**Legenda:**

node, mote and bracelet are used synonimously

node child = mote child = child’s bracelet

node parent = mote parent = parent’s bracelet

**smartBracelets.h file:**

I decided to create a structure with 7 fields:

* msg\_type, a 8-bit integer for the type of the message (PAIRING at the beginning, INFO for usual messages from node child to node parent)
* key, a 8-bit integer which encapsulates the key of the bracelet
* src\_address, a 8-bit integer with the mote id, (TOS\_NODE\_ID) of the node sending the message
* stop\_pairing, it is a 8-bit integer, actually a boolean field, which signals the will of sop the pairing phase
* x\_coordinate, a 8-bit integer which represents the X coordinate of the child mote.
* y\_coordinate, a 8-bit integer which represents the Y coordinate of the child mote.
* kinematic\_status, a 8 bit integer which represents the kinematic status of the child.

I decided to define 2 random keys, since we’ve to simulate an environment with 2 couples of motes, so 4 motes in total for the simulation; of these 4 motes, 2 use the RANDOM\_KEY\_1, the other 2 use the RANDOM\_KEY\_2.

* Parent\_1 -> mote\_ID == 1
* Child\_1 -> mote\_ID == 2
* Parent\_2 -> mote\_ID == 3
* Child\_2 -> mote\_ID == 4

So there’s the necessity to use 2 different keys, simulating the two different communications between Parent\_1 and Child\_1, and between Parent\_2 and Child\_2.

The kinematic status are defined after the keys.

**smartBraceletsC.nc file:**

*PAIRING PHASE steps:*

1) *event Boot.booted*

starting from the boot, we launch the SplitControl command.

2) *event SplitControl.startDone()*

When SplitControl start is done, we start also MilliTimer1, it is useful to resend broadcast message of type pairing in case it is not received by the other bracelet which it is addressed to.

3) *event SplitControl.startDone()*

We send the pairing message with the sendPairing() function.

4) *function sendPairing()*

The sendPairing() function prepares the message to be send. We have 4 possibilities, depending on which is the node\_ID we’re. In any case, the msg\_type is PAIRING; the key depends upon our mote\_ID:

* Parent\_1 and Child\_1 use RANDOM\_KEY\_1
* Parent\_2 and Child\_2 use RANDOM\_KEY\_2

Then we send in broadcast this message

4s) *function fillKeyField(my\_msg\_t\* rcm, char\* key)*

This function is used, as suggested, to fill the key field of the rcm message with the proper key (RANDOM\_KEY\_1 or RANDOM\_KEY\_2).

5) *event AMSend.sendDone()*

When the broadcast message is sent, AMSend.sendDone event is triggered, we check whether the packet just sent is equal to the buf parameter of the event sendDone, if yes alright, we print on the debug the successful in sending the packet.

6) *event Receive.receive()*

Supposing the broadcast pairing message is correctly received from the mote child, in this mote child is triggered the Receive.receive event. It checks whether the message received is equal to the size of the message it is supposed to receive, then prints on debug the message received and checks whether the message is of type PAIRING. In that case, another node wants to pair with us, we check our node\_ID.

-> in case our node\_ID is 1 or 2, we’ve to check if the key received inside the packet is equal to our RANDOM\_KEY\_1;

-> in case our node\_ID is 3 or 4, we’ve to check if the key received inside the packet is equal to our RANDOM\_KEY\_2;

in any case, we save the address of the mote who wants to pair with us, and if the stop\_pairing field of the message received is set to false, we have to signal to that node that the pairing is done, so we call the function sendPairingDone();

6s) *function equalKeys(my\_msg\_t\* rcm, char\* secondKey)*

This function is used to check whether the key received as part of the message rcm received is equal to the proper mote’s key.

7) *function sendPairingDone()*

We flag the pairing\_done flag, as the child mote (or the parent mote) has received the PAIRING message and the key is equal to our key.

The sendPairingDone() function is similar to the sendPairing() function, the only difference is that in this case, the stop\_pairing field of the message is set to true, and the address to send the message is not broadcast but unicast, we’ve to send the message only to the saved\_address we obtained in the Reeive.receive() event before.

8) *event Receive.receive()*

If the message stop\_pairing sent from the mote child is correctly received by the mote parent, or vice-versa, the parent (or the child) prints on dbg the received message, checks its own node\_ID

-> if its node\_ID == 1 (or 2), it checks again the key of the message received, and set the saved \_address to the address of the other bracelet to be connected to. If the stop\_pairing field is set to true, the pairing is done, so this mote sets the pairing\_done flag to true.

9) *event MilliTimer1.fired()*

Each second, this timer fires, and checks whether the pairing is done or not, if it is not done, (maybe because the message is lost somehow), it re-calls the sendPairing() function. So, every second we re-try to pair the two motes.

*Until now, we’ve managed the pairing between the parent mote and the child mote. At the end of this 9 initial steps, each mote is correctly connected to the other mote. This is the end of the pairing phase.*

*OPERATION MODE steps:*

10) *event MilliTimer1.fired()*

At a certain point, a node child fires the MilliTimer1 and it realizes the pairing is done, so it starts another timer, called MilliTimer2, which starts the phase 2: the operation mode. The parent motes don’t start the MilliTimer2, since it is used by child motes to send the kinematic status, but they start MilliTimer3, used to send eventually a MISSING alarm if the child doesn’t send the position into the INFO message after 1 minute. Since now, the packets sent by the child motes have to be acknowledged by the parents motes, to be sure they receive them, so we set to true the flag flag\_ACK\_requested.

11) *event MilliTimer2.fired()*

The firing of timer 2 calls the function sendInfo(), to send the info read from the child’s sensors to the proper parent mote.

12) *function sendInfo()*

When MilliTimer2 is fired, a child node reads the values from the (fake) sensor. The reading is done in this way:

a) The value of the coordinate X is read by using the command Read.read(). The Read.readDone() event modifies a global variable, called value\_x, (by shifting 8 bits the “data” variable) which is used by the caller (MilliTimer2.fired() function) to encapsulate the value into the coordinate\_x field of the message to be sent;

b) The value of the coordinate Y is obtained by making the module out of the “data” variable;

c) The value of the kinematic status is obtained by generating another random value between 0 and 255 (included), making the sum of value\_x + value\_y and making the module; then we divide the interval [0, 255] in 4 subparts:

I) The first subpart goes from 0 (included) to 76 (included), and if the value falls in this range, the kinematic status is STANDING;

II) The second subpart goes from 77 (included) to 153 (included), if the value falls in this range, the kinematic status is WALKING;

III) The third subpart goes from 154 (included) to 230 (included), if the value falls in this range, the kinematic status in RUNNING;

IV) The fourth subpart goes from 231 (included) to 255 (included), if the value falls in this range, the kinematic status is FALLING.

As we can see, the first 3 subparts, cover approximatively 30% of the interval: [0, 230], while the last one covers the last 10% of the interval: [231,255]. In this way the first three events have a probability of 0.3 each, while the last event has a probability of 0.1. In any way, the value generated generates a kinematic status that is either STANDING, WALKING, RUNNING or FALLING. Other values are not possible, in that case we launch an appropriate dbgerror.

Now we’re ready to fill the remaining fields of the packet, depending on the mote who’s executing this piece of code.

* If it’s mote 2 (Child 1), the src\_address is 2 and the key is RANDOM\_KEY\_1
* If it’s mote 4 (Child 2), the src\_address is 4 and the key is RANDOM\_KEY\_2

Other values of the mote\_ID are not valid and there’s an error, in that case we print on dbgerror.

Now we can send the packet properly filled, calling the AMSend.send() command, passing the proper address of the parent, exploiting the variable saved\_address we initialized before.

These packets we require to be acknowledged by the parent mote.

13) *event AMSend.sendDone()*

The AMSend.sendDone() event unlocks the channel if the packet is the correct one, then checks whether the ACK arrives or not; in case the ACK doesn’t arrive, it re-sends the message through the sendInfo() function, reading new values from the (fake) sensor.

*At the end of this 13th step, the child mote is correctly sending values read from its sensor to the parent mote, 1 message every 10 seconds. This is the end of the Operation mode phase.*

*ALERT MODE steps:*

14) *event Receive.receive()*

The parent mote receives the INFO message from the child... so we’re in the Receive.receive() event, we check whether it is a PAIRING message or a INFO message and we enter the INFO branch. Immediately we save the position received from the child, filling the variables last\_x for the last X coordinate received, and last\_y for the last Y coordinate received. Then we check if the kinematic status of the child is FALLING, in that case we launch an alarm (in this case, a print on the dbg), signaling the status FALLING of the child, its node\_ID and its position (X,Y)

In any case, since we received a message from the child, we stop the MilliTimer3 and restart it for other 60 seconds.

15) *event MilliTimer3.fired()*

Since the MilliTimer3 fires only when the parent mote doesn’t receive any info from its child, this means the child has not sent any message in the last minute. So we launch a MISSING alarm on the dbg, reporting the child ID, the parent mote which expects the message that didn’t arrive, and the last position received (X,Y) from the child.

*At the end of this 15th step, the motes are communicating, with the child motes that periodically sends its INFO messages and the parent mote that receives them and does the proper thing, depending on the message received by its own child.*

**topology.txt file:**

We set the gain of the communication channel between each node and the other 3 nodes, so gain of the communication channel between 1 and 2, gain between 1 and 3, gain between 1 and 4, gain between 2 and 1 and so on... Totally 12 definition of the gains, we set the gain of each communication channel equal to the others. (-60 dB).

**smartBraceletsAppC.nc file:**

Here we’ve the pairing of the interfaces of the smartBracelets.nc file with the components defined in this file.

**runSimulationScript.py file:**

Here we create the nodes, the radio channel, add the noise model, the topology file. We start the motes at different times. Then we start the simulation with TOSSIM.

**logFile.txt file:**

Here we can see the TOSSIM simulation, printed by redirecting the output of the runSimulationScript.py file on a new text file. We can see the PAIRING MESSAGES, the INFO messages containing different status of the child motes, properly received by the parent mote, eventually the FALL ALARM printed by the parent mote whether the child sends a FALL message and so on…

**Possible improvements:**

- Make possible for a parent mote to associate it many child motes.

Modifications to do in that case: we would have to make an array saved\_addresses[] in the smartBraceletsC.nc file instead of a simple int variable saved\_address, this if the mote is a parent mote, otherwise if it is a child mote, the saved address is one and only one. Only the parent can have many saved addresses, one for each child.

**Notes:**

Since I have a MTU of 28 bytes, the structure my\_msg\_t needs to have all fields uint\_8. Remember that this message brings inside a 21-char key(20 char for the key and 1 char for the string terminator), then the message\_type, stop\_pairing, src\_address, x\_coordinate, y\_coordinate, kinematic\_status. So each one can occupy max 1 byte, so every field must be uint8. Otherwise, we would have to split packets.