# 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 sensordata 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 eachother. This is pretty clear I think.

# 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

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.

**#include "printf.h"**

#include "Timer.h"

#include "MultihopOscilloscope.h"

## 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 sensordata 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 an other location. The attentive reader may wonder why I simply did not add another local instead of adding these varaibles. 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.

The other events for the printf service.

//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) {}

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.

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++;**

}

...

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.

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

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

….

....

if (!suppress\_count\_change)

**counta++;**

suppress\_count\_change = FALSE;

**readinga = 0;**

}

**else**

**awaiting = TRUE;**

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.

# Results

//screenshot toevoegen

# 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.