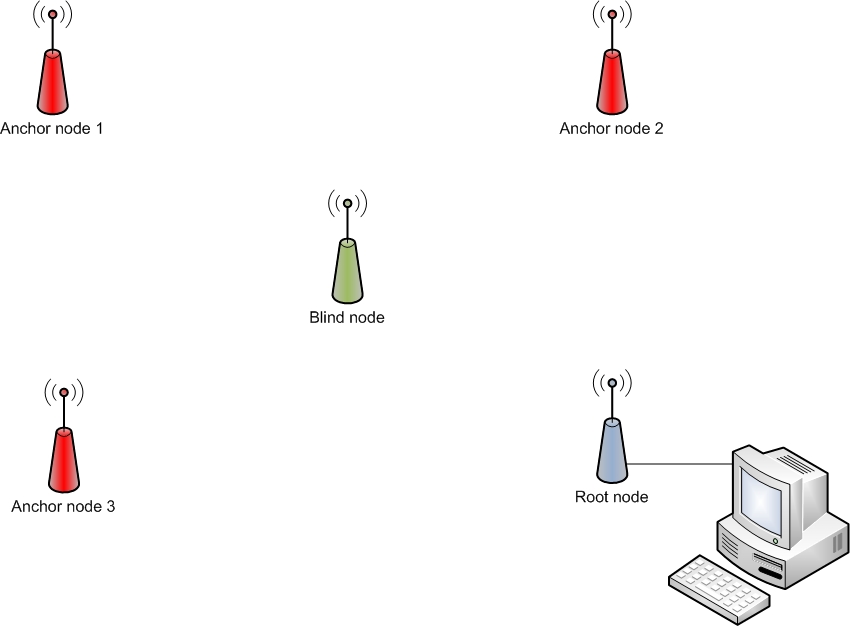
Telos rev.b application

Senseless

# What

## 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.

### Telos rev. b

Telos rev. B 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

### 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.

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 a 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: Telos rev. B 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.

## software

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.



## installation off the application

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.