We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900
Open access books available

186,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

5

Rapid application development for wireless sensor networks

Mohammad Mostafizur Rahman Mozumdar
and Luciano Lavagno
Politecnico di Torino, Torino
Italy
Laura Vanzago
STMicroelectronics, Milano
Italy

1. Introduction

In the last decade, the landscape of wireless sensor network (WSN) applications has been extending rapidly in many fields such as factory and building automation, environmental monitoring, security systems and in a wide variety of commercial and military areas. Advancements in microelectro-mechanical systems and wireless communication have motivated the development of small and low power sensors and radio equipped modules which are now replacing traditional wired sensor systems. These tiny modules usually called "motes" can communicate with each other by radio and act like as neurons to collect information from the environment. Platforms for WSNs, including processors, sensors, radios, power supplies, operating systems and protocol stacks, are almost as diverse as the application areas, with only a few standards (e.g. TinyOS (Levis et al., 2004) and the ZigBee (2006) protocol), which are still far from being universally recognized and truly interoperable.

Application development for WSNs is quite challenging, because in principle it would require both detailed knowledge of the application area and of the available hardware and software platforms. Moreover, design aids, in the form of both functional simulation, power and performance analysis and on-target debugging are still very rudimentary. Many hardware and software platforms include only LEDs as a debugging aid.

The available functional analysis packages, such as TOSSIM (Levis et al., 2003) for debugging of TinyOS application, OmNet (1992) and NS-2 (2001), fall into two main categories. One is very platform- and OS-specific (such as TOSSIM), and provides essentially a binary API to model the OS and the motes, with limited facilities for re-using existing channel models, tracing, collecting statistics and so on. The other are generic network simulators (such as OmNet, NS, etc.), sometimes enhanced with models tailored to the radios and channels used by WSNs. Both have significant drawbacks when it comes to complex application development. The first group makes it virtually impossible to port an application to a different platform (e.g. from TinyOS to MANTIS (Bhatti et al., 2005) or to a

ZigBee compliant platform or vice versa). The second group still leaves a lot of detailed platform-dependent code to be developed and debugged. Integrated use of a network simulator followed by a platform simulator is the most commonly used path, but still requires one to port code between a number of environments. Moreover, in case a bug is found at the end, one has to resort to led-based debugging, which is extremely time consuming.

In order to solve these problems, we wanted to be able to model the application using high level abstractions, and simulate it using configurable and realistic topologies for the network itself. Then we wanted to be able to automatically generate code for several target operating systems. In this chapter, we present a framework (Mozumdar et al., 2008a) for modeling, simulation and automatic code generation of sensor network applications based on MathWorks (1984) tools. In our framework, applications can be modeled using Stateflow state charts (SF. 2009) (and Simulink block diagrams, even though StateCharts were the best tool for the application we considered as case study). Then the application developer can configure the connectivity of the sensor network nodes and can perform behavioral simulation and functional verification of the application. After modeling and simulation, this framework can generate the complete application code for several target operating systems from the simulated model.

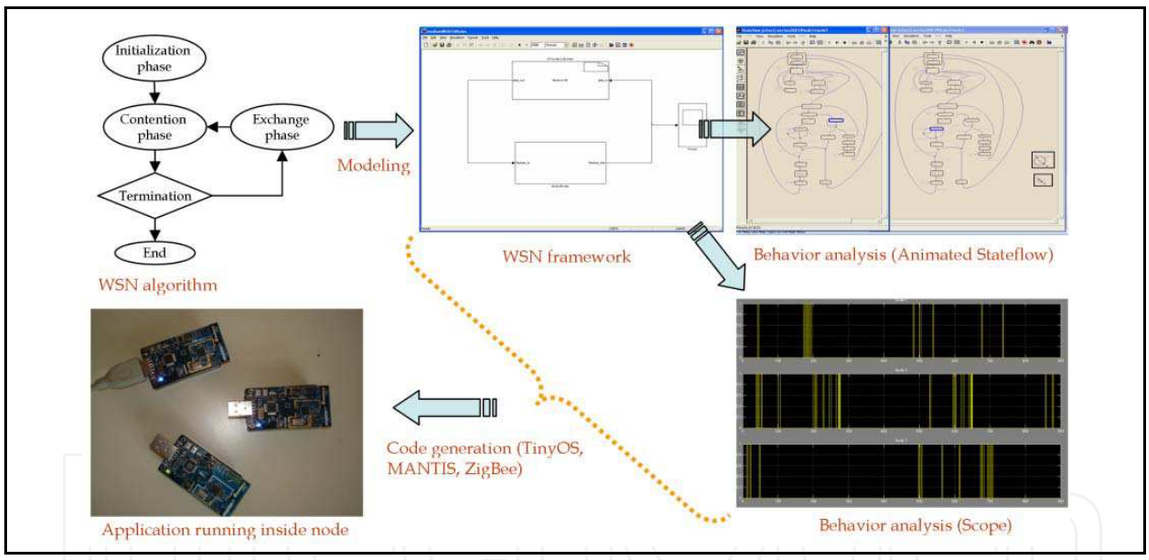


Fig. 1. A complete view of the framework

The application developer can thus use the broad variety of debugging and analysis tools provided by MathWorks, such as animated state chart displays, scopes, plots, as well as exploit a large number of available pre-designed Simulink blocks. To the best of our knowledge, this is the first time that a framework of this sort has been developed and tested. A complete view of the whole framework is depicted in figure 1.

While working on code generation for the various kinds of target platforms, we also identified a coding style for functions written in ANSI C that maximizes the ease of porting the code, especially if coupled with the basic platform abstraction API that we developed.

In this chapter, we use as example target platforms TinyOS, MANTIS and the Ember implementation of the standard ZigBee, since they provide very different programming

models and abstractions (e.g. non-preemptive scheduler with split-phase coding versus multi-threaded kernel). Hence they are maximally different representatives of the programming platforms used by WSN developers.

In (Cheong et al., 2005), a graphical development and simulation environment for TinyOS-based applications called *Viptos* is described. *Viptos* provides graphical development and interrupt-level simulation of actual TinyOS programs, with packet-level simulation of the network. It also allows the developer to use other models of computation available in Ptolemy II (Eker et al, 2003) for modeling various parts of the system. To model an algorithm using *Viptos*, the users are bound to code it for TinyOS, which implies that the user should have sufficient knowledge of TinyOS. In our framework, the users can model the application by using Stateflow and need not have any knowledge of TinyOS, MANTIS or ZigBee. In short, our framework provides more freedom by decoupling the application from the platform and also supports several platforms for code generation.

In (Vieira et al.2005), the authors describe a visual development framework for multi-platform wireless sensor networks, which is capable of generating application code for TinyOS and Yet Another Tiny Operating System (Yatos) (Almeida et al. 2003). This tool supports only code generation of the developed model for the WISDOM (Vieira et al. 2005) framework and it does not support functional verification of the designed model. Here also the model development is biased to TinyOS and Yatos, since these two target platforms share the same component based programming style.

The idea of generating WSN application code from a single higher level abstraction has also been demonstrated in (Abdelzaher et al., 2004, Gummadi et al., 2005, Newton & Welsh 2004, Bakshi et al., 2005) using functional and macro-programming. All these approaches introduce new programming languages, while in our case we advocate either to use a specific programming style in C, or to use an existing well-known graphical language (Stateflow). Although the approaches listed above introduce higher level abstractions, they did not propose a methodology to generate application code for multiple software platforms (all of these approaches generate application code only for TinyOS). In this chapter, we identified a single programming style (Mozumdar et al, 2008b) that is compatible with most kinds of WSN software platforms (e.g. MANTIS, TinyOS and Zigbee).

2. Methodology

The complete framework for modeling, simulation and automatic code generation is depicted in figure 2. The WSN algorithm (application, middleware or device drivers) will be at first modeled by using Simulink and Stateflow blocks. We have designed blocks that specifically help WSN modeling such as the *sensor node* and *communication medium* described later. These blocks are completely parameterized and can be used for model development like usual Simulink blocks. *Sensor node* blocks are connected to the *communication medium* block which provides a mechanism for the application developer to define the connectivity between the nodes in sensor network.

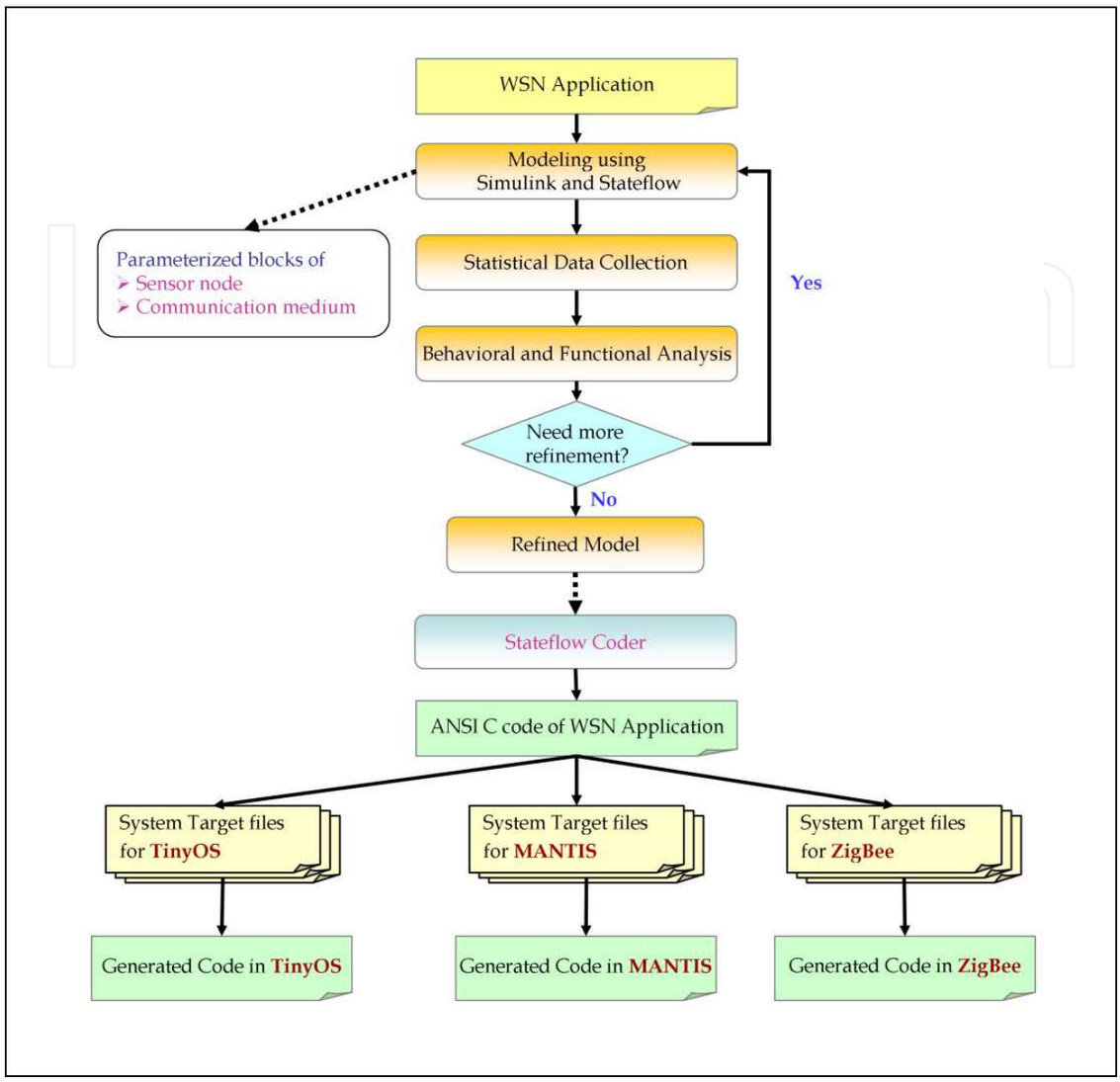


Fig. 2. Framework for modeling, simulation and code generation of WSN application

The *communication medium* block is implemented in C, so it can be modified to reuse any existing channel and connectivity models. The *sensor node* contains mainly a timing generator, a random number generator, and a parameterized Stateflow block which actually implements the application running inside each single node (shown in figure 3). The application block is a library object and each *sensor node* contains an instance of it. Therefore, every node of the framework is running an independent copy of same algorithm. It is of course also possible to model sensor networks having different algorithms running in different nodes. In that case, one needs to create a small Stateflow library and instantiate objects from it as needed. To model a new sensor network application based on this framework, the application developer only needs to modify the template algorithm implementation and set the connectivity of the nodes in the *communication medium* block. Then simulation can be started and statistical data can be collected using animated state charts, scopes and displays to perform functional analysis of the algorithm. The algorithm

implementation can be refined if the analysis of the results suggest to do so. Eventually the developer will get a refined model which represents the desired behavior.

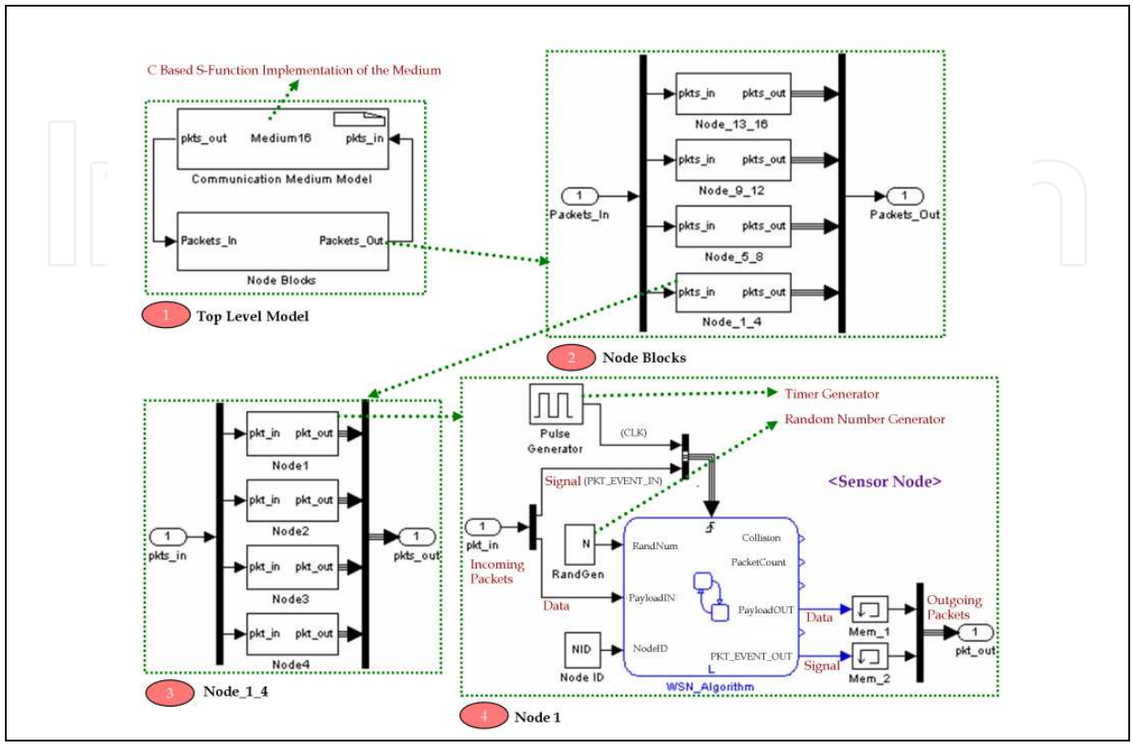


Fig. 3. A simple simulation framework

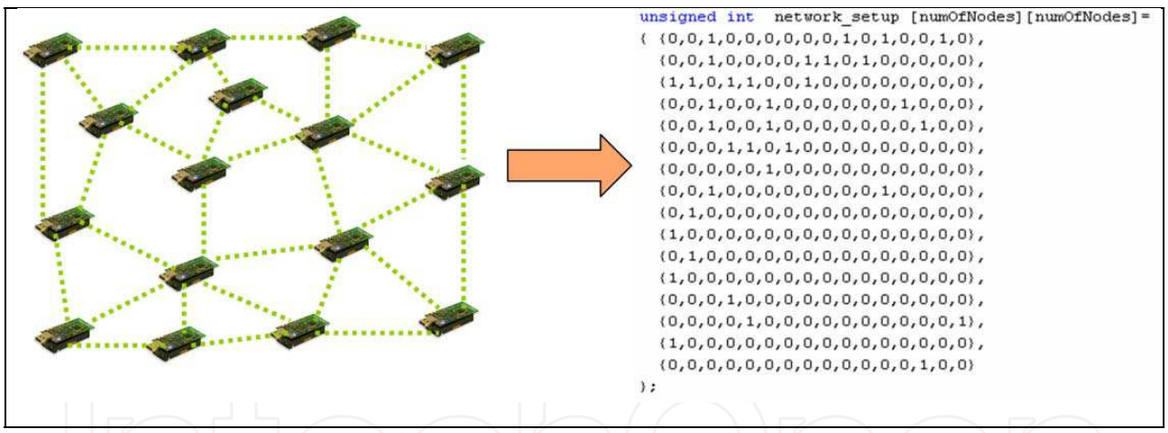
The next step will be to generate code automatically for TinyOS, MANTIS or ZigBee from the Stateflow representation of the algorithm, using a customization of Stateflow Coder (SF. 2009) which can generate ANSI C code for Stateflow blocks. In order to adapt the generated ANSI C code to the target operating system, Target Language Compiler (TLC) (RTW. 2009) scripts are used. TLC provides mechanisms by which one can generate platform specific code by taking sections (such as includes, defines, functions, etc) from ANSI C code and also by adding custom code for the target platform. In order to ease platform independent development, we provide a set of generic library functions which can be used from Stateflow to access platform specific operating system functionalities (such as *led_toggle*, *led_on*, *led_off*, *sendPacket*, *receivePacket*). From the Stateflow implementation perspective, the application developer does not have to think about the actual implementation of these generic functions in TinyOS, MANTIS or in ZigBee, since they have been implemented in the TLC library and can be targeted to any of the supported operating system and hardware nodes. By using TLC scripts (which are also called *System Target Files*), the developer now can generate automatically a TinyOS application (composed of .nc, .h and makefiles) or a MANTIS application (composed of .c, .h and makefiles) or a ZigBee end device application (also composed of .c, .h and configuration files), and then can compile and execute them for the target platform without any modification.

3. A Simple Simulation Framework

We will now demonstrate a simple sensor network model (shown in figure 3) that has been designed based on our framework components (*sensor node*, *communication medium*). In this model, we consider sixteen sensor nodes, all connected to the communication medium block to form a sensor network. At the top level, the model has two major components:-

3.1 Communication Medium Model

This block contains the medium logic and also models the connectivity between nodes. The logic of the communication medium block is implemented by a C based S-Function (MathWorks. 1984), which contains a (parameterized) 16x16 matrix to define the connectivity of the nodes in the sensor network (shown in the figure 4). In a synergistic research effort, we are also working on a library of radio channel and protocol stack modules at different levels of abstraction (bit, packet). For example in figure 4, node 1 (row 1) is connected to nodes 3, 10, 12 and 15. Packets are the inputs and outputs of the communication medium block, where incoming packets from the nodes will be at first processed by the medium logic and then fed to the appropriate nodes based on the connectivity setup of the sensor network. In this chapter, we consider a very abstract model describing a simple medium logic which at any point of time computes the input (packet) of a node as the summation of outputs (packets) of nodes connected to it.



```
unsigned int  network_setup [numOfNodes][numOfNodes]=
{ {0,0,1,0,0,0,0,0,1,0,1,0,0,1,0},
  {0,0,1,0,0,0,0,1,1,0,1,0,0,0,0},
  {1,1,0,1,1,0,0,1,0,0,0,0,0,0,0},
  {0,0,1,0,0,1,0,0,0,0,0,1,0,0,0},
  {0,0,1,0,0,1,0,0,0,0,0,0,1,0,0},
  {0,0,0,1,1,0,1,0,0,0,0,0,0,0,0},
  {0,0,0,0,0,1,0,0,0,0,0,0,0,0,0},
  {0,0,1,0,0,0,0,0,0,0,1,0,0,0,0},
  {0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
  {0,1,0,0,0,0,0,0,0,0,0,0,0,0,0},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
  {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0},
  {0,0,0,0,1,0,0,0,0,0,0,0,0,0,1},
  {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
  {0,0,0,0,0,0,0,0,0,0,0,0,1,0,0}
};
```

Fig. 4. Connectivity matrix for the 16 nodes sensor network

3.2 Node Block

This block contains sixteen nodes as shown in figure 3. The individual node model is fully parameterized and contains mainly a timer generator, a random number generator and a Stateflow application block. The timer is used for generating *CLK* events for the algorithm running inside the Stateflow block. Incoming and outgoing packets of nodes consist of *data* and *signal* information.

The *data* field contains the payload of the packet and *signal* (which triggers the Stateflow block) generates a packet arrival event which is processed by the Stateflow algorithm inside. The application developer now can perform functional analysis of the algorithm and modify it based on execution data provided by Simulink and Stateflow. In this example, we have

shown a framework of sixteen nodes but the user can easily design a network with a larger number of nodes by slightly modifying the sensor node and communication medium blocks.

4. Multi-Platform Code Generation

After functional analysis of the algorithm, the next step is to generate application code automatically for the target operating systems. For WSN application development one currently has two options, either with a research WSN operating systems solution (such as TinyOS, MANTIS, Contiki (Dunkels et al. 2004), FreeRTOS (Barry, 2003), etc.) or with the ZigBee industry standard. Most of the operating systems are built on a very lightweight event based mechanism (such as TinyOS, Contiki, etc.), however some use a more traditional thread based model (such as MANTIS). On the other hand, ZigBee only defines some layers of the WSN protocol stack and it does not cover the operating system and its libraries. Application development is done on the top of software platforms where user code calls non-standard APIs to interact with the platform. Such heterogeneity of software platforms seriously hinders application porting, and motivated us to consider look for different software platforms used in the sensor network domain and to look for possibilities to port code between them. In this context, we have chosen three software platforms that cover a broad range of programming styles. The candidates are TinyOS (event based), MANTIS (thread based) and ZigBee (an event-based industrial standard).

Our approach is to model sensor network application independent of the platform by using Stateflow, and then automatically generate platform specific application code from that abstraction. We will illustrate the flow by taking a simple WSN application and modeling it in Stateflow (as explained in the next section). For this goal, we designed generic functions that are used to bridge between the model-generated code and the underlying platform (TinyOS, MANTIS and ZigBee).

When developing an application on a platform like those described above, one must consider the services that it provides, in particular:

- the tasking and synchronization models,
- the libraries implementing frequently used functions.

If we consider the first aspect, TinyOS and ZigBee are reasonably similar, because they do not provide a preemptive task abstraction. Hence lengthy library calls cannot be implemented synchronously (by calling and waiting), but must be implemented asynchronously, by splitting them into a request and a response. This splitting allows one to write extremely efficient code, since context swapping can be implemented simply by the interrupt handling hardware. However, writing code in this style is more tedious, since it forces one to save and restore "permanent" state information by hand. MANTIS on the other hand offers a more traditional multi-tasking (to be more precise, multi-threading) environment, which is more familiar and friendly to programmers.

In order to write portable code with these different tasking models, we had to resort to a "reentrant state machine" programming paradigm, where the user code for an application is written as a single procedure, which can be called and return as part of a single "reaction" to external events. Fortunately this programming paradigm is supported by code generation tools for *synchronous reactive* models (Halbwachs, 1993), (e.g. including the StateChart model supported by Stateflow) which:

- Provide the programmer with a procedural abstraction giving the illusion of several concurrent threads of code, all running synchronously with respect to each other and hence all fully deterministic,
- Generate a reentrant state machine code that can be run in a non-preemptive environment.

This programming style is perhaps less efficient than the "native" split-phase programming mode of TinyOS and ZigBee, because it requires one to save the FSM state. But it makes control much more explicit and easy to follow than code in which the "state" is implicitly kept by the signaling between cooperating software and hardware components.
In the next section we introduce a simple WSN application that will be used as an example for porting code between WSN platforms. Sections 6, 7 and 8 describe techniques to port the application in MANTIS, TinyOS and ZigBee respectively. In these sections, we also provide a short description of the platform itself. The code writing approaches that we will explain in the following sections can also be automated by scripts in a model-based design context (for example by TLC scripts). The application code of MANTIS, TinyOS and ZigBee is very different from each others, so the script performs the following tasks to generate platform specific code:

- Copy into the target files the platform specific application independent base code including a type conversion header file that will convert all C types to platform specific types and platform specific implementations of the library functions.
- Generate platform specific application files by taking different sections (such as includes, defines, functions, etc) from the C code generated from Stateflow.
- Generate *make* or *configuration* files for each platform.

5. A Simple WSN Example

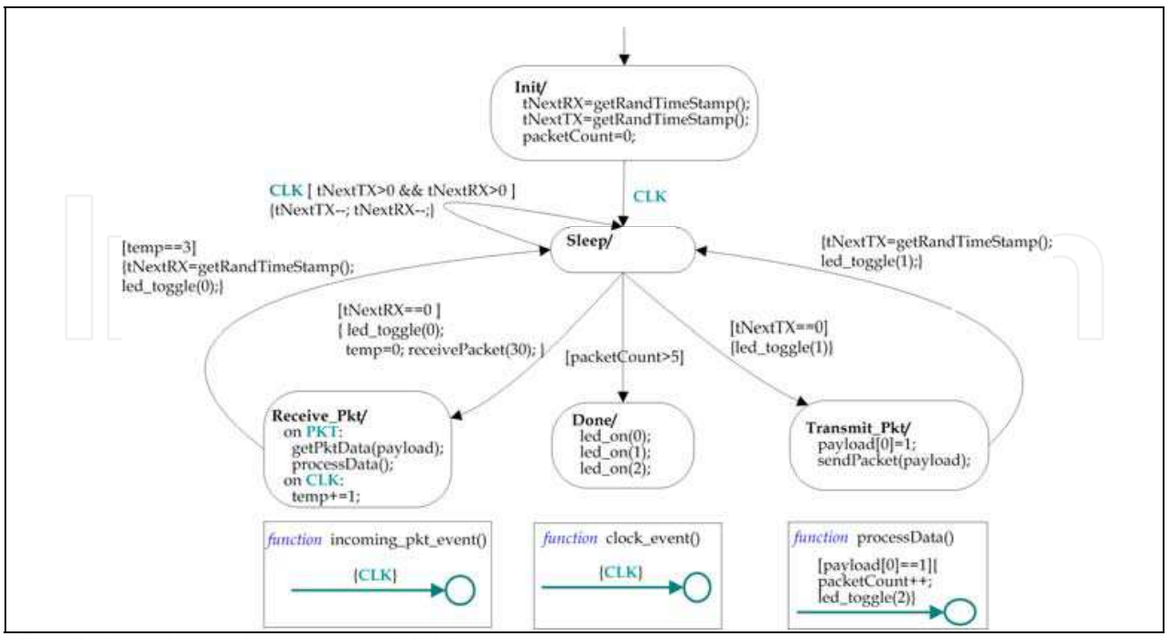


Fig. 5. Stateflow of the simple WSN application

In this section, we will describe a simple WSN application to illustrate application development in MANTIS, TinyOS and ZigBee. This simple application contains most typical ingredients of sensor network applications such as transmitting, receiving, processing of packets and sleeping. In this application example, we do not include a sensing task, but the corresponding development problems are covered by other functionalities, such as incoming events and processing data (which are included in our simple application).

The application transmits and receives packets randomly until it receives six packets, then it stops communications and turns on all LEDs of the node. We also performed more extensive application modelling in this style, but for simplicity we use this very simple application. The Stateflow model of the application is shown in Figure 5.

PKT and *CLK* are external inputs of the algorithm. The *PKT* event is generated after receiving a packet and the *CLK* event is generated when the periodic timer expires. Here, the periodic timer is set to generate a *CLK* event every 10ms. The application starts by initializing the next receiving (*tNextRX*) and transmitting (*tNextTX*) timestamps. To set these timestamps, it calls a library function *getRandTimeStamp* which returns a random number. Then it sets the number of received packets to zero. At the next *CLK* event, the application moves to the *Sleep* state from the *Init* state. In the *Sleep* state, the receiving and transmitting timestamps will be decremented by one at every occurrence of the *CLK* event. At the expiration of the transmit timestamp, the algorithm will make a transition to the *Transmit_Pkt* state and toggle led 1. In this state, it sets the first byte of the payload to 1 and sends the packet by calling library function *sendPacket*. After transmitting the packet, the application makes a transition to the Sleep state, sets the next transmission time-stamp and toggles led 1. In the same way, when the receiving time-stamp expires, the algorithm makes a transition from the Sleep state to *Receive_Pkt* state and it calls the *receivePacket* function to configure the radio in receiving mode for a specified duration (in this case 30ms). In the *Receive_Pkt* state, the algorithm waits for the *PKT* and *CLK* events. After receiving a *PKT* event, it calls library function *getPktData*, which copies the packet data field into a local variable (payload). Now the algorithm calls a local function *processData* where it checks the first byte of the packet data and if it is equal to 1, then it increases the received packet counter and toggles led 2 to give us a visual indication of successful reception of a packet. After expiration of the receiving time slot, the algorithm makes a transition to the *Sleep* state from the *Receive_Pkt* state. While making the transition, it sets the next receiving timestamp and toggles led 0.

In this manner, the algorithm makes transitions between the *Sleep*, *Transmit_Pkt*, and *Receive_Pkt* states until in the *Sleep* state it notices that the number of received packets is greater than five. Then it makes the final transition to the *Done* state where it turns on all three LEDs and stops all communications to the external world.

This simple application, just like many protocol components and WSN applications, can be conveniently modeled as a state machine, either written directly in C/C++ or generated (by Real Time Workshop) from the Stateflow model as shown in Example 1. Interactions of the state machine with the rest of the platform (HW and protocol stack) are dependent on the underlying software architecture. In this case, the incoming events are *CLK* and *PKT* and the outgoing actions are sending a packet (*sendPacket*), setting the radio in listening mode for certain amount of time (*receivePacket*) and switching the leds on the board (*led_toggle, led_on*). Handling these incoming events and outgoing actions depend on the underlying software platform while the rest of the implementation of the state machine remains mostly the same.

In the next sections, we will show how to port this C implementation of the state machine in MANTIS, TinyOS and ZigBee respectively.

6. MANTIS

MANTIS is a light-weight multi-threaded operating system that is capable of multi-tasking on energy constrained distributed sensor networks. The scheduler of MANTIS supports thread preemption which allows the operating system to switch between active threads without waiting. So the responsiveness of the operating system to critical events can be faster than in TinyOS which is non-preemptive. The scheduler of MANTIS is priority-based with round robin. The kernel ensures that all low priority threads execute after the higher priority threads. When there is no thread scheduled for execution, the system moves to sleep mode by executing the idle-thread. Kernel and APIs of MANTIS are written in standard C.

```
void state_machine(void)
{If (for_the_first_time) {
  current_state = IN_Init;                  // Storing the current state
  tNextRX = getRandNumber();        // Generic function to get random number
  tNextTX = getRandNumber();
  packetCount = 0;
}else{
  switch(current_state) {
  case IN_Init:
    if(incoming_event== event_CLK)  // Handling CLK event
      current_state =  IN_Sleep; break;
  case IN_Sleep:
    if((incoming_event== event_CLK) && ((tNextTX > 0) && (tNextRX > 0))){
      tNextTX--;tNextRX--;  current_state = IN_Sleep;
    }else if (tNextRX == 0) {
      led_toggle(0); temp=0;                  // Generic function to toggle led
      receivePacket(30);                      // Generic function to receive packets
      current_state =  IN_Receive_pkt;
    }else if(packetCount > 5) {
      current_state = IN_done; led_on(0);led_on(1);led_on(2);
    }else if (tNextTX == 0)){
      led_toggle(1); current_state = IN_Transmit_pkt;  payload[0] = 1;
      sendPacket(payload);              // Generic function to send packet
      }
    break;
  case IN_Receive_pkt:
    if(temp == 3){
      tNextRX =  getRandNumber();  led_toggle(0); current_state =  IN_Sleep;
    }else {
      if(incoming_event== event_PKT) {  // Handling PKT event
        getPktData(payload);          // Generic function to get packet content
        process_data();}
```

```
    if(incoming_event== event_CLK)    // Handling CLK event
      temp++;
   }
   break;
 case IN_Transmit_pkt:
   tNextTX =  getRandNumber(); led_toggle(1); current_state=  IN_Sleep;    break;
 case IN_done:
   break;
 default:
   current_state =  IN_NO_ACTIVE_CHILD; break;
 }
}
}
```

Example 1. C code generated by RTW for the state machine of figure 1

6.1 Application porting in MANTIS

MANTIS provides a convenient environment to develop WSN applications. All applications begin with a *start* which is similar to *main* in C programming. One can spawn new threads by calling *mos_thread_new*. MANTIS supports a comprehensive set of APIs for sensor network application development (MOS. 2003), most frequently used APIs are listed below for simple application development based on categories.

- Scheduler : *mos_thread_new*, *mos_thread_sleep*
- Networking : *com_send*, *com_recv*, *com_recv_timed*, *com_ioct*, *com_mode*
- Visual Feedback (Leds) : *mos_led_on*, *mos_led_off*, *mos_led_toggle*
- On board sensors (ADC) : *dev_write*, *dev_read*

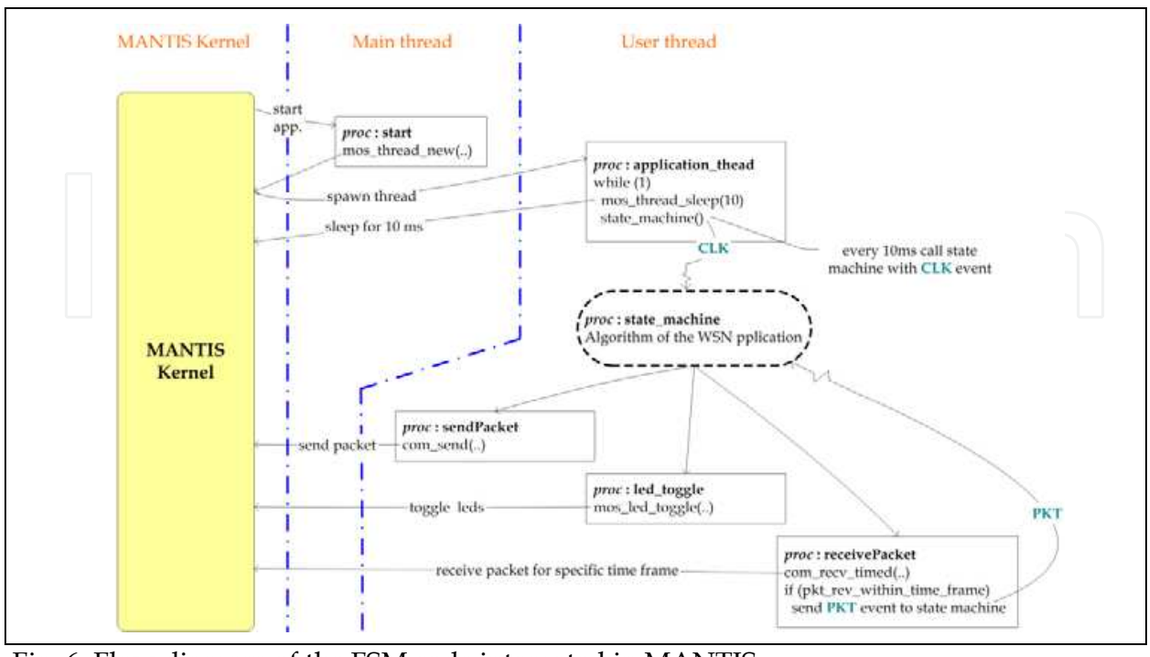


Fig. 6. Flow diagram of the FSM code integrated in MANTIS.

We can port easily the automatically generated code of the state machine in MANTIS. For this, a new thread is spawned from the *start* procedure. In the newly created thread, the state machine is called in every 10 milliseconds, as required in the algorithm. Here the *CLK* is virtually implemented by calling *mos_thread_sleep(10)* . Figure 6 shows the skeleton of the simple application implementation in MANTIS. For receiving packets, the user can use *com_recv* which waits until a successful reception of a packet by blocking the thread. But for implementing our simple application, the program needs to be in the receiving state for certain amount of time. This can be done by another API which is *com_recv_timed*. It turns on the radio in receiving mode for a certain amount of time. When it receives a packet, it calls the state machine with the incoming packet event (*PKT* event of the state machine). Implementation of other outgoing actions such as to sending a packet and switching the leds is also easy, by calling *com_send*, *mos_led_toggle* and *led_on* APIs.

7. TinyOS

The programming model of TinyOS is based on components. In TinyOS, a conceptual entity is represented by two types of components, *Module* and *Configuration*. A component implements interfaces. The interface declares signature of the *commands* and *events* which must be implemented by the provider and user of the interface respectively. Events are the software abstractions of hardware events such as reception of packet, completion of sensor sampling etc. On the other hand, commands are used to trigger an operation such as to start sensor reading or to start the radio for receiving or transmitting etc. TinyOS uses a split-phase mechanism, meaning that when a component calls a command, it returns immediately, and the other component issues a callback event when it completes. This approach is called split-phase because it splits invocation and completion into two separate phases of execution. The scheduler of TinyOS is based on an event-driven paradigm where events have the highest priority, run to completion (i.e. interrupts cannot be nested) and can preempt and schedule *tasks*. Tasks contain the main computation of an application. TinyOS applications are written in nesC which is an extension of the C language.

7.1 Application porting in TinyOS

In TinyOS, application coding uses several interfaces. The skeleton of the simple application implementation is shown in figure 7. Module *simpleAppM* uses interfaces *Boot*, *Timer* and others. When an application module uses an interface then it can issue the commands provided by that interface and it should also implement all the events that could be generated from the interface. For example, the *Boot.booted* event of the *Boot* interface is implemented in the module *simpleAppM*. Among the several interfaces available in the library of TinyOS, we listed those most frequently used for constructing simple applications.

- Initialization: *Init*, *Boot*, *Timer*
- Networking: *Send*, *Receive*, *AMSend*, *SplitControl*, *Packet*, *AMPacket*
- Visual Feedback (Leds): *Leds*

Details of the TinyOS operating system can be found in (TOS. 2000). To implement the simple application, at first a periodic timer (*CLKtimer.startPeriodic*) is initialized from the *Boot.booted* event handler. The period of the timer is set to 10 milliseconds as required in the algorithm. After initialization has been done, a timer event is generated (*CLKtimer.fired*).

Inside this event handler, the state machine is called as a *task* (implementing the *CLK* event of the state machine). The algorithm needs to be in receiving mode for specific amount of time (30 milliseconds). Hence in the *receivePacket* method, we set a one shot timer (for 30 milliseconds) and at the same time start the radio. After expiration of this timer the radio needs to be stopped (done in the event handler of *RXwindowTimer.fired*). When TinyOS receives a packet it generates an event (*Receive.receive*). Inside this event we post the task of the state machine with the incoming packet event (implementing the *PKT* event of the state machine). We used the *LowPowerListening* interface to control the radio explicitly in receiving or transmitting mode. For handling outgoing actions from the state machine, such as to send a packet, the state machine calls the *sendPacket* method. Inside this method, we at first set the radio in transmit mode and then start it. When the radio is started (it generates *Radio.StartDone* event), the method checks whether the radio is turned on for sending a packet or not. If so, we use the *AMSend.send* command of the *AMSend* interface to send the packet.

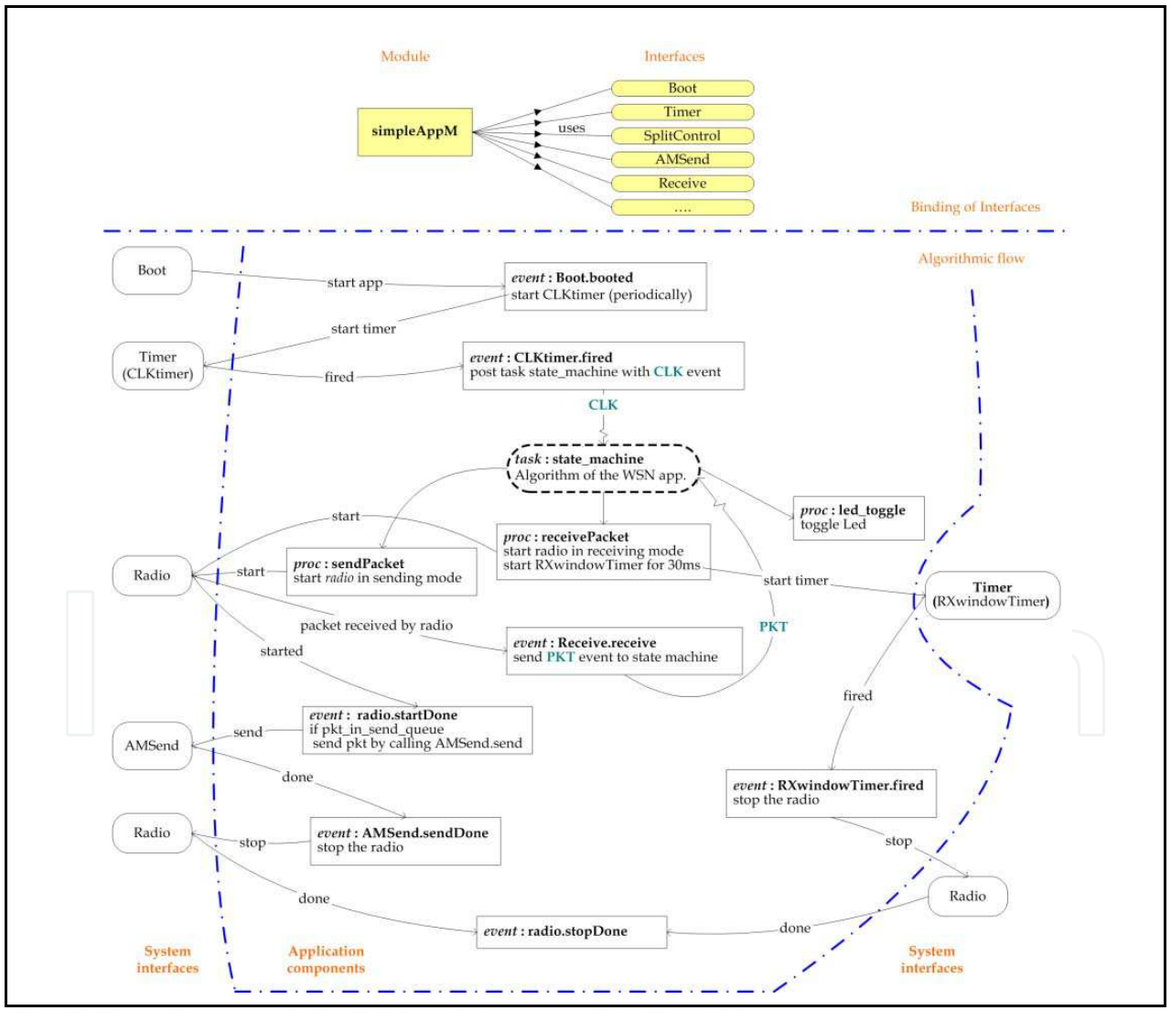


Fig. 7. Flow diagram of the FSM code integrated in TinyOS

When the packet is sent then TinyOS generates a call back event *AMSend.sendDone* which provides the status of the sending processing. Inside this event handler, we stop the radio. There are some commands in TinyOS which are qualified as *async* and do not generate callback events. We used *async* commands for switching the leds from the state machine.

8. ZigBee

ZigBee is a specification that enables reliable, cost effective, low power, wireless networked, monitoring and control products based on an open global standard. ZigBee is targeted at the WSN domain because it supports low data rate, long battery life and secure networking. At the physical and MAC layers, ZigBee adopted the IEEE 802.15.4 standard. It includes mechanisms for forming and joining a network, a CSMA mechanism for devices to listen for a clear channel, as well as retries and acknowledgment of messages for reliable communication between adjacent devices. These underlying mechanisms are used by the ZigBee network layer to provide reliable end to end communications in the network. The 802.15.4 standard is available from (IEEE. 2003).

At the network layer, ZigBee supports different kinds of network topologies such as *Star*, *Tree* and *Mesh*. The ZigBee specification supports networks with one *coordinator*, multiple *routers*, and multiple *end devices* within a single network. A ZigBee coordinator is responsible for forming the network. Router devices provide routing services to network devices, and can also serve as end devices. End devices communicate only with their parent nodes and, unlike router devices, cannot relay messages intended for other nodes. Details of the ZigBee specification can be found at (ZigBee. 2006).

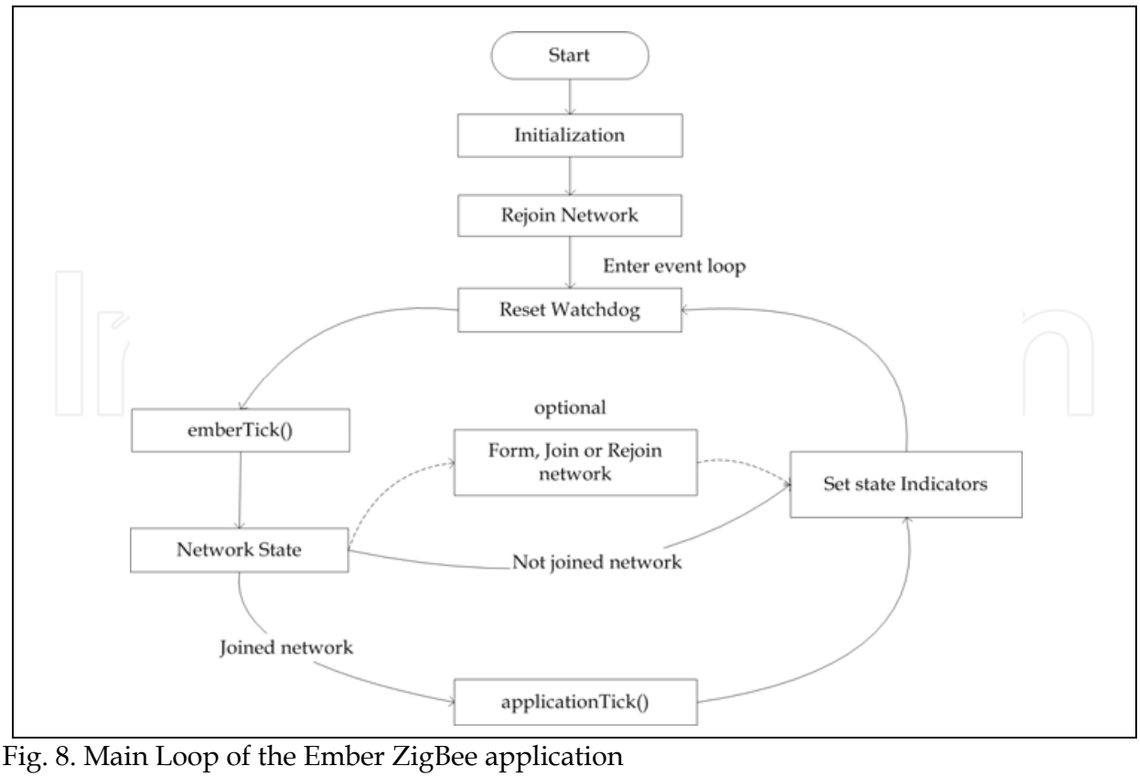


Fig. 8. Main Loop of the Ember ZigBee application

8.1 Application porting in ZigBee

Several implementations of the ZigBee stack are available on the market (such as from Texas Instruments, Ember Corporation, Freescale etc). We will describe our simple application implementation by using the Ember implementation (EMBER. 2008). The main source file of a ZigBee application must begin by defining some parameters involving *endpoints*, *callbacks* and *global variables*. Endpoints are required to send and receive messages, so any device (except a basic network relay device) will need at least one of these. Just like C, an application starts from *main*. The initialization and event loop phases (shown in figure 8) of a ZigBee application are shortly described below.

Among the initialization tasks, serial ports (SPI, UART, debug or virtual) need to be initialized. It is also important to call *emberInit()* which initializes the radio and the ZigBee stack. Prior to calling *emberInit()*, it needs to initialize the Hardware Abstraction Layer (HAL) and also to turn on interrupts. After calling *emberInit()*, the device rejoins the network if previously it had been connected, sets the security key, initializes the application state and also sets any status or state indicators to the initial state.

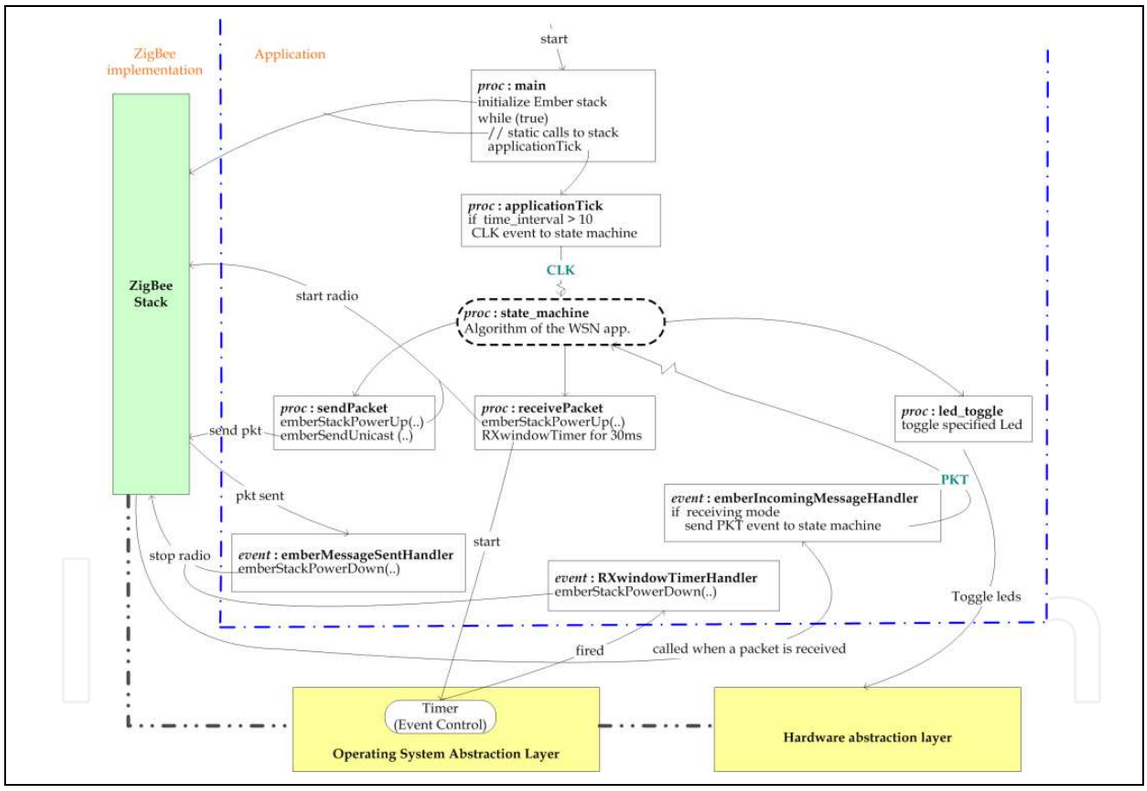


Fig. 9. Flow diagram of the FSM code integrated in ZigBee

The network state is checked once during each cycle of the event loop. If the state indicates joined (in case of router and end device) or formed (for the coordinator) network, then the *applicationTick* function is executed. Inside this function the developer will put the application code. If the network is not joined or formed, then the node will try to join or form the network. State indicators are simply LEDs but could be an alphanumeric display or some other state indicator. The function *emberTick* is a periodic tick routine that should be

called in the application's main event loop after *emberInit*. The watchdog timer should also be reset once per event loop by calling *halResetWatchdog*.
The skeleton of the simple application implementation in ZigBee is shown in figure 9. Here, the state machine is called from *applicationTick*. The state machine is called at 10 millisecond intervals, which implements the *CLK* of the state machine. When the *receivePacket* method is called from the state machine, we start the radio by calling the *emberStackPowerUp* API and then schedule an event (*RXwindowTimer*) which will generate a callback event after expiration of receiving timer (30ms). When this callback event (*RXwindowTimerHandler*) occurs, we stop the radio. In this time frame, if a packet is received by the ZigBee stack, it calls an incoming message handler function *emberIncomingMessageHandler*. Inside this function, the state machine is called with the incoming packet event (*PKT* event of the state machine). When the *sendPacket* method is called from the state machine, again we start the radio and send the packet by calling the *emberSendUnicast* API which afterward calls back the *emberMessageSentHandler* function. Inside this event handler, we stop the radio. Implementations of *led_toggle* and *led_on* methods are simple like in MANTIS and TinyOS.

9. Conclusion

We described an extensible framework for modeling, simulation and multi-platform code generation of sensor network algorithms based on MathWorks tools. We developed parameterized blocks for the *sensor node* and *communication medium* to ease the modeling and simulation of WSN applications. Portability of application between multiple platforms is an open problem, especially in the WSN domain because of the lack of a single platform standard. We presented application porting in MANTIS, TinyOS and ZigBee using a simple application. We identified a single code writing style, namely state machine-like, that can be ported easily across different platforms by just creating an API abstraction layer for sensors, actuators and non-blocking OS calls. This FSM-like code can be written by or generated from different StateChart-like or Synchronous Language models, which also makes the generation of the adaptation layer to each platform easier. The reason for choosing the MathWorks tools over, for example, TOSSIM, NS, OmNet, is that they are well known and already provide rich libraries for digital signal processing and control algorithm behavior simulation.

10. References

Abdelzaher, T. ; Blum, B. ; Cao, Q. ; Evans, D.; George, J.; George, S. ; He, T. ; Luo, L. ; Son, S. ; Stoleru, R.; Stankovic, J. & Wood, A. (2004). Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. *In the Proceedings of the IEEE International Conference on Distributed Computing Systems*, Tokyo, Japan

Almeida, V.; Vieira, L.; Vitorino, B.; Vieira, M. ; Fernandes, A.; Silva, D. & Coelho, C. (2003) Microkernel for Nodes of Wireless Sensor Networks, *In the poster session of the 3rd Student Forum SBCCI*, Chip in Sampa, Brasil.

Barry, R. 2003. FreeRTOS, A FREE open source RTOS for small embedded real time systems. http://www.freertos.org/PC/.

Bakshi, A.; Prasanna, V. K.; Reich, J. & Larner, D. (2005). The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. *In the Proceedings of End-to-end, sense-and-respond systems, applications and services*, pages 19–24.

Bhatti, S.; Carlson, J.; Dai, H.; Deng, J. ; Rose, J. ; Sheth, A. ; Shucker, B. ; Gruenwald, C. ; Torgerson. A. & Han., R. (2005). MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *In the journal of MONET*, pages 563-579

Cheong, E.; Lee, E. & Zhao, Y. (2005). Viptos: A graphical development and simulation environment for TinyOS-based wireless sensor networks. *In the Proceedings of 3rd International Conference on Embedded Networked Sensor Systems*, *SenSys*, page 302

Dunkels, A.; Gronvall, B. & Voigt, T.(2004). Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors, Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, ISBN 0-7695-2260-2, pages 455--462, USA

Eker, J.; Janneck, J.; Lee, E. A. ; Liu, J. ; Liu, X. ; Ludvig, J. ; Sachs, S. & Xiong Y. (2003) Taming heterogeneity - the Ptolemy approach, *Proceedings of the IEEE*, volume 99(1), pages: 127-144

EMBER. (2001). Zigbee Wireless Semiconductor Solutions by Ember. www.ember.com.

Gay, D.; Levis, P.; Behren, J. R.; Welsh, M.; Brewer, E. A. & Culler, D. E. (2003) The nesC language: A holistic approach to networked embedded systems. *In the Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 1-11,

Gummadi, R. ; Gnawali, O. & Govindan, R. (2005). Macro-programming wireless sensor networks using kairos. *In the Proceedings of the 1st International Conference on Distributed Computing on Sensor Systems*, pages 126-140

Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers

Levis, P.; Lee, N.; Welsh, M. & Culler, D.E. (2003) TOSSIM: accurate and scalable simulation of entire tinyOS applications. *In the Proceedings of 1st International Conference on Embedded Networked Sensor Systems*, SenSys, pages 126-137

Levis, P.; Madden, S.; Gay, D.; Polastre, J.; Szewczyk, R.; Woo, A.; Brewer, E. A. & Culler, D. E. (2004) The Emergence of Networking Abstractions and Techniques in TinyOS. *In the Proceedings of 1st Symposium on Networked Systems Design and Implementation*, NSDI, pages 1-14, 2004

Necchi, L. ; Bonivento, A. ; Lavagno, L. ; Vanzago, L. & Sangiovanni-Vincentelli, A. (2007) EERINA: an Energy Efficient and Reliable In-Network Aggregation for Clustered Wireless Sensor Networks. *In the Proceedings of Wireless Communications and Networking Conference*, *WCNC*, pages 3364-3369

Newton, R. & Welsh, M. (2004). Region streams: functional macroprogramming for sensor networks. *In the Proceedings of the 1st International Workshop on Data Management for Sensor Networks*, pages 78-87

MathWorks. (1984). MATLAB and Simulink for Technical Computing. www.mathworks.com/

MOS. (2003). MANTIS, MultimodAl NeTwork of In-situ Sensors. http://mantis.cs.colorado.edu/index.php/tiki-index.php

Mozumdar, M.M.R. ; Gregoretti, F. ; Lavagno, L.; Vanzago, L. & Olivieri, S. (2008a). A framework for modeling, simulation and automatic code generation of sensor network application, In the Proceedings of 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, pages 515--522.

Mozumdar, M.M.R. ; Gregoretti, F. ; Lavagno, L. & Vanzago, L. (2008b). Porting application between wireless sensor network software platforms: TinyOS, MANTIS and ZigBee, In the Proceedings of IEEE International Conference on Emerging Technologies and Factory Automation, pages 1145-1148.

NS-2. 2001. The Network Simulator. 2001. http://www.isi.edu/nsnam/ns

OMNeT. (1992). Community Site. http://www.omnetpp.org/

Vieira, L. F. M. ; Vitorino, B. A. D.; Vieira, M. A. M.; Silva, D. C. & Loureiro, A. O. (2005). WISDOM: A Visual Development Framework for Multi-platform Wireless Sensor Networks. *In the Proceedings of 10th IEEE International Conference on Emerging Technologies and Factory Automation*, *ETFA*, Catania, Italy.

RTW. (2009). Real-Time Workshop - Generate C code from Simulink models and MATLAB code. http://www.mathworks.com/products/rtw/

SF. (2009). Stateflow-Design and simulate state machines and control logic. http://www.mathworks.com/products/stateflow/

TOS. (2000). TinyOS Community Forum, An open-source OS for the networked sensor regime. http://www.tinyos.net/

ZigBee. (2006). ZigBee Alliance. http://www.zigbee.org/.

IntechOpen

Factory Automation

Edited by Javier Silvestre-Blanes

ISBN 978-953-307-024-7
Hard cover, 602 pages
Publisher InTech
Published online 01, March, 2010
Published in print edition March, 2010

Factory automation has evolved significantly in the last few decades, and is today a complex, interdisciplinary, scientific area. In this book a selection of papers on topics related to factory automation is presented, covering a broad spectrum, so that the reader may become familiar with the various fields, and also study them in more depth where required. Within various chapters in this book, special attention is given to distributed applications and their use of networks, since it is one of the most relevant subjects in the evolution of factory automation. Different Medium Access Control and networks are analyzed, while Ethernet and Wireless networks are looked at in more detail, since they are among the hottest topics in recent research. Another important subject is everything concerning the increase in the complexity of factory automation, and the need for flexibility and interoperability. Finally the use of multi-agent systems, advanced control, formal methods, or the application in this field of RFID, are additional examples of the ideas and disciplines that experts around the world have analyzed in their work.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Mohammad Mostafizur Rahman Mozumdar and Luciano Lavagno (2010). Rapid Application Development for Wireless Sensor Networks, Factory Automation, Javier Silvestre-Blanes (Ed.), ISBN: 978-953-307-024-7, InTech, Available from: http://www.intechopen.com/books/factory-automation/rapid-application-development-for-wireless-sensor-networks

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2010 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License, which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.

IntechOpen

IntechOpen