem();
  }

 }

        uint16_t getRssi(message_t *msg){
                local.RSSI = (uint16_t) call CC2420Packet.getRssi(msg);
                rssiIsRead = TRUE;
                return local.RSSI;
        }
```

In the above events, we fill our packet with the data it needs.

```
        void fillPacket() {
                uint16_t tmp;
                call CollectInfo.getEtx(&tmp);
                local.quality = tmp;
                call CollectInfo.getParent(&tmp);
                local.parentId = tmp;
        }
```

The function fillPacket further fills the packet we want to transmit with certain data like quality and parentID. This function is called on when we have read the different sensors.

```
        event message_t* MobileReceive.receive(message_t* msg, void* payload, uint8_t len) {

                oscilloscope_t *in = (oscilloscope_t*)payload;
                oscilloscope_t *out;

                report_received();
                if(!sendBusy)
                {
                        out = (oscilloscope_t *)call CollectSend.getPayload(&sendbuf);
                        if (len != sizeof(oscilloscope_t))
                        {
                                return msg;
                        }
                        else
                        {
                                memcpy(out, in, sizeof(oscilloscope_t));
                        }

                        post collectSendTask();
                }
                else
```

```
		{
			message_t *newmsg = call RadioMessagePool.get();
			if (newmsg == NULL)
			{
				report_problem();
				return msg;
			}

			out = (oscilloscope_t*)call CollectSend.getPayload(newmsg);
			memcpy(out, in, sizeof(oscilloscope_t));

			if (call RadioQueue.enqueue(newmsg) != SUCCESS)
			{
			    call RadioMessagePool.put(newmsg);
				fatal_problem();
				return msg;
			}
		}
		sendBusy = TRUE;
		return msg;
	}
```

The Mobilereceive is the next main part of the application. In this event we receive the different packets transmitted by the anchor nodes. We take the payload from these packages and transmit them if the bool sendBusy = FALSE. If not then we put the data on a queue in the right format.

VISUALIZATION AND DIRECTION WITH THE USE OF RSSI

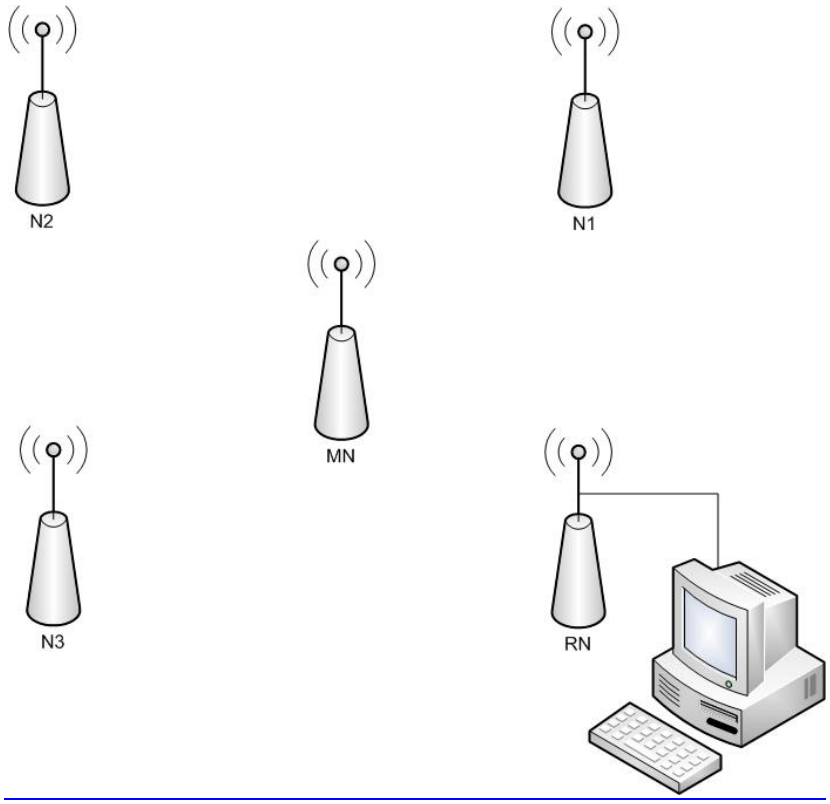
INTRODUCTION

This is the final application, it supports the following:

- Multihop routing
- Compatible with the Java GUI of the master students
- Gives information through the printf interface to go in the direction of the alarming anchor node
- Interfaces specific to use by the anchor nodes: a anchor node transmits with this interface so that the mobile node can receive the RSSI

So basically we brought all the previous applications together and added support for the anchor nodes.

WHAT DOES IT DO?



We build a network with 5 nodes. 1 node acts as the root node (RN) and is connected to a computer. It must have the node ID 0. On this node runs an application of master students. We don't need the root for our application, it is only to make it compatible with their visualization software. Nodes N1, N2 and N3 function as anchor points. They need to have the ID 1, 2 or 3. They are used as reference points in the network. The last node is the mobile node (MN) and it can have any ID except the one we previously defined. The mobile node receives packages from the anchor nodes. Anchor nodes can send an alarm. The mobile node receives this signal and tries to move in the direction of the alarmed node with the help of RSSI. With our application you receive directions through the printf interface, because there was no robot available.

MODULE FILE

```
components new AMSenderC(AM_OSCILLOSCOPE) as AnchorSender;
MultihopOscilloscopeC.AnchorSend -> AnchorSender;
```

We add the component AMSenderC to transmit the data and RSSI from the anchor node to the mobile node. We connect the AMSenderC to the Send interface.

IMPLEMENTATIN FILE

MODULE PART

```
interface AMSend as AnchorSend;
```

Interface for the AMSend

IMPLEMENTATION PART

```
task void anchorSendTask();
```

We declare a task to transmit the anchor data.

```
bool anchorBusy = FALSE;
bool anchor = FALSE;
```

We use these bools to indicate if the assigned ID of the node is equal with the predefined anchor ID's and if the anchor is busy with transmitting his data.

EVENT BOOT.BOOTED

```
if (TOS_NODE_ID == 1 || TOS_NODE_ID == 2 || TOS_NODE_ID == 3)
        anchor = TRUE;
else if(TOS_NODE_ID == 0)
        root = TRUE;
```

We check if the assigned ID is equal with one of the predefined anchor ID's, which are 1, 2 and 3. The root has the ID 0 and the mobile node can be any other.

TASK VOID ANCHORSENDTASK

```
{
        if (!anchorBusy && anchor) {
                oscilloscope_t * o = (oscilloscope_t *)call AnchorSend.getPayload(&sendbuf);
                memcpy(o, &local, sizeof(oscilloscope_t));
                if (call AnchorSend.send(4, &sendbuf, sizeof(oscilloscope_t)) == SUCCESS){
                        anchorBusy = TRUE;
                }else{
```

```
                    report_problem();
                }
            }
        }
```

We declare the task 'anchorsendtask' to transmit the data of the anchor nodes. The benefit of a task is, that it runs in the background. We check if we are dealing with an anchor node and if the anchor isn't already busy with sending.

ANCHORSEND.SENDDONE

```
    {
        if (error != SUCCESS){
            report_problem();
            anchorBusy = FALSE;
        }else{
            anchorBusy = FALSE;
            local.reply = NO_REPLY;
            report_sent();
        }
    }
```

We check if the transmission is successful from the anchor node.

EVENT TIMER.FIRED

```
                    if (call ReadVoltage.read() == SUCCESS) voltageIsRead = TRUE;

                    if (call ReadLight.read() == SUCCESS) lightIsRead = TRUE;

                    if (call ReadTemperature.read() == SUCCESS) tempIsRead = TRUE;

                    if (call ReadHumidity.read() == SUCCESS) humidityIsRead = TRUE;
                    getRssi(&sendbuf);



    if(voltageIsRead && lightIsRead && tempIsRead && humidityIsRead && rssiIsRead)
    {
        if (anchor)
            post anchorSendTask();
        else
        {
            if ( !wait )
            {

                local.count++;
                if(!root)
                {
                    oscilloscope_t *out;
                    if (!sendBusy)
```

```
                                        {
                                        out = (oscilloscope_t *)call CollectSend.getPayload(&sendbuf);
                                                memcpy(out, &local, sizeof(oscilloscope_t));
                                                post collectSendTask();
                                        }
                                        else
                                        {
                                                message_t *newmsg = call RadioMessagePool.get();
                                                if (newmsg == NULL)
                                                {
                                                        report_problem();
                                                }

                                                out = (oscilloscope_t*)call CollectSend.getPayload(newmsg);
                                                memcpy(out, &local, sizeof(oscilloscope_t));

                                                if (call RadioQueue.enqueue(newmsg) != SUCCESS)
                                                {
                                            call RadioMessagePool.put(newmsg);
                                                        fatal_problem();
                                                }
                                        }

                                }
                        }
                }
        }
}
```

The event Timer.fired is the central event of the application. Here we make the distinguish between an anchor or mobile node. The anchor node (ID 1, 2, 3) has to transmit his data (with AnchorSendTask) to the mobile node (ID 4). The mobile node transmits his own data and forwards the data received from the anchor nodes to the root node (with CollectSendTask).