Telos rev. b application

Senseless





Tim Van Overtveldt

Peter De Cauwer

Artesis, Applied Engineering: Electronics-ICT (2008-2009)

WSN localization for SCALA

Inhoud

[In general 3](#_Toc228012276)

[Framework 3](#_Toc228012277)

[application 3](#_Toc228012278)

[Telos rev. b 4](#_Toc228012279)

[nesc 5](#_Toc228012280)

[Tinyos 5](#_Toc228012281)

[installation 7](#_Toc228012282)

[software 7](#_Toc228012283)

[installation off the application on the mote 7](#_Toc228012284)

[Code senseless 8](#_Toc228012285)

[header file 9](#_Toc228012286)

[Senselessappc file 12](#_Toc228012287)

[Senselessc 15](#_Toc228012288)

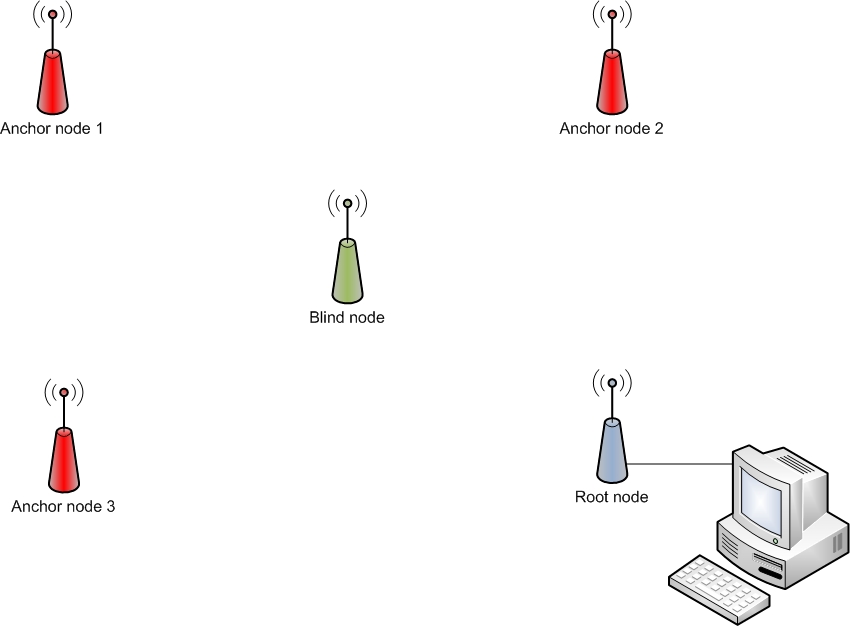
[module part of the file 15](#_Toc228012289)

[implementation part of the file 18](#_Toc228012290)

# In general

## Framework

This application is part of the Senseless framework. With this framework we are going to locate blind nodes with the help of anchor nodes (nodes that know their position). In figure 1, you see the most common set up for localizing a blind node with 3 anchor nodes.



*Figure 1: Set up for localizing a blind node*

## application

The application collects data (most importantly RSSI) and is installed on the Telos rev. B node. It takes samples from all the sensors (temperature, light, humidity and voltage) with a default sampling period of 100s. The RSSI is measured with a default sampling period of 10s. The default sampling rates can be altered manually in the header file or remotely with the GUI. All the data is sent through the serial USB port to a parser. The application is written in nesC.

For performance reasons, this application will exist out of different messages:

* Sensor measurement message, this will be send to the controller to report about the measurements of the sensors, this message exists out of:
  + Battery: Voltage of the processor
  + Light: Measurement of the light sensor
  + Humidity: Measurement of the humidity sensor
  + Temperature: Measurement of the temperature sensor
  + Button: Button pressed or not
  + Mote ID
* Status message, this will be send to the controller in 2 cases: if a node enters or returns to the network then it automatically transmits such a message and in the second case as response to a request sent from the GUI. This message exists out of:
  + Mote ID
  + Active: Participating in the localization or not
  + AN: Anchor node
  + Posx: X coordinate of the node
  + Posy: Y coordinate of the node
  + Samplerate: Sample period for the sensors of the nodes
  + locRate: Sample rate for the anchor nodes to broadcast beacon messages
  + leds: Decimal number that represents the leds
  + power: Power level of transmission (0 to 31)
  + frequency: Frequency on which the packets will be transmitted
  + conn: Contains the number of motes the BN is connected to
* Localization message, this is used to transmit the coordinates of the blind nodes and their RSSI, this message exist out of:
  + Mote id
  + ANmoteid: ID of the anchor where you got the RSSI from
  + VANs: Grade of the anchor node of the source (0= BN, 1=AN and 2=VAN)
  + VANr: Grade of the anchor node of the receiver (0= BN, 1=AN and 2=VAN)
  + hopCount: the hop count
  + RSSI: Received Signal Strength Indication

These messages will be transmitted to the root of the network. The root will forward these messages to the controller through the serial port and the parser.

### Telos rev. b

Telos rev. B (Figure 2) is an ultra low power wireless sensor module (“mote” or “node”) for research and experimentation. It is developed by UC Berkeley to enable wireless sensor network (WSN) research.

The node has the following specification:

* TI MSP430 microcontroller with 10kB RAM
* IEEE 802.15.4 compliant radio
* Integrated temperature, light, humidity and voltage sensor
* TinyOS 2.x compatible
* Programmable via USB interface
* Integrated antenna



*Figure 2: Telos rev. B node*

### nesc

nesC is an extension to the C programming language designed to embody the structuring concepts and execution model of TinyOS.

The basic concepts behind nesC are:

* Separation of construction and composition: programs are built out of *components*, which are assembled ("wired") to form whole programs. Components have internal concurrency in the form of *tasks*. Threads of control may pass into a component through its interfaces. These threads are rooted either in a task or a hardware interrupt.
* Specification of component behavior in terms of set of *interfaces*. Interfaces may be *provided or* used by components. The provided interfaces are intended to represent the functionality that the component provides to its user, the used interfaces represent the functionality the component needs to perform its job.
* Interfaces are bidirectional: they specify a set of functions to be implemented by the interface's provider (*commands*) and a set to be implemented by the interface's user (*events*). This allows a single interface to represent a complex interaction between components (e.g., registration of interest in some event, followed by a callback when that event happens). This is critical because all lengthy commands in TinyOS (e.g. send packet) are non-blocking; their completion is signaled through an event (send done). By specifying interfaces, a component cannot call the send command unless it provides an implementation of the sendDone event.
* Typically commands call downwards, i.e., from application components to those closer to the hardware, while events call upwards. Certain primitive events are bound to hardware interrupts.
* Components are statically linked to each other via their interfaces. This increases runtime efficiency, encourages robust design, and allows for better static analysis of programs.
* nesC is designed under the expectation that code will be generated by whole-program compilers. This should also allow for better code generation and analysis.

### 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.

# installation

A node needs to be connected to the computer through the serial USB port. The node can be configured as the root (node id 0) of the network or as a blind node (any other node id). The network makes use of Collection: the network is symbolized as a tree with the root at the base, where the data is sent to.

## software

We program and install the applications on the motes with 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.



## installation off the application on the mote

In the directory of your application ( /…/Senseless ) we open a terminal and type:

* *Make telosb:* De TinyOS application is compiled from this directory
* *Make telosb reinstall,nodeID:* This compiles an image from the application with the noteID, which is compatible with the Telos rev. B platform. If you choose nodeID 0, then the node is set as the root of the network, if you choose any other number that the node is a blind node (node without a know location).

# Code senseless

The application contains mainly 3 files:

* Senseless.h: the header file has a structs for every different message in the WSN
* SenselessAppc.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
* Senseless.nc: the module file contains the implementation of the application. It may use every command from the interface, which it implements.

## header file

|  |
| --- |
| #ifndef SENSELESS\_H  #define SENSELESS\_H  #include <AM.h>  enum {  AM\_COLLMSG = 0x93,  AM\_LOCMSG = 0x90,  AM\_STATMSG = 0x82,  AM\_DISSMSG = 0x71,  AM\_OSCILLOSCOPE = 0x94,  BROADCAST\_DIS\_KEY = 42,  DEFAULT\_SAMPLING\_PERIOD = 100000,  DEFAULT\_LOC\_PERIOD = 10000,  LOC\_PERIOD = 0x11,  SAMPLING\_PERIOD = 0x22,  ACTIVE\_REQUEST = 0x33,  AN\_REQUEST = 0x44,  LED\_REQUEST = 0x55,  POWER\_REQUEST = 0x77,  FREQUENCY\_REQUEST = 0x88,  POSX\_REQUEST = 0x99,  POSY\_REQUEST = 0xAA,  STATUS\_REQUEST = 0xBB  };  typedef nx\_struct CollMsg {  nx\_am\_addr\_t moteid;  nx\_uint8\_t type;  nx\_uint16\_t battery;  nx\_uint16\_t lightreading;  nx\_uint16\_t tempreading;  nx\_uint16\_t humidityreading;  nx\_bool button;  } CollMsg\_t;  typedef nx\_struct StatMsg {  nx\_am\_addr\_t moteid;  nx\_uint8\_t type;  nx\_bool active;  nx\_bool AN;  nx\_uint16\_t posx;  nx\_uint16\_t posy;  nx\_uint16\_t sampleRate;  nx\_uint16\_t locRate;  nx\_uint8\_t leds;  nx\_uint8\_t power;  nx\_uint8\_t frequency;  nx\_uint8\_t conn;  } StatMsg\_t;  typedef nx\_struct LocMsg {  nx\_am\_addr\_t moteid;  nx\_am\_addr\_t ANmoteid;  nx\_uint8\_t VANs;  nx\_uint8\_t VANr;  nx\_uint8\_t hopCount;  nx\_uint8\_t RSSI;  } LocMsg\_t;  typedef nx\_struct DissMsg {  nx\_am\_addr\_t targetid;  nx\_uint8\_t request;  nx\_uint16\_t parameter;  } DissMsg\_t;  typedef nx\_struct broadcast {  nx\_am\_addr\_t id;  nx\_uint16\_t hopCount;  nx\_uint16\_t VAN;  } broadcast\_t;  #endif SENSELESS\_H |

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:

* AM\_COLLMSG: acts as the port which the sensor message will be sent to
* AM\_LOCMSG: acts as the port which the location message will be sent to
* AM\_STATMSG: acts as the port which the status message will be sent to
* AM\_DISSMSG: acts as the port which the dissemination message will be sent to
* AM\_OSCILLOSCOPE: acts as the port which the broadcast/beacon message will be sent to
* DEFAULT\_SAMPLING\_PERIOD: period for sampling the sensors of the node
* DEFAULT\_LOC\_PERIOD: period for broadcasting beacon message
* LOC\_PERIOD: request to change location period
* SAMPLING\_PERIOD: request to change sampling period of the sensors
* ACTIVE\_REQUEST: request to change the status of active
* AN\_REQUEST: request to change the node to an anchor or blind node
* LED\_REQUEST: request to change the leds
* POWER\_REQUEST: request to change the power level of transmission
* FREQUENCY\_REQUEST: request to change the frequency
* POSX\_REQUEST: request to change the x coordinate of the node
* POSY\_REQUEST: request to change the y coordinate of the node
* STATUS\_REQUEST: request to receive a status report

The struct CollMsg is used in the program to store the data from the sensors:

* Moteid: MoteID of sending mote
* Type: Empty field
* Battery: Voltage of node (demosensor)
* Lightreading: Light measurement
* Tempreading: Temperature measurement
* Humidityreading: Humidity measurement
* Button: Button pressed (1=YES / 0=NO)

The struct StatMsg is used to return the status of the WSN network:

* Moteid: MoteID of sending mote
* Type: Empty field
* Active: Active in the localization process? (TRUE = yes & FALSE = no)
* AN: Anchor Node? (TRUE = Anchor Node & FALSE = Blind Node)
* Posx: X coordinate of the node
* Posy: Y coordinate of the node
* sampleRate: Period of sensor message
* locRate: Period of location message
* leds: Leds (0 to 7)
* power: Power level (0 to 31)
* frequency: Frequency (11 to 26)
* conn: Contains the number of motes the BN is connected to

The struct LocMsg is used to return data that is necessary for the localization process:

* moteid: MoteID of sending mote
* ANmoteid: MoteID of anchor node
* VANs: Grade of the anchor node of the source (0= BN, 1=AN and 2=VAN)
* VANr: Grade of the anchor node of the receiver (0= BN, 1=AN and 2=VAN)
* hopCount: the hop count
* RSSI: Received Signal Strength Indication

The struct DissMsg is used to send request to the network:

* Targeted: MoteID of targetmote
* Request: Code for possible request
* Parameter: Parameter for the request

The struct broadcast is used by the anchor node to send beacon messages to the blind nodes

* Id: MoteID of sending mote.
* hopCount: how many hops it has passed
* VAN: grade of the anchor node(1=AN and 2=VAN)

## Senselessappc file

|  |
| --- |
| #include "Senseless.h"    configuration SenselessAppC { }  implementation {  // Other  //----------------------------------------------------------------------  components MainC;  components SenselessC;  components LedsC;  components new TimerMilliC(), new TimerMilliC() as BroadCastTimerMilliC, new TimerMilliC() as AnchortimerMilliC;    SenselessC.Boot->MainC;  SenselessC.Timer->TimerMilliC;  SenselessC.Anchortimer->AnchortimerMilliC;  SenselessC.BroadcastTimer->BroadCastTimerMilliC;  SenselessC.Leds->LedsC;    // Sensors  //----------------------------------------------------------------------  components new HamamatsuS1087ParC() as LightSensor;  components new SensirionSht11C() as TempHumiditySensor;  components new DemoSensorC() as VoltageSensor;    SenselessC.ReadLight->LightSensor;  SenselessC.ReadTemp->TempHumiditySensor.Temperature;  SenselessC.ReadHumidity->TempHumiditySensor.Humidity;  SenselessC.ReadBattery->VoltageSensor;    // Button  //----------------------------------------------------------------------  components UserButtonC;    SenselessC.Notify->UserButtonC;    // Serial  //----------------------------------------------------------------------  components SerialActiveMessageC;  components new SerialAMReceiverC(AM\_DISSMSG) as SerialRequestReceiver;  components new SerialAMSenderC(AM\_COLLMSG) as SerialCollectSenderC,  new SerialAMSenderC(AM\_LOCMSG) as SerialCollectSenderL,  new SerialAMSenderC(AM\_STATMSG) as SerialCollectSenderS;    SenselessC.UartPacket->SerialActiveMessageC;  SenselessC.SerialControl->SerialActiveMessageC;  SenselessC.SerialSendC->SerialCollectSenderC.AMSend;  SenselessC.SerialSendL->SerialCollectSenderL.AMSend;  SenselessC.SerialSendS->SerialCollectSenderS.AMSend;  SenselessC.SerialReceive->SerialRequestReceiver;    //RSSI & Frequency & Powerlevel  //----------------------------------------------------------------------  components CC2420ActiveMessageC as Radio;  components CC2420ControlC;    SenselessC.CC2420Packet -> Radio;  SenselessC.CC2420Config -> CC2420ControlC;    // General radio Communication  //----------------------------------------------------------------------  components ActiveMessageC;    SenselessC.RadioControl->ActiveMessageC;  SenselessC.RadioPacket->ActiveMessageC;    // Broadcast  //----------------------------------------------------------------------  components new AMReceiverC(AM\_OSCILLOSCOPE) as BroadcastReceiver;  components new AMSenderC(AM\_OSCILLOSCOPE) as BroadcastSender;    SenselessC.BroadcastReceive->BroadcastReceiver;  SenselessC.BroadcastSend -> BroadcastSender;  // Calibration  //----------------------------------------------------------------------  components new AMReceiverC(0x95) as CalibrationReceiver;  components new AMSenderC(0x95) as CalibrationSender;    SenselessC.CalibrationReceive->CalibrationReceiver;  SenselessC.CalibrationSend -> CalibrationSender;    // Collection  //----------------------------------------------------------------------  components CollectionC as CollectorC,  CollectionC as CollectorL,  CollectionC as CollectorS; // Collection  components new CollectionSenderC(AM\_COLLMSG) as ColSendC,  new CollectionSenderC(AM\_LOCMSG) as ColSendL,  new CollectionSenderC(AM\_STATMSG) as ColSendS;    SenselessC.RoutingControlC->CollectorC;  SenselessC.RoutingControlL->CollectorL;  SenselessC.RoutingControlS->CollectorS;  SenselessC.CollectSendC->ColSendC;  SenselessC.CollectSendL->ColSendL;  SenselessC.CollectSendS->ColSendS;  SenselessC.SnoopC->CollectorC.Snoop[AM\_COLLMSG];  SenselessC.SnoopL->CollectorL.Snoop[AM\_LOCMSG];  SenselessC.SnoopS->CollectorS.Snoop[AM\_STATMSG];  SenselessC.CollectReceiveC->CollectorC.Receive[AM\_COLLMSG];  SenselessC.CollectReceiveL->CollectorL.Receive[AM\_LOCMSG];  SenselessC.CollectReceiveS->CollectorS.Receive[AM\_STATMSG];  SenselessC.RootControlC->CollectorC;  SenselessC.RootControlL->CollectorL;  SenselessC.RootControlS->CollectorS;    // Dissemination  //----------------------------------------------------------------------  components DisseminationC;  components new DisseminatorC(DissMsg\_t, BROADCAST\_DIS\_KEY);    SenselessC.DisseminationControl->DisseminationC;  SenselessC.RequestUpdate->DisseminatorC;  SenselessC.RequestValue->DisseminatorC;  // Memory (for the 3 different messages)  //----------------------------------------------------------------------  components new PoolC(message\_t, 5) as RadioMessagePoolP;  components new QueueC(message\_t\*, 5) as RadioQueueP;  SenselessC.RadioPoolC->RadioMessagePoolP;  SenselessC.RadioQueueC->RadioQueueP;  SenselessC.RadioPoolS->RadioMessagePoolP;  SenselessC.RadioQueueS->RadioQueueP;  SenselessC.RadioPoolL->RadioMessagePoolP;  SenselessC.RadioQueueL->RadioQueueP;    components new PoolC(message\_t, 5) as UartMessagePoolP;  components new QueueC(message\_t\*, 5) as UartQueueP;  SenselessC.UartPoolC->UartMessagePoolP;  SenselessC.UartQueueC->UartQueueP;  SenselessC.UartPoolS->UartMessagePoolP;  SenselessC.UartQueueS->UartQueueP;  SenselessC.UartPoolL->UartMessagePoolP;  SenselessC.UartQueueL->UartQueueP;    // Printf  //----------------------------------------------------------------------  components PrintfC;  SenselessC.PrintfControl->PrintfC;  SenselessC.PrintfFlush->PrintfC;  } |

In this file we declare a couple components and interfaces:

* MainC is the system interface the TinyOS boot sequence
* LesC is the basic TinyOS LEDs abstraction
* TimerMilliC is the virtualized millisecond timer abstraction. Instantiating this component gives an independent millisecond granularity timer
* HamamatsuS1087ParC is the light sensor
* SensirionSht11C is the temperature & humidity sensor
* DemosensorC is the voltage sensor
* UserButtonC is implementation of the user button for the telosb platform
* SerialActiveMessageC are serial messages for the communication between the mote and the computer
* SerialAMSenderC sends to the serial port
* SerialAMReceiverC receives from the serial port
* CC2420ActiveMessageC is the Active Message layer for the CC2420 radio
* CC2420ControlC is the implementation for configuring a ChipCon CC2420 radio. Used to set & get the frequency
* ActiveMessageC is the 'naming' wrapper around the CC2420 active message layer
* AMReceiverC is the virtualized AM reception abstraction
* AMSenderC is the virtualized active message send abstraction
* CollectionC is a data collecting service which uses the tree routing protocol to deliver data to the root
* CollectionsenderC is the virtual collection send abstraction (sends multihop RF)
* The DisseminationC component is the top-level interface to the dissemination protocol
* PoolC is a component that supplies the dynamic memory pool
* QueueC is a general FIFO queue component, with a certain size
* PrintfC is component to be able to print lines in console

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

## Senselessc

### module part of the file

In this file we declare the interface which we are going to use.

#### Initialization

|  |
| --- |
| module SenselessC {  uses {  interface Boot;  interface SplitControl as RadioControl;  interface SplitControl as SerialControl;  interface SplitControl as PrintfControl;  interface StdControl as RoutingControlC;  interface StdControl as RoutingControlL;  interface StdControl as RoutingControlS;  interface StdControl as DisseminationControl; |

The above interfaces are used for initializing the main interfaces :

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

#### dissemination

|  |
| --- |
| interface DisseminationUpdate<DissMsg\_t> as RequestUpdate;  interface DisseminationValue<DissMsg\_t> as RequestValue; |

The above interfaces are necessary to make use of the dissemination protocol:

* DisseminationUpdate is used to update a network shared (disseminated) value
* DisseminationValue is used to read a network shared (disseminated) variable

#### Collection

There are three different collection trees for the main 3 messages in the WSN.

|  |
| --- |
| interface Send as CollectSendC;  interface Send as CollectSendL;  interface Send as CollectSendS;  interface Receive as SnoopC;  interface Receive as SnoopL;  interface Receive as SnoopS;  interface Receive as CollectReceiveC;  interface Receive as CollectReceiveL;  interface Receive as CollectReceiveS;  interface RootControl as RootControlC;  interface RootControl as RootControlS;  interface RootControl as RootControlL; |

The above interfaces are necessary to make use of the collection protocol:

* CollectSendC / CollectSendL / CollectSendS: used to send with the collection protocol
* SnoopC / SnoopL / SnoopS: is used to overhear messages
* CollectReceiveC / CollectReceiveL / CollectReceiveS: is used to receive message with the collection protocol
* RootControlC / RootControlS / RootControlL: is used to set a node as the root node of the collection tree

#### Serial

There are 3 different AMSend interfaces because the three main messages need their own private interface to be able to be correctly transmitted to the parser.

|  |
| --- |
| interface AMSend as SerialSendC;  interface AMSend as SerialSendL;  interface AMSend as SerialSendS;  interface Receive as SerialReceive;  interface Packet as UartPacket; |

The above interfaces are needed to send & receive serial:

* SerialSendC / SerialSendL / SerialSendS: is used to transmit serial
* SerialReceive: is used to receive serial
* UartPacket is the message data type accessor

#### Sensors

|  |
| --- |
| interface Read<uint16\_t> as ReadLight;  interface Read<uint16\_t> as ReadTemp;  interface Read<uint16\_t> as ReadHumidity;  interface Read<uint16\_t> as ReadBattery; |

The above interfaces are used to measure the sensors of the node:

* ReadLight: is used to take a reading of the light
* ReadTemp: is used to take a temperature reading
* ReadHumidity: is used to measure the humidty
* ReadBattery: is used to measure the voltage of the processor

#### Pushbutton

|  |
| --- |
| interface Notify<button\_state\_t>; |

The above interface is used for getting values from self-triggered devices like the pushbutton.

#### memory

We use for each different message a different queue and pool. We also use a different queue/pool for radio and the serial messages.

|  |
| --- |
| interface Queue<message\_t \*> as UartQueueC;  interface Pool<message\_t> as UartPoolC;    interface Queue<message\_t \*> as UartQueueL;  interface Pool<message\_t> as UartPoolL;    interface Queue<message\_t \*> as UartQueueS;  interface Pool<message\_t> as UartPoolS;    interface Queue<message\_t \*> as RadioQueueC;  interface Pool<message\_t> as RadioPoolC;    interface Queue<message\_t \*> as RadioQueueL;  interface Pool<message\_t> as RadioPoolL;    interface Queue<message\_t \*> as RadioQueueS;  interface Pool<message\_t> as RadioPoolS; |

The above interfaces are used to create a queue and pool:

* Queue<message\_t \*>: is used to create a queue for the type message\_t
* Pool<message\_t> is used to create a pool for the type message\_t

#### broadcast

|  |
| --- |
| interface AMSend as BroadcastSend;  interface Receive as BroadcastReceive; |

The above interfaces are used for sending and receiving broadcast/beacon messages:

* BroadcastSend: is used by an achor node to broadcast beacon messages to a blind node
* BroadcastReceive: is used by a blind node to receive the beacon messages

#### calibration

|  |
| --- |
| interface AMSend as CalibrationSend;  interface Receive as CalibrationReceive; |

The above interfaces are used for sending and receiving broadcast/beacon messages:

* CalibrationSend: is used by an achor node to broadcast beacon messages to a blind node
* CalibrationReceive: is used by a blind node to receive the beacon messages

#### CC2420

|  |
| --- |
| interface CC2420Packet;  interface CC2420Config; |

The above interfaces are specific for the radio of the Telos rev. B node:

* CC2420Packet: is used to get RSSI/LQI & set/get power level
* CC2420Config: is an HAL abstraction of the ChipCon CC2420 radio (setting frequency)

#### other

|  |
| --- |
| interface Timer<TMilli>;  interface Timer<TMilli> as BroadcastTimer;  interface Timer<TMilli> as Anchortimer;  interface Leds;  interface Packet as RadioPacket;  interface PrintfFlush; |

The remaning interfaces:

* Timer: is used for sampling the sensors of the node
* BroadcastTimer: is used by the anchor node to broadcast beacon messages
* Anchortimer: is used for calibration: anchor nodes transmit beacon messages to each other
* Leds: is used to control the leds
* RadioPacket: is the message data type accessor
* PrintfFlush: is used to be able to print message in console

### implementation part of the file

#### declaration of fucntions

|  |
| --- |
| implementation {  void Status();  task void RadioSendTaskL();  task void RadioSendTaskC();  task void RadioSendTaskS();  task void BroadcastTask();  task void CalibrationTask();    task void SerialSendTaskL();  task void SerialSendTaskC();  task void SerialSendTaskS();    static void ReportProblem();  static void ReportSent();  static void ReportReceive();  static void FatalProblem();    static void ReportProblem() { printf("Report Problem!\n"); }  static void ReportSent() { printf("Report Sent!\n"); }  static void ReportReceive() { printf("Report Receive!\n"); }  static void FatalProblem() { printf("Report fatal!\n");} |

* RadioSendTaskL() / RadioSendTaskC() / RadioSendTaskS() / BroadcastTask() / CalibrationTask() are tasks which are used to send data over the radio to the root. 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.
* SerialSendTaskL() / SerialSendTaskC() / SerialSendTaskS() is a task which is used to send data over the serial interface to the parser.
* ReportProblem() prints a message
* ReportSent()prints a message
* ReportReceive() prints a message
* FatalProblem() prints a message

#### declaration of variables

|  |
| --- |
| message\_t SndMsgC, SndMsgL, SndMsgS, SndMsgB, UartBufC, UartBufL, UartBufS;    bool FwdBusy, SendBusy, UartBusy, TaskBusy;    bool Root = FALSE;  bool CheckForOldAnchors = FALSE;  bool FreqReq = FALSE;  int Freq = 26;    bool LightIsRead = FALSE;  bool TempIsRead = FALSE;  bool HumidityIsRead = FALSE;  bool ButtonIsPressed = FALSE;  bool VoltageIsRead = FALSE;    CollMsg\_t LocalCollMsg;  LocMsg\_t LocalLocMsg;  StatMsg\_t LocalStatMsg;  broadcast\_t local;  struct node  {  uint8\_t node\_ID;  uint8\_t VAN;  bool Checked;  struct node \*next;  };    struct node \*n = NULL; |

* SndMsgC, SndMsgL, SndMsgS, SndMsgB, UartBufC, UartBufL and UartBufS 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. The variables are used with the transmission off messages over RF and the serial port.
* SendBusy, UartBusy and TaskBusy are bools and are used in the program to signal when it is possible to send something with RF of uart.
* LocalCollMsg, LocalLocMsg, LocalStatMsg and local are linked to your structs from the header file.
* LightIsRead, TempIsRead, HumidityIsRead, ButtonIsPressed and VoltageIsRead are variables to check if sensor measurements are taken.
* Root is also a bool: we use it to see if the node is configured as the root or not.
* CheckForOldAnchors is a bool which is used to check for old anchor nodes in the linked list
* FreqReq is a bool used by the root node when a request is sent to change the frequency channel
* Freq is an integer used to store the parameter sent by an request
* Struct node: is used to create a linked list. This list will be used to store the id of the node to which a blind node is connected. If it is in range of three anchor nodes, it becomes a virtual anchor node. So, the network is multihop.

#### boot event

This event is signaled when the system has booted successfully. Components can assume the system has been initialized properly.

In this event we initialize:

* Data ( members of the structs and the bools)

|  |
| --- |
| event void Boot.booted()  {    LocalCollMsg.moteid = TOS\_NODE\_ID;  LocalCollMsg.lightreading = 0;  LocalCollMsg.tempreading = 0;  LocalCollMsg.humidityreading = 0;  LocalCollMsg.button = FALSE;  LocalCollMsg.type = 1;  LocalCollMsg.battery = 0;    LocalStatMsg.moteid = TOS\_NODE\_ID;  LocalStatMsg.type = 0;  LocalStatMsg.active = FALSE;  LocalStatMsg.AN = FALSE;  LocalStatMsg.posx = 0;  LocalStatMsg.posy = 0;  LocalStatMsg.sampleRate = DEFAULT\_SAMPLING\_PERIOD;  LocalStatMsg.locRate = DEFAULT\_LOC\_PERIOD;  LocalStatMsg.leds = 0;  LocalStatMsg.power = 31;  LocalStatMsg.frequency = 26;  LocalStatMsg.conn = 0;    LocalLocMsg.moteid = TOS\_NODE\_ID;  LocalLocMsg.ANmoteid = TOS\_NODE\_ID;  LocalLocMsg.VANs = 0;  LocalLocMsg.VANr = 0;  LocalLocMsg.hopCount = 0;  LocalLocMsg.RSSI = 0;    local.id = TOS\_NODE\_ID;  local.posx = 13;  local.posy = 37;    SendBusy = FALSE;  UartBusy = FALSE; |

* Radio (we start de radio)

|  |
| --- |
| if (call RadioControl.start() != SUCCESS)  FatalProblem(); |

* Collection service

|  |
| --- |
| if (call RoutingControlC.start() != SUCCESS)  FatalProblem();    if (call RoutingControlL.start() != SUCCESS)  FatalProblem();    if (call RoutingControlS.start() != SUCCESS)  FatalProblem(); |

* Dissemination service

|  |
| --- |
| if (call DisseminationControl.start() != SUCCESS)  FatalProblem(); |

* Serial interface (only with the root node)

|  |
| --- |
| if (TOS\_NODE\_ID == 0)  {  Root = TRUE;  if (call SerialControl.start() != SUCCESS)  FatalProblem();  } |

* Pushbutton

|  |
| --- |
| call Notify.enable(); |

* Printf

|  |
| --- |
| call PrintfControl.start(); |

#### startdone event of radiocontrol

This event is fired when the radiocontrol is started. When the radio is started we set the node with ID 0 as the root node of the three different collection trees.

We start de timer for sampling the sensors and the timer for broadcasting beacon messages.

When these commands are handled then we transmit a status message to the parser.

|  |
| --- |
| event void RadioControl.startDone(error\_t err) {  if (err != SUCCESS)  FatalProblem();  else  {  if (Root)  {  call RootControlL.setRoot();  call RootControlS.setRoot();  call RootControlC.setRoot();  }  //stop running Timers  if (call BroadcastTimer.isRunning())  call BroadcastTimer.stop();  if (call Timer.isRunning())  call Timer.stop();  //start the sensor timer  call Timer.startPeriodic(LocalStatMsg.sampleRate);  if(!Root)  {  call BroadcastTimer.startPeriodic(LocalStatMsg.locRate);  call Anchortimer.startPeriodic(60000);  }    Status();  }  } |

#### notify event

The notify event is handled when the pushbutton is pressed

|  |
| --- |
| event void Notify.notify(button\_state\_t state) {  if (state == BUTTON\_PRESSED)  LocalCollMsg.button = TRUE;  } |

#### status function

This function is used by nodes that aren’t the root node to transmit a status message to the parser.

First it checks if there aren’t any other tasks running. If not, then it proceeds to prepare the data to be transmitted with the RadioSendTaskS. If there is a task running then we place the message on a queue so it can be transmitted later in time.

Preparing the data happens with memcpy. The function memcpy(out, &LocalStatMsg, sizeof(StatMsg\_t) copies multiple fields of the struct to another and has 3 parameters:

* Out is the destination
* &LocalStatMsg is the source
* Sizeof(StatMsg \_t) is the size of the data that needs to be copied

|  |
| --- |
| void Status()  {  if (!Root)  {    StatMsg\_t \*out;    if (!TaskBusy)  {  out = (StatMsg\_t \*)call CollectSendS.getPayload(&SndMsgS);  memcpy(out, &LocalStatMsg, sizeof(StatMsg\_t));    printf("Status(): !root && !Taskbusy\n");    TaskBusy = TRUE;  post RadioSendTaskS();  }  else  {  message\_t \*NewMsg = call RadioPoolS.get();  if (NewMsg == NULL)  ReportProblem();  out = (StatMsg\_t \*)call CollectSendS.getPayload(NewMsg);  memcpy(out, &LocalStatMsg, sizeof(StatMsg\_t));  if (call RadioQueueS.enqueue(NewMsg) != SUCCESS) {  call RadioPoolS.put(NewMsg);  FatalProblem();  }  }  }  } |

#### event Serial receive

This event is called when a message is received over the serial interface. This message is a request sent by the GUI. The root will disseminated this request to the other nodes in the network.

In the special case, that the request is to change the frequency. The root will first disseminate this request to all the nodes in the network and then change the bool FreqReq to true. When the timer of the sensors fires then the root will change its own frequency so it can receives the messages on the new frequency. It must be implemented like this, because if you change it directly on receipt then the dissemination won’t work correctly.

|  |
| --- |
| event message\_t\* SerialReceive.receive(message\_t\* Msg, void\* Payload, uint8\_t Length)  {  DissMsg\_t \*NewRequest = Payload;    if (Length == sizeof(DissMsg\_t))  call RequestUpdate.change(NewRequest);  if (NewRequest->targetid == 0xFFFF && NewRequest->request == FREQUENCY\_REQUEST)  {  Freq = NewRequest->parameter;  FreqReq = TRUE;  }    call PrintfFlush.flush();  return Msg;  } |

#### event Requestvalue changed

This is event is called when the root node disseminates a request and a node receives it. It will call the function ProcessRequest to process the request.

|  |
| --- |
| event void RequestValue.changed() {  DissMsg\_t \*NewRequest = (DissMsg\_t \*)call RequestValue.get();  ProcessRequest(NewRequest);  } |

#### function ProcessRequest

This function will process all the possible requests:

* DEFAULT\_SAMPLING\_PERIOD: period for sampling the sensors of the node
* DEFAULT\_LOC\_PERIOD: period for broadcasting beacon message
* LOC\_PERIOD: request to change location period
* SAMPLING\_PERIOD: request to change sampling period of the sensors
* ACTIVE\_REQUEST: request to change the status of active
* AN\_REQUEST: request to change the node to an anchor or blind node
* LED\_REQUEST: request to change the leds
* POWER\_REQUEST: request to change the power level of transmission
* FREQUENCY\_REQUEST: request to change the frequency
* POSX\_REQUEST: request to change the x coordinate of the node
* POSY\_REQUEST: request to change the y coordinate of the node
* STATUS\_REQUEST: request to receive a status report

|  |
| --- |
| void ProcessRequest(DissMsg\_t \*NewRequest) {  if ((NewRequest->targetid == TOS\_NODE\_ID) || (NewRequest->targetid == 0xFFFF)) {  switch (NewRequest->request) {    case SAMPLING\_PERIOD:  LocalStatMsg.sampleRate = NewRequest->parameter;  call Timer.stop();  call Timer.startPeriodic(LocalStatMsg.sampleRate);  break;    case LOC\_PERIOD:  if (LocalStatMsg.AN == TRUE)  {  LocalStatMsg.locRate = NewRequest->parameter; call BroadcastTimer.stop();  call BroadcastTimer.startPeriodic(LocalStatMsg.locRate);  }  break;    case ACTIVE\_REQUEST:  if(NewRequest->parameter == 1)  {  LocalStatMsg.active = TRUE;  if(LocalStatMsg.AN == TRUE)  call BroadcastTimer.startPeriodic(LocalStatMsg.locRate);    }  else if(NewRequest->parameter == 0)  {  LocalStatMsg.active = FALSE;    if(LocalStatMsg.AN == TRUE)  call BroadcastTimer.stop();    }  break;    case AN\_REQUEST:  if(NewRequest->parameter == 1)  {  LocalStatMsg.AN = TRUE;  LocalLocMsg.VANr = 1;  local.VAN = 1;  }  else if(NewRequest->parameter == 0)  {  if (call BroadcastTimer.isRunning())  call BroadcastTimer.stop();  LocalLocMsg.VANr = 0;  local.VAN = 0;  LocalStatMsg.AN = FALSE;  }  break;    case LED\_REQUEST:  call Leds.set(NewRequest->parameter);  LocalStatMsg.leds = NewRequest->parameter;  break;    case POWER\_REQUEST:  LocalStatMsg.power = NewRequest->parameter;  break;    case FREQUENCY\_REQUEST:  LocalStatMsg.frequency = NewRequest->parameter;  call CC2420Config.setChannel(LocalStatMsg.frequency);  call CC2420Config.sync();  break;    case POSX\_REQUEST:  LocalStatMsg.posx = NewRequest->parameter;  break;    case POSY\_REQUEST:  LocalStatMsg.posy = NewRequest->parameter;  break;    case STATUS\_REQUEST:  Status();  break;    default:  break;  }  }  } |

#### event timer fired

The event is handled when the timer fires. In this event we change the frequency channel of the root if such a request has been made. We also take sensor measurements and if all the sensors are measured, we will transmit the data with the task RadioSendTaskC. If the radio is busy then the data will be put on a Queue.

|  |
| --- |
| event void Timer.fired() {  if( FreqReq == TRUE )  {  FreqReq == FALSE;  call CC2420Config.setChannel(Freq);  call CC2420Config.sync();  }    if (call ReadLight.read() == SUCCESS)  LightIsRead = TRUE;  if (call ReadTemp.read() == SUCCESS)  TempIsRead = TRUE;  if (call ReadHumidity.read() == SUCCESS)  HumidityIsRead = TRUE;  if (call ReadBattery.read() == SUCCESS)  VoltageIsRead = TRUE;    if (LightIsRead && TempIsRead && HumidityIsRead && VoltageIsRead) {  if (!Root) {    CollMsg\_t \*out;  if (!SendBusy)  {  out = (CollMsg\_t \*)call CollectSendC.getPayload(&SndMsgC);  memcpy(out, &LocalCollMsg, sizeof(CollMsg\_t));  TaskBusy = TRUE;  post RadioSendTaskC();  }  else  {  message\_t \*NewMsg = call RadioPoolC.get();  if (NewMsg == NULL)  ReportProblem();  out = (CollMsg\_t \*)call CollectSendC.getPayload(NewMsg);  memcpy(out, &LocalCollMsg, sizeof(CollMsg\_t));  if (call RadioQueueC.enqueue(NewMsg) != SUCCESS)  {  call RadioPoolC.put(NewMsg);  FatalProblem();  }  }  }  }  } |

#### event readdone

When the read() command is executed, it generates a readDone event. This event delivers the data from the sensor. We have 4 sensors:

* Light
* Temperature
* Humidity
* Voltage

|  |
| --- |
| event void ReadLight.readDone(error\_t err, uint16\_t reading) {  if (err != SUCCESS) {  reading = 0xFFFF;  ReportProblem();  }  else  {  LocalCollMsg.lightreading = reading;  }  }    event void ReadTemp.readDone(error\_t err, uint16\_t reading) {  if (err != SUCCESS) {  reading = 0xFFFF;  ReportProblem();  }  else {  LocalCollMsg.tempreading = reading;  }  }  event void ReadHumidity.readDone(error\_t err, uint16\_t reading) {  if (err != SUCCESS) {  reading = 0xFFFF;  ReportProblem();  }  else {  LocalCollMsg.humidityreading = reading;  }  }    event void ReadBattery.readDone(error\_t err, uint16\_t reading) {  if (err != SUCCESS) {  reading = 0xFFFF;  ReportProblem();  }  else  {  LocalCollMsg.battery = reading;  }  } |

#### Event Collect receive C

In this event, the root node receives packages with sensor data from the other nodes, transmitted with the RadioSendTaskC task.

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

The root will transmit the data over the serial interface to the computer (parser) with the task SerialSendTaskC if it isn’t busy. If it is busy, the message will be put on a queue. When the queue has no more space for new messages, then these will be dropped.

|  |
| --- |
| event message\_t\* CollectReceiveC.receive(message\_t\* Msg, void\* Payload, uint8\_t Length) {  CollMsg\_t\* inColl = (CollMsg\_t\*)Payload;  CollMsg\_t\* outColl;  if (UartBusy == FALSE)  {  outColl = (CollMsg\_t\*)call SerialSendC.getPayload(&UartBufC);  if (Length != sizeof(CollMsg\_t))  return Msg;  else  memcpy(outColl, inColl, sizeof(CollMsg\_t));    post SerialSendTaskC();  }  else  {  message\_t \*NewMsg = call UartPoolC.get();  if (NewMsg == NULL)  {  ReportProblem();  return Msg;  }  outColl = (CollMsg\_t\*)call SerialSendC.getPayload(NewMsg);  memcpy(outColl, inColl, sizeof(CollMsg\_t));  if (call UartQueueC.enqueue(NewMsg) != SUCCESS) {  call UartPoolC.put(NewMsg);  FatalProblem();  return Msg;  }  }  return Msg;    } |

#### Event Collect receive L

In this event, the root node receives packages with location data from the other nodes, transmitted with the RadioSendTaskL task.

This event is similar to the previous event.

|  |
| --- |
| event message\_t\* CollectReceiveL.receive(message\_t\* Msg, void\* Payload, uint8\_t Length) {  LocMsg\_t\* inLoc = (LocMsg\_t\*)Payload;  LocMsg\_t\* outLoc;    if (UartBusy == FALSE)  {  outLoc = (LocMsg\_t\*)call SerialSendL.getPayload(&UartBufL);  if (Length != sizeof(LocMsg\_t))  return Msg;  else  memcpy(outLoc, inLoc, sizeof(LocMsg\_t));    post SerialSendTaskL();  }  else  {  message\_t \*NewMsg = call UartPoolS.get();  if (NewMsg == NULL)  {  ReportProblem();  return Msg;  }  outLoc = (LocMsg\_t\*)call SerialSendL.getPayload(NewMsg);  memcpy(outLoc, inLoc, sizeof(LocMsg\_t));  if (call UartQueueL.enqueue(NewMsg) != SUCCESS) {  call UartPoolL.put(NewMsg);  FatalProblem();  return Msg;  }  }  return Msg;    } |

#### Event Collect receive S

In this event, the root node receives packages with data about the status of the node, transmitted with the RadioSendTaskS task.

This event is similar to the previous event.

|  |
| --- |
| event message\_t\* CollectReceiveS.receive(message\_t\* Msg, void\* Payload, uint8\_t Length) {  StatMsg\_t\* inStat = (StatMsg\_t\*)Payload;  StatMsg\_t\* outStat;    if (UartBusy == FALSE)  {  outStat = (StatMsg\_t\*)call SerialSendS.getPayload(&UartBufS);  if (Length != sizeof(StatMsg\_t))  return Msg;  else  memcpy(outStat, inStat, sizeof(StatMsg\_t));    post SerialSendTaskS();  }  else  {  message\_t \*NewMsg = call UartPoolS.get();  if (NewMsg == NULL)  {  ReportProblem();  return Msg;  }  outStat = (StatMsg\_t\*)call SerialSendS.getPayload(NewMsg);  memcpy(outStat, inStat, sizeof(StatMsg\_t));  if (call UartQueueL.enqueue(NewMsg) != SUCCESS) {  call UartPoolL.put(NewMsg);  FatalProblem();  return Msg;  }  }  return Msg;    } |

#### event snoop receive

These events are used to listen to transmitted message in the three different collection trees (sensor, location and status tree)

|  |
| --- |
| event message\_t\* SnoopC.receive(message\_t\* Msg, void\* Payload, uint8\_t Length) {  return Msg;  }  event message\_t\* SnoopL.receive(message\_t\* Msg, void\* Payload, uint8\_t Length) {  return Msg;  }  event message\_t\* SnoopS.receive(message\_t\* Msg, void\* Payload, uint8\_t Length) {  return Msg;  } |

#### event broadcast receive

This event is handled when a blind node is active and receives a broadcast message from an anchor node. The node checks if the id from where the packet came from, is already in the linked list. If not, then it will be added. It also checks if the radio is free to transmit the location message with the task RadioSendTaskL to the root node. If the radio is busy then the message will be put on a queue.

|  |
| --- |
| event message\_t\* BroadcastReceive.receive(message\_t\* msg, void\* payload, uint8\_t len) {    broadcast broadcast\_t \*omsg = (broadcast\_t\*)payload;  LocMsg\_t \*out;    if (!Root && LocalStatMsg.active && !LocalStatMsg.AN )  {  LocalLocMsg.ANmoteid = omsg->id;  LocalLocMsg.VANs = omsg->VAN;  LocalLocMsg.RSSI = call CC2420Packet.getRssi(msg);  LocalLocMsg.hopCount = omsg->hopCount;  LocalStatMsg.power = call CC2420Packet.getPower(msg);    if (list\_search( &n, LocalLocMsg.ANmoteid, LocalLocMsg.VANs) == NULL )  addToList(&n, LocalLocMsg.ANmoteid, LocalLocMsg.VANs);  if (!SendBusy)  {  out = (LocMsg\_t \*)call CollectSendL.getPayload(&SndMsgL);    memcpy(out, &LocalLocMsg, sizeof(LocMsg\_t));  TaskBusy = TRUE;  post RadioSendTaskL();  }  else  {  message\_t \*NewMsg = call RadioPoolL.get();  if (NewMsg == NULL)  ReportProblem();  out = (LocMsg\_t \*)call CollectSendL.getPayload(NewMsg);  memcpy(out, &LocalLocMsg, sizeof(LocMsg\_t));  if (call RadioQueueL.enqueue(NewMsg) != SUCCESS)  {  call RadioPoolL.put(NewMsg);  FatalProblem();  }  }  }  call PrintfFlush.flush();  return msg;  } |

#### broadcast timer fired

When this timer is fired, an anchor node broadcast a beacon message to the blind nodes with the task BroadcastTask.

|  |
| --- |
| event void BroadcastTimer.fired() {    if (!SendBusy && !Root && LocalStatMsg.AN)  {  broadcast\_t \* o = (broadcast\_t \*)call BroadcastSend.getPayload(&SndMsgB);    memcpy(o, &local, sizeof(broadcast\_t));  TaskBusy = TRUE;  post BroadcastTask();    }  else  ReportProblem();  } |

#### Broadcast task

This task broadcasts a beacon message to the blind nodes. The blind nodes use this message to get the RSSI of the anchor node. The bool SendBusy is set to TRUE, because the radio will be busy.

|  |
| --- |
| task void BroadcastTask()  {  if (call BroadcastSend.send(AM\_BROADCAST\_ADDR, &SndMsgB, sizeof(broadcast\_t)) == SUCCESS)  {  SendBusy = TRUE;  TaskBusy = FALSE;  }  else  ReportProblem();  } |

#### event broadcastsend senddone

This event is handled when the transmission of the beacon message is finished, so we put the bool SendBusy back to false.

|  |
| --- |
| event void BroadcastSend.sendDone(message\_t\* msg, error\_t error) {  SendBusy = FALSE;  } |

#### event calibration receive

This event is handled when an anchor node is active and receives a broadcast message from an another anchor node. It checks if the radio is free to transmit the location message with the purpose of calibration with the task RadioSendTaskL to the root node. If the radio is busy then the message will be put on a queue.

|  |
| --- |
| broadcast\_t \*omsg = (broadcast\_t\*)payload;  printf(" Received calibration message from node: %u\n", omsg->id);  if (!Root && LocalStatMsg.AN )  {  LocMsg\_t \*out;    LocalLocMsg.ANmoteid = omsg->id;  LocalLocMsg.VANs = omsg->VAN;  LocalLocMsg.RSSI = call CC2420Packet.getRssi(msg);  LocalLocMsg.hopCount = omsg->hopCount;  if (!SendBusy)  {  out = (LocMsg\_t \*)call CollectSendL.getPayload(&SndMsgL);  memcpy(out, &LocalLocMsg, sizeof(LocMsg\_t));  TaskBusy = TRUE;  post RadioSendTaskL();  }  else  {  message\_t \*NewMsg = call RadioPoolL.get();  if (NewMsg == NULL)  ReportProblem();  out = (LocMsg\_t \*)call CollectSendL.getPayload(NewMsg);  memcpy(out, &LocalLocMsg, sizeof(LocMsg\_t));  if (call RadioQueueL.enqueue(NewMsg) != SUCCESS)  {  call RadioPoolL.put(NewMsg);  FatalProblem();  }    }  }  call PrintfFlush.flush();  return msg;  } |

#### Anchor timer fired

When this timer is fired, an anchor node broadcasts a beacon message to the anchor nodes with the task CalibrationTask. It also put the nodes in the linked list to the status of unchecked. The next time the timer fires the nodes that still have the status unchecked will be removed from the list.

|  |
| --- |
| event void Anchortimer.fired(){  if ( !SendBusy && !Root && LocalStatMsg.active && LocalStatMsg.AN )  {  broadcast\_t \* o = (broadcast\_t \*)call CalibrationSend.getPayload(&SndMsgB);  memcpy(o, &local, sizeof(broadcast\_t));  TaskBusy = TRUE;  post CalibrationTask();  }  else  ReportProblem();    if(CheckForOldAnchors)  {  list\_remove(InactiveNodes(&n));  }  else  SetStatusToFalse(&n);    CheckForOldAnchors = !CheckForOldAnchors;    } |

#### Calibration task

This task broadcasts a beacon message to the anchor nodes. The anchor nodes use this message to get the RSSI of the anchor node. The bool SendBusy is set to TRUE, because the radio will be busy.

|  |
| --- |
| task void BroadcastTask()  {  if (call CalibrationSend.send(AM\_BROADCAST\_ADDR, &SndMsgB, sizeof(broadcast\_t)) == SUCCESS)  {  SendBusy = TRUE;  TaskBusy = FALSE;  }  else  ReportProblem();  } |

#### event broadcastsend senddone

This event is handled when the transmission of the beacon message is finished, so we put the bool SendBusy back to false.

|  |
| --- |
| event void CalibrationSend.sendDone(message\_t\* msg, error\_t error) {  SendBusy = FALSE;  } |

#### Radiosend C task

This task is used to transmit sensor messages to the root node of the collection tree. First we check if the node is not the root node and if the radio isn’t busy.

When the message is transmitted we place the status of the button back to false, other while we can’t detect if the button has been pressed again.

|  |
| --- |
| task void RadioSendTaskC()  {  if (!Root && !SendBusy)  {  if (call CollectSendC.send(&SndMsgC, sizeof(CollMsg\_t)) == SUCCESS)  {  SendBusy = TRUE;  TaskBusy = FALSE;  }  else  ReportProblem();  }  LocalCollMsg.button = FALSE;  } |

#### event Collectsend C senddone

This event is handled when the transmission of the sensor message is finished. We put the bool SendBusy to false because the transmission is over.

Next, we check if there are still sensor messages on the queue that need to be transmitted.

|  |
| --- |
| event void CollectSendC.sendDone(message\_t\* Msg, error\_t err) {    SendBusy = FALSE;  if (call RadioQueueC.empty() == FALSE)  {  message\_t \*QueueMsg = call RadioQueueC.dequeue();  if (QueueMsg == NULL) {  FatalProblem();  return;  }  memcpy(&SndMsgC, QueueMsg, sizeof(CollMsg\_t));  if (call RadioPoolC.put(QueueMsg) != SUCCESS) {  FatalProblem();  return;  }    post RadioSendTaskC();  }    ReportSent();  } |

#### Radiosend L task

This task is used to transmit the status messages to the root node of the collection tree. First we check if the node is not the root node and if the radio isn’t busy.

|  |
| --- |
| task void RadioSendTaskL()  {  if (!Root && !SendBusy)  {  if (call CollectSendL.send(&SndMsgL, sizeof(LocMsg\_t)) == SUCCESS)  {  SendBusy = TRUE;  TaskBusy = FALSE;  }  else  ReportProblem();  }  } |

#### event Collectsend L senddone

This event is handled when the transmission of the location message is finished. We put the bool SendBusy to false because the transmission is over.

Next, we check if there are still location messages on the queue that need to be transmitted.

|  |
| --- |
| event void CollectSendL.sendDone(message\_t\* Msg, error\_t err) {  SendBusy = FALSE;    if (call RadioQueueL.empty() == FALSE)  {  message\_t \*QueueMsg = call RadioQueueL.dequeue();  if (QueueMsg == NULL) {  FatalProblem();  return;  }  memcpy(&SndMsgL, QueueMsg, sizeof(LocMsg\_t));  if (call RadioPoolL.put(QueueMsg) != SUCCESS) {  FatalProblem();  return;  }  post RadioSendTaskL();  }    ReportSent();  } |

#### Radiosend S task

This task is used to transmit the location messages to the root node of the collection tree. First we check if the node is not the root node and if the radio isn’t busy.

|  |
| --- |
| task void RadioSendTaskS()  {  if (!Root && !SendBusy)  {  if (call CollectSendS.send(&SndMsgS, sizeof(StatMsg\_t)) == SUCCESS)  {  SendBusy = TRUE;  TaskBusy = FALSE;  }  else  ReportProblem();  }  } |

#### event Collectsend S senddone

This event is handled when the transmission of the status message is finished. We put the bool SendBusy to false because the transmission is over.

Next, we check if there are still status messages on the queue that need to be transmitted.

|  |
| --- |
| event void CollectSendS.sendDone(message\_t\* Msg, error\_t err) {  SendBusy = FALSE;  if (call RadioQueueS.empty() == FALSE)  {  message\_t \*QueueMsg = call RadioQueueS.dequeue();  if (QueueMsg == NULL) {  FatalProblem();  return;  }  memcpy(&SndMsgS, QueueMsg, sizeof(StatMsg\_t));  if (call RadioPoolS.put(QueueMsg) != SUCCESS) {  FatalProblem();  return;  }    post RadioSendTaskS();  }  ReportSent();  } |

#### serialsend C task

This task is used to transmit the sensor messages to a computer (parser). When the message is being sent, then the serial interface is busy and we put the bool UartBusy to true.

SerialSendC is bound to SerialAMSenderC.AMSend (configuration file) and sends the received data to the computer through the serial port. So SerialSendC.send(0xFFFF, &UartbufC, Uartlen) broadcasts a packet with a data payload of a certain length to the broadcast address. It has 3 parameters:

* 0xFFFF is the address of the destination
* &UartbufC is the message with the data
* Uartlen is the length of the payload

|  |
| --- |
| task void SerialSendTaskC() {    UartLen = sizeof(CollMsg\_t);  if (call SerialSendC.send(0xFFFF, &UartBufC, UartLen) == SUCCESS)  {  UartBusy = TRUE;  }  else  ReportProblem();  } |

#### event serialsend C senddone

This event is handled when the status message is sent over the serial interface to the computer (parser). We put the bool UartBusy to false, because the serial interface is free.

We check if there are still messages on the queue that need to be transmitted over the serial interface.

|  |
| --- |
| event void SerialSendC.sendDone(message\_t\* Msg, error\_t err) {  UartBusy = FALSE;  if (call UartQueueC.empty() == FALSE) {  message\_t \*QueueMsg = call UartQueueC.dequeue();  if (QueueMsg == NULL) {  FatalProblem();  return;  }  memcpy(&UartBufC, QueueMsg, sizeof(CollMsg\_t));  if (call UartPoolC.put(QueueMsg) != SUCCESS) {  FatalProblem();  return;  }  post SerialSendTaskC();  }  ReportSent();  } |

#### serialsend L task

This task is used to transmit the location messages to a computer (parser). When the message is being sent, then the serial interface is busy and we put the bool UartBusy to true.

This task is similar to the SerialSendC task.

|  |
| --- |
| task void SerialSendTaskL() {    UartLen = sizeof(LocMsg\_t);  if (call SerialSendL.send(0xFFFF, &UartBufL, UartLen) == SUCCESS)  {  UartBusy = TRUE;  }  else  ReportProblem();  } |

#### event serialsend L senddone

This event is handled when the location message is sent over the serial interface to the computer (parser). We put the bool UartBusy to false, because the serial interface is free.

We check if there are still messages on the queue that need to be transmitted over the serial interface.

|  |
| --- |
| event void SerialSendL.sendDone(message\_t\* Msg, error\_t err) {  UartBusy = FALSE; if (call UartQueueL.empty() == FALSE) {  message\_t \*QueueMsg = call UartQueueL.dequeue();  if (QueueMsg == NULL) {  FatalProblem();  return;  }  memcpy(&UartBufL, QueueMsg, sizeof(LocMsg\_t));  if (call UartPoolL.put(QueueMsg) != SUCCESS) {  FatalProblem();  return;  }  post SerialSendTaskL();  }  ReportSent();  } |

#### serialsend S task

This task is used to transmit the status messages to a computer (parser). When the message is being sent, then the serial interface is busy and we put the bool UartBusy to true.

This task is similar to the SerialSendC task.

|  |
| --- |
| task void SerialSendTaskS()  {  UartLen = sizeof(StatMsg\_t);  if (call SerialSendS.send(0xFFFF, &UartBufS, UartLen) == SUCCESS)  {  UartBusy = TRUE;  }  else  ReportProblem();  } |

#### event serialsend S senddone

This event is handled when the status message is sent over the serial interface to the computer (parser). We put the bool UartBusy to false, because the serial interface is free.

We check if there are still messages on the queue that need to be transmitted over the serial interface.

|  |
| --- |
| event void SerialSendS.sendDone(message\_t\* Msg, error\_t err) {  UartBusy = FALSE; if (call UartQueueS.empty() == FALSE) {  message\_t \*QueueMsg = call UartQueueS.dequeue();  if (QueueMsg == NULL) {  FatalProblem();  return;  }  memcpy(&UartBufS, QueueMsg, sizeof(StatMsg\_t));  if (call UartPoolS.put(QueueMsg) != SUCCESS) {  FatalProblem();  return;  }  post SerialSendTaskS();  }  ReportSent();  } |

#### Linked list

Add a node to the linked list, this is used by a blind node to know to which nodes it is connected to:

Parameter 1: pointer to the linked list

Parameter 2: the id of the node that you want to add

Parameter 3: Virtual anchor node or not

|  |
| --- |
| struct node \*addToList(struct node \*\*p, uint8\_t id, uint8\_t VANs)  {  struct node \*n = malloc(sizeof(struct node));  if (n == NULL)  {  ReportProblem();  ReportSent();  ReportReceive();  }    n->next = \*p; /\* the previous element (\*p) now becomes the "next" element \*/  \*p = n; /\* add new empty element to the front (head) of the list \*/  n->node\_ID = id;  n->VAN = VANs;  n->Checked = TRUE;    return \*p;  } |

Check if the node is in range of 3 anchor or virtual anchor nodes.

Parameter 1: pointer to the linked list

|  |
| --- |
| struct node \*\*VAN(struct node \*\*n)  {  uint8\_t ans = 0;  uint8\_t vans = 0;    while (\*n != NULL)  {  if ((\*n)->VAN == 1)  {  ans++;  printf("VAN: Found anchornode %u in list \n", (\*n)->node\_ID);  }  else if ((\*n)->VAN == 2)  {  vans++;  printf("VAN: Found Virtual anchornode %u in list \n", (\*n)->node\_ID);  }  n = &(\*n)->next;  }  if ( (ans + vans) >= 3)  return 2;  else  return 0;  } |

Search in the list for a node with the id and put the status to checked if it is in the list

Parameter 1: pointer to the linked list

Parameter 2: the id of the node that you want to search

Parameter 3: Virtual anchor node or not

|  |
| --- |
| struct node \*\*list\_search(struct node \*\*n, uint8\_t id, uint8\_t VANs)  {  while (\*n != NULL)  {  if ((\*n)->node\_ID == id)  {  (\*n)->VAN = VANs;  (\*n)->Checked = TRUE;  return n;  }  n = &(\*n)->next;  }  return NULL;  } |

Check which nodes are inactive in the network

Parameter 1: pointer to the linked list

|  |
| --- |
| struct node \*\*InactiveNodes(struct node \*\*n)  {  while (\*n != NULL)  {  if ((\*n)->Checked == FALSE)  {  return n;  }  n = &(\*n)->next;  }  return NULL;  } |

Print out the elements in the list.

Parameter 1: pointer to the linked list

|  |
| --- |
| void list\_print(struct node \*n)  {  if (n == NULL)  {  printf("list is empty\n");  }  while (n != NULL)  {  printf("Data in list %u\n", n->node\_ID);  n = n->next;  }  call PrintfFlush.flush();  } |

Remove inactive nodes from the list.

Parameter 1: pointer to the linked list

|  |
| --- |
| void list\_remove(struct node \*\*p) /\* remove head \*/  {  if (\*p != NULL)  {  struct node \*n = \*p;  \*p = (\*p)->next;  free(n);  }  } |

To check if nodes are inactive, we set their status to false when the timer fires. If no packet is received in a certain time then the nodes will be removed from the list.

Parameter 1: pointer to the linked list

|  |
| --- |
| void SetStatusToFalse(struct node \*\*n)  {  while (\*n != NULL)  {  (\*n)->Checked = TRUE;  n = &(\*n)->next;  }  } |