A comparison of software platforms for Wireless Sensor Networks: MANTIS, TinyOS and ZigBee

A comparison of software platforms for Wireless Sensor Networks: MANTIS, TinyOS and ZigBee

Mohammad Mostafizur Rahman Mozumdar, Luciano Lavagno
Politecnico Di Torino
and
Laura Vanzago
STMicroelectronics

Wireless sensor networks are characterized by very tight code size and power constraints, and by a lack of well-established standard software development platforms such as Posix. In this paper, we present a comparative study between a few fairly different such platforms, namely MANTIS, TinyOS and ZigBee, when considering them from the application developer's perspective, i.e. by focusing mostly on functional aspects, rather than on performance or code size. In other words, we compare both the tasking model used by these platforms and the API libraries they offer. Sensor network applications are basically event based, so most of the software platforms are also built on considering event handling mechanism, however some use a more traditional thread based model. In this paper, we consider implementations of a simple generic application in MANTIS, TinyOS and the Ember ZigBee development framework, with the goal of depicting major differences between these platforms, and suggesting a programming style aimed at maximizing portability between them.

Categories and Subject Descriptors: D.4.7 [**Operating systems**]: Organization and Design

General Terms: TinyOS, MANTIS, ZigBee

Additional Key Words and Phrases: Wireless sensor networks, Software platform, Application porting

1. INTRODUCTION

Recent advancements in microelectro-mechanical systems and wireless communications motivated the development of small and low power sensors and radio equipped modules that are replacing traditional wired sensor systems. These modules can communicate with each other by radio to receive and transmit data and form a wireless sensor network (WSN). A WSN application developer currently has two broad options: freely available academic operating systems (such as TinyOS[Levis et al. 2004], MANTIS[Bhatti et al. 2005], Contiki [Dunkels et al. 2004], FreeRTOS[Barry

This research was supported by STMicroelectronics, Agrate, Milan, Italy.
Authors' addresses: Mohammad Mostafizur Rahman Mozumdar and Luciano Lavagno, Department of Electronics, Politecnico di Torino, Corso Duca degli Abruzzi 24, 10129 Torino, Italy, email: {mohammad.mozumdar,lavagno}@polito.it. Laura Vanzago is currently working with STMicroelectronics, Agrate, Milan, Italy, email: laura.vanzago@st.com.

2003], etc), or commercial implementations of the ZigBee standard [ZigBee 2002].

However, ZigBee only defines some layers of the protocol stack, but it does not cover the more traditional OS level, including for example how to model concurrency and synchronization. Hence ZigBee implementations actually use a non-standard OS layer. Ember [Ember 2001] provides one which has been used in this paper, while TI [Texas 1996], Freescale[BeeKit 2004] and Jennic [BOS 2006] provide other implementations, with slightly different tasking models. However, we claim that the three kinds of platforms considered in this paper are representative of the available alternatives.

In this paper, we outline the main features of three platforms, namely MANTIS, TinyOS and the Ember ZigBee implementation, from the functional point of view, i.e. by considering aspects such as breaking up an application into tasks, threads, event handlers and so on. We do not look at issues such as performance or power consumption, since they would require a very different methodology and could not use a simple application as a paradigmatic example. In addition, we make suggestions on the coding style, aimed at improving the code portability among those platforms, by adopting an FSM style to develop the user application.

When developing an application on a platform such as 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 the ZigBee implementation by Ember are reasonably similar, because they do not provide a preemptive task abstraction. Hence lengthy library calls cannot be implemented synchronously (by calling and blocking), 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, but heavier to implement in terms of both code size, execution time and interrupt response time.

On the library side, ZigBee implementations clearly provide the richest and most powerful set of network management and utility functions, followed by TinyOS (which has both the advantages and the problems of an open-source programming environment), while MANTIS offers the smallest set of reusable functionality.

The contribution of this paper is two-fold:

(1) on the tutorial side, we identified key coding style differences that make porting of an application to a different WSN platform more difficult than needed, and we illustrate by means of code and graphics those differences.
(2) on the design method side, we identified a coding style that is the "least common denominator" between the requirements of these WSN platforms, and which enables easy porting.

In order to write portable code with different tasking models, we had to resort to a "reentrant" Finite State Machine-like programming paradigm, where the user

code for a given piece of functionality is written as a single `switch`-based function, which can be called and return as part of each "reaction" handling and/or posting events, without ever blocking waiting for a response which may take time.

In terms of ease of writing, understanding and maintaining, this coding style is in the middle between traditional procedural code with blocking function calls, and the split-phase callback-based code required by both TinyOS and Ember, which forces the programmer to distribute a single piece of functionality over several small handlers for individual events.

Moreover, this programming paradigm is supported by code generation tools for *synchronous reactive* models [Halbwachs 1993], 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.

Initially, sensor network research concentrated on the development of customized protocols targeted to specific hardware or application domains. Recently, researchers realize that the protocol-centric methodology does not suffice with rapid application development demand in sensor networks [Bakshi et al. 2002], [Mannion et al. 2005], [Bakshi and Prasanna 2004]. Higher levels of abstraction in WSN application modeling are proposed in [Abdelzaher et al. 2004] ,[Gummadi et al. 2005],[Newton and Welsh 2004], [Bakshi et al. 2005]. Most of these approaches use functional or macro programming and introduce new programming languages to model the application at higher levels, while we advocate either to use a specific programming style in standard C or C++, or to use an existing well-known graphical language (Stateflow, but the same result can be obtained from any synchronous reactive language [Halbwachs 1993]) to generate code following that style. 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 paper, we identified a single programming style that is compatible with most kinds of WSN software platforms (e.g. MANTIS, TinyOS and Zigbee). This idea has also been demonstrated in Mozumdar et al. [2008; 2008], by using Stateflow [SF 2008] for application modeling and Real Time Workshop [RTW 2008] for multi-platform code generation targeting MANTIS and TinyOS.

Data-centric approaches such as TinyDB[Madden et al. 2005], Cougar[Fung et al. 2002] and SINA[Jaikaeo et al. 2000] also use domain-specific languages created for databases to describe some specific classes of WSN applications at a higher abstraction level. All these systems treat the sensor network as a distributed database and data is requested from the network by formulating abstract queries. These requests are then translated into distributed queries with while optimizing cost/performance goals such as minimization of energy consumption or communication. Since these approaches are mostly query based, they do not provide sufficient flexibility to model all sorts of applications needed in the WSN domain.

In the following section we describe a simple WSN application and show its implementation by using Stateflow and ANSI C. Section 3, 4 and 5 describe how

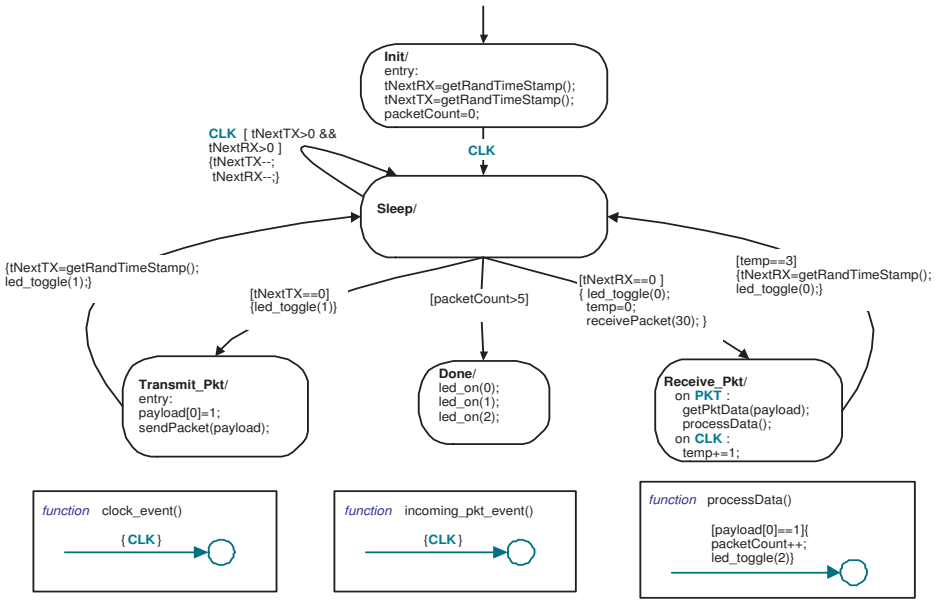

Fig. 1. Stateflow of the Simple Application

the simple application can be ported to MANTIS, TinyOS and ZigBee respectively, both by using the FSM-style code and by natively writing it for the specific platform. Finally in section 6, we summarize our contributions.

2. SIMPLE APPLICATION

In this section we will describe a simple WSN application to illustrate the 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 also 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, 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. The Stateflow modeling of the application is shown in figure 1[1].

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 10 milliseconds. The application starts by initializing the next receiving (*tNextRX*) and transmitting (*tNextTX*) timestamps. To set these timestamps, it calls a library function *getRandNumber* which returns a random number. Then it sets the number of received packets to zero. At the next *CLK* event, the application moves

[1]As we mentioned above, we use StateFlow only for convenience. The same programming style can be used when coding by hand.

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 time-stamp, 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 30 milliseconds). 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 time-stamp and toggles led 0.

Example 1: C code generated by Real Time Workshop for the state machine of figure 1

```
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;
  }
 }
}
```

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.

In the following sections, we will address how to integrate this C code in MANTIS, TinyOS and ZigBee. We will show that the FSM programming style, albeit tedious to use by hand, provides a convenient least common denominator that can be easily ported to the different programming models of the various platforms. We will also show native implementations of the same application using the native programming and tasking paradigm of each platform, in order to give a better idea of how each platform can be programmed more efficiently but less portably. The programming languages of MANTIS, TinyOS and the Ember based development framework are either C or extensions of C. In the state machine, the incoming events are *CLK* and *PKT* and outgoing actions are sending packet (*sendPacket*), setting the radio for receiving 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 depends on the underlying software and hardware platforms, and in our approach are handled by a simple platform-independent API layer, while rest of the implementation of the code remains the same.

3. MANTIS

MANTIS is a light-weight operating system that is capable of multi-threading on energy constrained distributed sensor networks. The scheduler of MANTIS supports thread preemption, hence the responsiveness 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. The kernel and APIs of MANTIS are written in standard C.

3.1 Application Development 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 1.0 supports a comprehensive set of APIs for sensor network application development. The most frequently used ones are listed below.

—Scheduler : *mos_thread_new, mos_thread_sleep*

—Networking : *com_send, com_recv, com_recv_timed, com_ioct, com_mode*

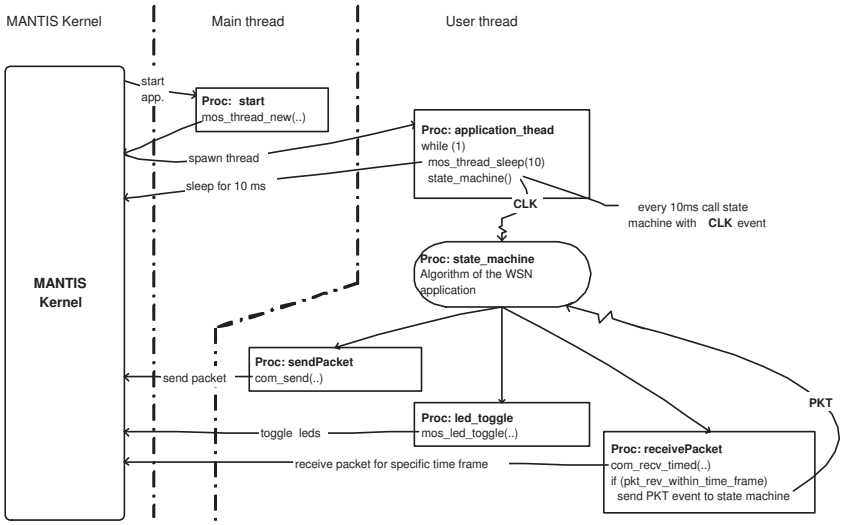

Fig. 2. Flow diagram of the FSM code integrated in MANTIS

—Visual Feedback (Leds) : *mos_led_on, mos_led_off, mos_led_toggle*

—On board sensors (e.g. ADC) : *dev_write, dev_read*

Figure 2 illustrates the interactions between the various threads that implement the simple application and the services it uses in MANTIS (the actual code skeleton is shown in Appendix A). A new thread is spawned from the *start* procedure and inside this thread, the state machine is called every 10 milliseconds, as required in the algorithm. Here the *CLK* is implemented by calling *mos_thread_sleep(10)*. For receiving packets, one could use *com_recv* which waits until a successful reception of a packet by blocking the thread. However our application needs to be in the receiving state only for a limited amount of time. This can be done by calling *com_recv_timed*, which turns on the radio in receiving mode for a given 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 sending a packet and switching the LEDs is also easy, by calling *com_send*, *mos_led_toggle* and *led_on* APIs.

If we manually implement the same application, the flow diagram will be similar to what is shown in figure 3 (the actual code skeleton is shown in Appendix B). The application thread computes the next transmit and receive times at random, then it goes to sleep for the smallest among them. After sending a packet, it sets the next transmit time and after receiving a packet (listening for 30 milliseconds), it sets the next receive time.

4. 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 the signature of the

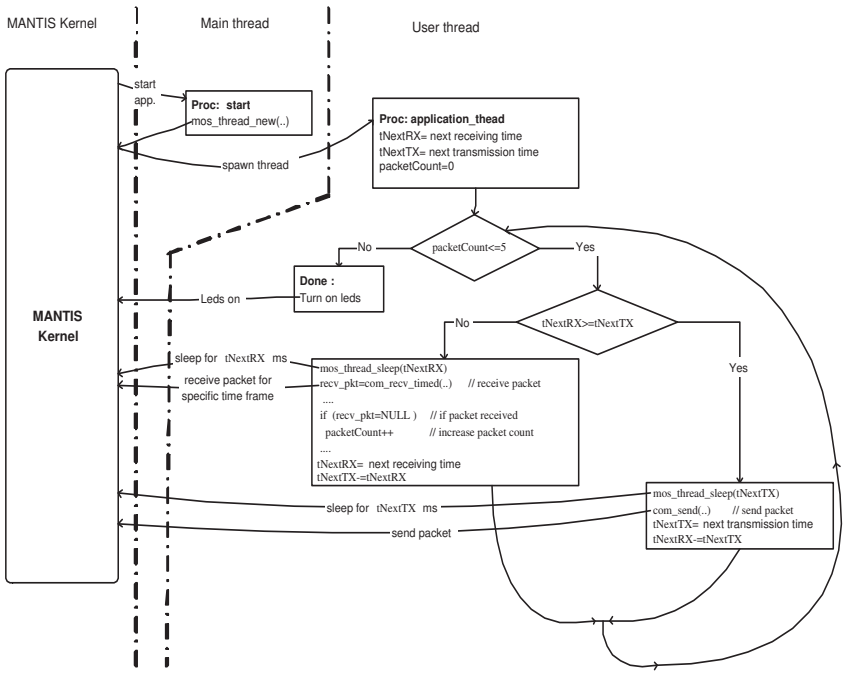

Fig. 3. Flow diagram of the non-portable code integrated MANTIS

commands and *events* which must be implemented by the provider and user of the interface respectively. Events are the software abstraction 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 client component calls a command of a server component, this returns immediately, and the server issues a callback event to the client 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.

4.1 Application Development in TinyOS

In TinyOS, application coding uses several interfaces. The flow diagram of the FSM-like code is shown in figure 4 (the actual code skeleton is shown in Appendix C). Module *simpleAppM* uses interfaces *Boot*, *Timer*. 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 2.0,

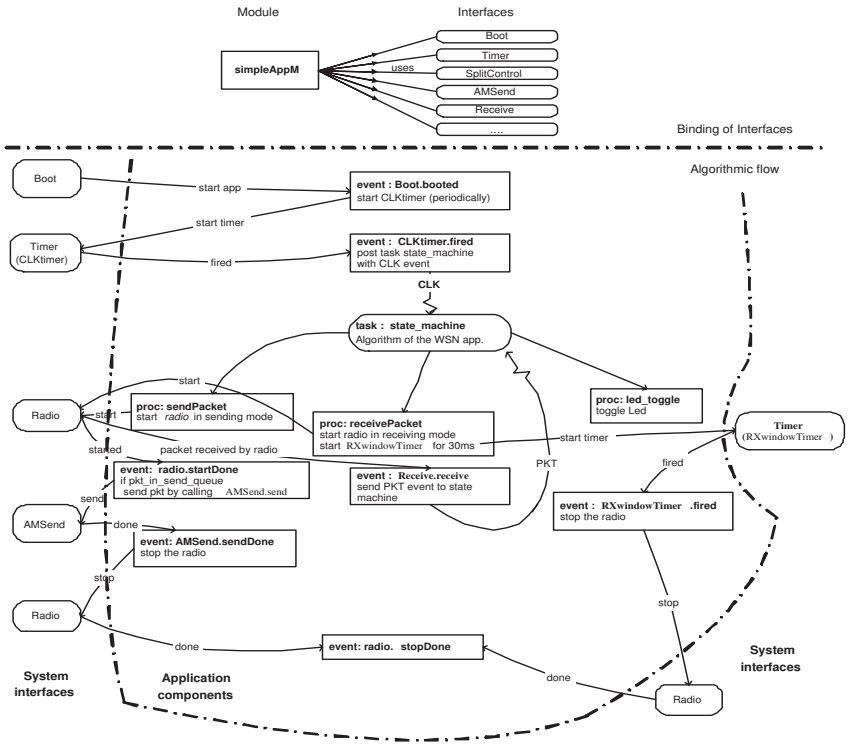

Fig. 4. Flow diagram of the FSM code integrated in TinyOS

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 at [Levis et al. 2004]. To implement the simple application, 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 (which is *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 a given amount of time (30 milliseconds). Hence in the *receivePacket* method we need to 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 (in the event handler *RXwindowTimer.fired*). When TinyOS receives a packet it generates an event (*Receive.receive*). Inside its event handler, 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 packet, the

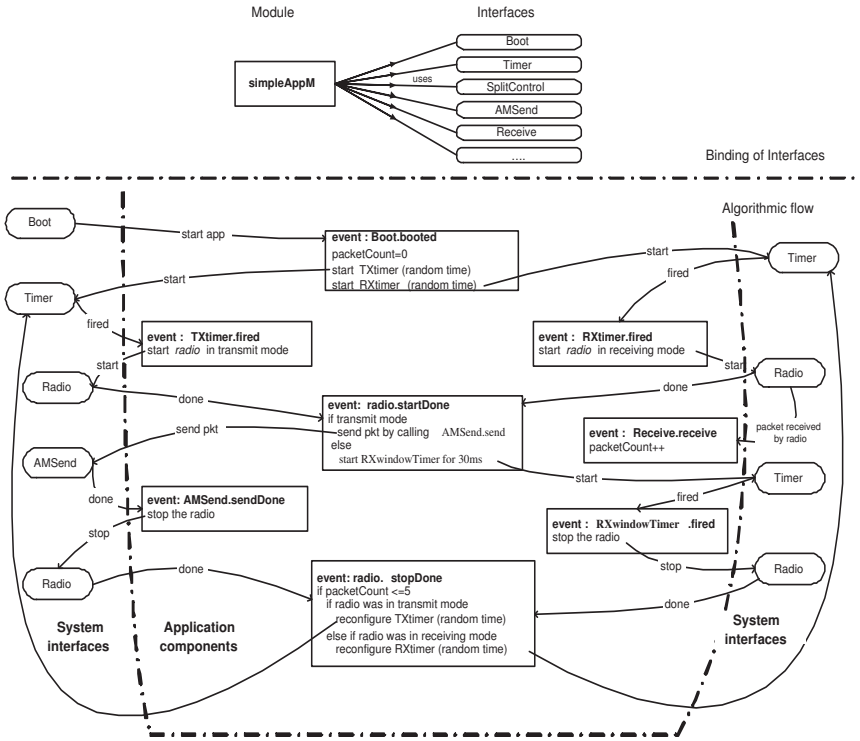

Fig. 5. Flow diagram of the non-portable code integrated in TinyOS

state machine calls the *sendPacket* method. Inside this method, at first we set the radio in transmit mode and then start it. When the radio is started (it generates the *Radio.StartDone* event), the method checks whether the radio is turned on for sending packet or not. If so, we use the *AMSend.send* command of the *AMSend* interface to send the packet. 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 called *async* and do not generate callback events. We used *async* commands for switching the LEDs from the state machine.

The flow diagram of the manual implementation of the same application in TinyOS is shown in figure 5 (the actual code skeleton is shown in Appendix D). The split phase mechanism of TinyOS results in significant differences with the MANTIS-specific code. In the *Boot.booted* event handler, two timers are scheduled for transmit and receive (having random values for both timers). After expiration of each timer, the application starts the radio and sets the radio in a specific mode (receive or transmit). After sending the packet, it stops the radio and when the radio is stopped it sets the timer for the next transmission. In the same way, after receiving for 30 ms, it sets the timer for next reception.

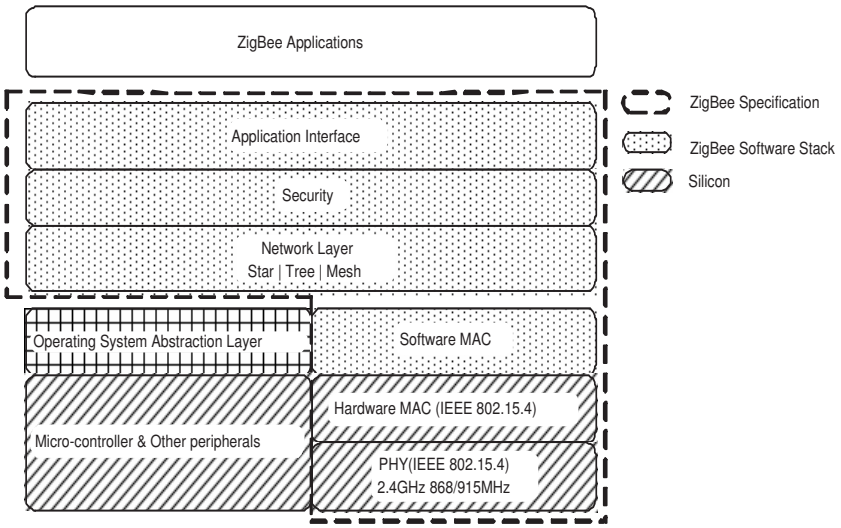

Fig. 6. ZigBee architecture

5. ZIGBEE

ZigBee is a specification that enables reliable, cost effective, low power, wirelessly 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*. A ZigBee coordinator is responsible for forming the network. Router devices provide routing services, and can also serve as end devices. End devices communicate only with their parent router nodes and cannot relay messages intended for other nodes. Details of the ZigBee specification can be found at [ZigBee 2002].

The whole ZigBee architecture is shown in figure 6. The operating system is not part of the ZigBee specification, hence each ZigBee implementer (such as TI, Jennic, Freescale) developed a thin operating system abstraction layer. ZigBee application code as well as the ZigBee stack itself is built on top of it. For example, Jennic states that their ZigBee stack is implemented on the top of BOS (Basic Operating System)[BOS 2006]. BOS has a simple non-preemptive task scheduler, where tasks run to completion and each task has the same level of priority.

The TI implementation of ZigBee introduces OSAL (Operating System Abstraction Layer) [Texas 1996] under both the application and the ZigBee stack. The

components of the ZigBee stack are implemented as separate tasks and information is passed inside the stack using the OSAL messaging system. Task scheduling is done with a sequential traversal of all registered tasks. There is no task priority and traversal restarts after a pending event is found. In Freescale, their ZigBee MAC 2.01 based applications run under the control of a priority based, non-preemptive, event-driven task scheduler [Freescale 2008]. Interestingly, all of these task schedulers are non-preemptive and in most cases (e.g. TI, Jennic) does not even support priorities, but uses an event based FIFO scheduler like TinyOS. The major duties of the operating system abstraction layer are task handling, inter-task message handling, task synchronization, timer services, interrupt handling, memory allocation, etc.

5.1 Application Development 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 by using the Ember ZigBee implementation (version 3.0.0). 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 of a ZigBee application are shortly described below and depicted in figure 7.

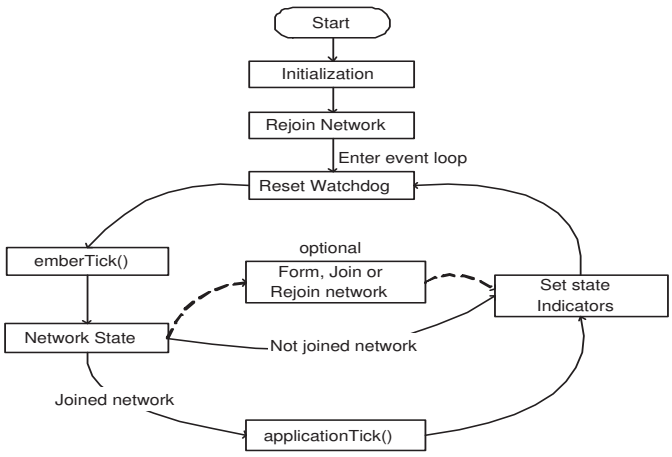

Fig. 7. Main Loop of the Ember ZigBee application

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 connected, sets the security key, initial-

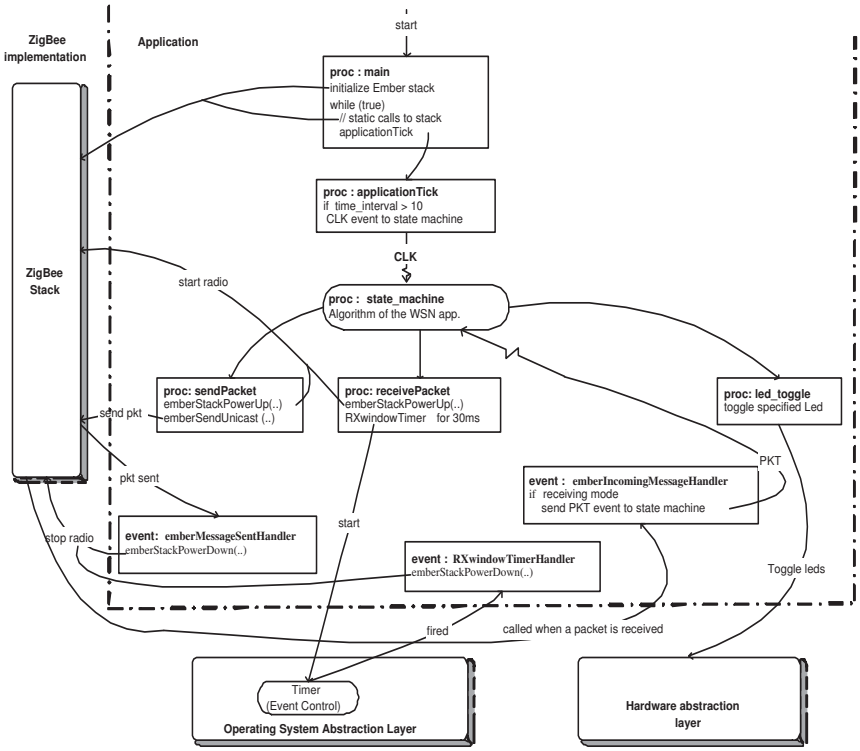

Fig. 8. Flow diagram of the FSM code integrated in Ember ZigBee

izes the application state and also sets any status or state indicators to the initial state.

The network state is checked once during each cycle of the *event loop*. If the state indicates that the network has been joined (in case of router and end device) or formed (for the coordinator), then the *applicationTick* function is executed. Inside this function the developer will put the application code. Otherwise, the node will try to join or form the network.

The flow diagram of the FSM-like code in ZigBee is shown in figure 8 (the actual code skeleton is shown in Appendix E). Here, the state machine is called from the *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 the timer (30ms). When this callback event (*RXwindowTimer-Handler*) 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 *emberIncomingMessage-Handler*. 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*

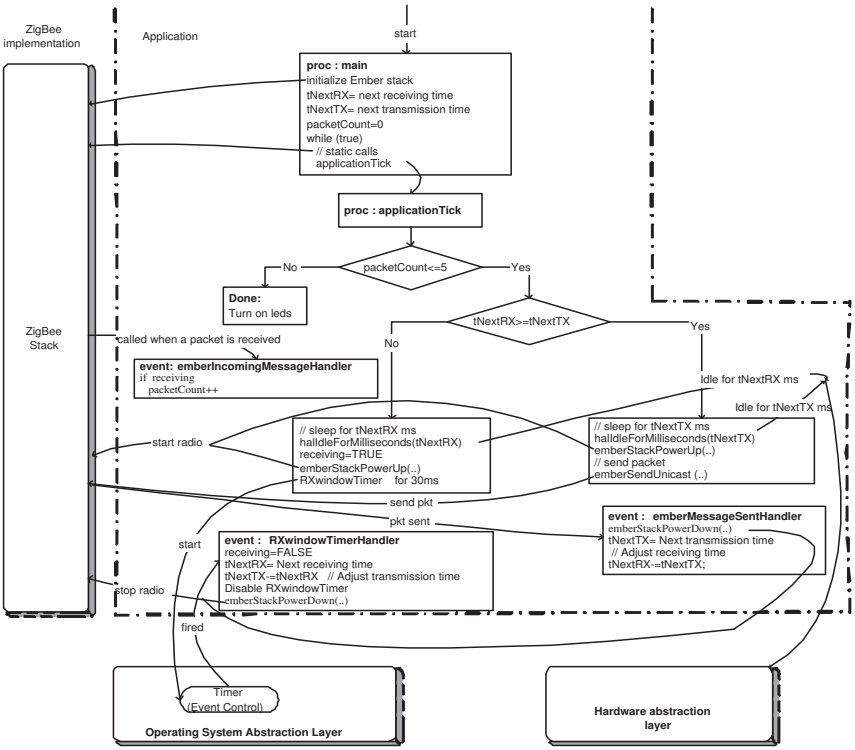

Fig. 9. Flow diagram of the non-portable code integrated in Ember ZigBee

function. Inside this event handler, we stop the radio. Implementations of *led_toggle* and *led_on* methods are simple like in MANTIS and TinyOS.

A manual implementation of the same algorithm is shown in figure 9 (the actual code skeleton is shown in Appendix F). In the main procedure, we compute the next receive and transmit times. The rest of the functionality is implemented in *applicationTick*. First, we check which time is earlier among *tNextTX* and *tNextRX*, and go into power-saving idle mode using *halIdleForMilliseconds*. After sending a packet, the algorithm again sets the next transmit time inside the *emberMessage-SentHandler*. For receiving packets, we initialize a timer event (*RXwindowTimer*) and schedule it for 30 milliseconds. In this time frame, if a packet is received, the packet count is incremented. When this event fires, we set the next receiving time and also deactivate the timer (*RXwindowTimer*).

A ZigBee application implementation depends on the node type. For example, the coordinator needs to form the network, while a router or end device needs to join in the network (figure 8 and figure 9 show the skeleton for the coordinator code). The main advantage of the ZigBee platform is that users need not think about network formation algorithms or MAC layer protocols, but can concentrate only on application development.

In summary, TinyOS and ZigBee share some common programming styles (such as event handling mechanism). In ZigBee and TinyOS, an external or internal

event activates some handler coded as a separate function, whereas MANTIS is thread-based and blocks the thread if it is waiting for specific events.

6. CONCLUSION

We examined two kinds of software platforms for wireless sensor networks: free academic operating systems, and proprietary network stack implementations. We compared the programming models imposed by the different platforms, and noted that there are two main paradigms: conventional multi-threaded or multi-tasking (embodied by MANTIS OS) and split-phase non-preemptive request/response programming (embodied by TinyOS and the Ember ZigBee implementation). While discussing differences, we also 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.

In addition we observed that, of course, academic platforms only provide barebones functionality, leaving most of the protocol stack writing to the application developer (note that a ZigBee implementation on top of TinyOS is under way). They are thus suitable only when requirements are very different, e.g. with respect to MAC or routing strategy, from what ZigBee or ZigBee Plus provide.

This FSM-like code can be written by hand or generated from different StateChart-like or Synchronous Language models, which also makes the generation of the adaptation layer to each platform easier.

Ideally, one would also like to generate automatically such code from procedural ANSI C code (with suitable restrictions, such as absence of recursion, and annotations about which blocking calls must be split). This code would be written using a standardized common WSN-specific API (similar in spirit to the Posix standard, but customized for the WSN application domain, and without assuming an underlying tasking semantics), and then customized by means of splitting across blocking calls. It is not clear at this stage how much this approach would lose in terms of code size and performance with respect to natively coding with a platform and tasking semantics in mind. Experience from the automotive software world suggests that it can be kept under a few percent, and in the end it may be the only option to preserve at least some platform independence, until a single unified standard (including API and computation model) will emerge for the WSN domain as well.

REFERENCES

Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., George, S., Gu, L., He, T., Krishnamurthy, S., Luo, L., Son, S., Stankovic, J., Stoleru, R., and Wood, A. 2004. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. 582 – 589.

Bakshi, A., Ou, J., and Prasanna, V. K. 2002. Towards automatic synthesis of a class of application-specific sensor networks. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems.*

Bakshi, A. and Prasanna, V. K. 2004. Algorithm design and synthesis for wireless sensor networks. In *Proceedings of the International Conference on Parallel Processing (ICPP04).*

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

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

BeeKit. 2004. *Freescale's BeeKit*. www.freescale.com/zigbee.

Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. 2005. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl. 10,* 4, 563–579.

BOS. 2006. *Jennic: Basic Operating System (BOS) API*. http://www.jennic.com/.

Dunkels, A., Gronvall, B., and Voigt, T. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. IEEE Computer Society, Washington, DC, USA, 455–462.

Ember. 2001. *Zigbee Wireless Semiconductor Solutions by Ember*. www.ember.com.

Freescale. 2008. *Freescale, 802.15.4 Media Access Controller MyWirelessApp User Guide*. http://www.freescale.com.

Fung, W. F., Sun, D., and Gehrke, J. 2002. Cougar: the network is the database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*.

Gummadi, R., Gnawali, O., and Govindan, R. 2005. Macro-programming wireless sensor networks using kairos. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS)*.

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

IEEE. 2003. *IEEE 802.15.4 WPAN-LR Task Group*. http://www.ieee802.org/15/pub/TG4.html.

Jaikaeo, C., Srisathapornphat, C., and Shen, C.-C. 2000. Querying and tasking in sensor networks. In *Proceedings of the 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation,and Control*.

Levis, P., Madden, S., Gay, D., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. 2004. *TinyOS: An Operating System for Sensor Networks*. Ambient Intelligence edited by W. Weber, J.Rabaey, and E. Aarts, Springer-Verlag.

Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2005. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*.

Mannion, R., Hsieh, H., Cotterell, S., , and Vahid, F. 2005. System synthesis for networks of programmable blocks. In *Proceedings of the conference on Design, Automation and Test in Europe*.

Mozumdar, M. M. R., Gregoretti, F., Lavagno, L., and Vanzago, L. 2008. Porting application between wireless sensor network software platforms: Tinyos, mantis and zigbee. In the Proceedings of the 13th IEEE International Conference on Emerging Technologies and Factory Automation, Hamburg, Germany.

Mozumdar, M. M. R., Gregoretti, F., Lavagno, L., Vanzago, L., and Olivieri, S. 2008. A framework for modeling, simulation and automatic code generation of sensor network application. In the Proceedings of the Fifth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks – SECON 2008.

Newton, R. and Welsh, M. 2004. Region streams: functional macroprogramming for sensor networks. In the Proceedings of the 1st International Workshop on Data Management for Sensor Networks, 78–87.

RTW. 2008. *Real-Time Workshop - Generate C code from Simulink models and MATLAB code*. http://www.mathworks.com/products/rtw/.

SF. 2008. *Stateflow - Design and simulate state machines and control logic*. http://www.mathworks.com/products/stateflow/.

Texas. 1996. *ZigBee: IEEE 802.15.4 from Texas Instruments*.

ZigBee. 2002. *ZigBee Alliance*. http://www.zigbee.org/.

7. APPENDIX

Appendix A: MANTIS (FSM-like portable code)

```
#include ....          // All required system includes
....
void state_machine()   // Algorithm of the simple application
{.. code shown in the example 1..  }

void clock_event(void)         { incoming_event = event_CLK;  state_machine(); }
void incoming_pkt_event(void)  { incoming_event = event_PKT;  state_machine(); }

void receivePacket(uint16_t receivingTimeStamp) // Called from state machine
{                                       //  to receive packet for specific time frame
    ....
    recv_pkt = com_recv_timed(IFACE_RADIO,receivingTimeStamp);
    if (recv_pkt!= NULL)
    { ....     incoming_pkt_event(); ...   } // Passing PKT event to state machine for the next transition
}

void sendPacket( uint8_t *packet_payload ) // Called from state machine
{                                       // to send a packet
    // construct the packet
    com_send(IFACE_RADIO, &send_pkt);      // Sending packet
}

void led_toggle (uint8_t ledN) // Called from state machine toggle Leds
{   mos_led_toggle(ledN); }

void application_thread()
{
    while(1) {
        mos_thread_sleep(10);     // Implementation of virtual CLK
        clock_event(); }          // Passing CLK event to state machine for the next transition
}
```

Appendix B: MANTIS (hand written non-portable code)

```
#include .... // All required system includes

void application_thread(){
  tNextRX=getRandNumber();
  tNextTX=getRandNumber();
  packetCount=0;
  while (packetCount<=5)  {
        if (tNextRX>=tNextTX){  // Check which timestamp is smaller
                mos_thread_sleep(tNextTX);         // Sleep until transmit time expires
                com_send(..);                      // Sending packet
                tNextTX=getRandNumber();           // Next transmission time
                tNextRX-=tNextTX;                  // Adjusting the receiving time
        } else{
               mos_thread_sleep(tNextRX);          // Sleep until receive time expires
               recv_pkt=com_recv_timed(IFACE_RADIO,30); // Receive packets for 30ms
               ....
              if  (recv_pkt=NULL )               // If a packet is received
              packetCount++;                     // Increase packet count
              ....
              tNextRX=getRandNumber();   // Next receiving time
              tNextTX-=tNextRX;          // Adjusting the transmission time
       }
   }
   // Received 6 packets.
   // Turn on all the leds
}

void start (void)
{  ....
   mos_thread_new (application_thread, 128, PRIORITY_NORMAL);
}
```

Appendix C: TinyOS (FSM-like portable code)

```
module simpleAppM
{   uses interface Boot;                          // It boots the application
    uses interface Timer<TMilli> as CLKtimer;   // Provides timing functions
    // uses all required interfaces
}
implementation{
      event void Boot.booted ()   { call CLKtimer.startPeriodic(10);}

      task  void state_Machine()
      {.. code shown in the example 1 ..  }

      void clock_event()          {incoming_event = event_CLK;  post state_Machine();}
      void incoming_pkt_event()   {incoming_event = event_PKT;  post state_Machine();}

      event void CLKtimer.fired()
      { clock_event(); }        // Passing CLK event to State Machine for the next transition

      void receivePacket(uint16_t receivingTimeStamp){
         call LPL.setLocalDutyCycle(...);       // Tune the radio in receive mode
         call Radio.start();                    // Start the radio
         call RXwindowTimer.startOneShot(receivingTimeStamp);   // 30ms for packet receiving
       }
```

```
      event void RXwindowTimer.fired()  // Receiving time expires (30 ms)
      { call Radio.stop();  }           // Stop the radio

      event message_t* Receive.receive(...)
      { incoming_pkt_event(); }  // Passing PKT event to State Machine for the next transition

      void sendPacket( uint8_t* packet_payload ){
        // Copy the sending packet data
        call LPL.setLocalDutyCycle(...);    // Tune the radio in transmit mode
        call Radio.start();
      }

      event void Radio.startDone(...) {  // Radio is turned on
        if (pkt_in_send_queue) {
           // Construct the packet
           call AMSend.send(..)}            // Send the packet
      }

      event void AMSend.sendDone(...)  // Packet sending complete
      { call Radio.stop();  }          // Stop the radio

      void led_toggle(uint8_t ledN){   // Toggling the leds
        switch (ledN) {
           case 0: call Leds.led0Toggle();
     break;
           ....
        }
       }
}
```

Appendix D: TinyOS (hand written non-portable code)

```
module simpleAppM
{ uses interface ...}
implementation
{
     event void Boot.booted(){ ...
         tNextRX=getRandNumber();
         tNextTX=getRandNumber();
         packetCount=0;
         call RXtimer.startOneShot(tNextRX);
         call TXtimer.startOneShot(tNextTX);
     }

   event void TXtimer.fired(){
     state=1;                          // Transmitting state
     call LowPowerListening.setLocalDutyCycle();  // Setting the radio totally in transmit mode
     call Radio.start();                         // Start the radio (activating the SplitControl)
     }

    event void RXtimer.fired(){
      state=2;   // Receiving state
      call LowPowerListening.setLocalDutyCycle();  // Setting the radio totally in receive mode
         call Radio.start();                    // Start the radio(activating the SplitControl)
    }

    event void Radio.startDone(error_t err) {
      if (state==1) call AMSend.send(..)                    // Sending the packet
      if (state==2) call RXwindowTimer.startOneShot(30);    // Set the time frame for receiving
    }

   event void RXwindowTimer.fired()
   {call Radio.stop(); }   // Receiving time frame expires, so stop the radio

   event void AMSend.sendDone(..)
   {call Radio.stop(); }   // Sending packet is done, so stop the radio

   event void Radio.stopDone(..){     // Radio is stopped
        if (packetCount<=5) {
        if (state==1){
       tNextTX=getRandNumber();// Next transmission time
       call TXtimer.startOneShot(tNextTX);
      }else if (state==2){
       tNextRX=getRandNumber(); // Next receiving time
       call RXtimer.startOneShot(tNextRX);}
       }else{
                 // Received 6 packets.
                // Turn on all the leds
       }
    }

 event message_t* Receive.receive(..) {    // This event is generated when a packet is received
       ...
       packetCount++;    // Increase the packet count
 }

}
```

Appendix E: ZigBee (FSM-like portable code)

```
#include  ....  // All required system includes
```

```
EmberEventControl RXwindowTimer;  // Timer object

int main(void)
{
  halInit(); INTERRUPTS_ON(); emberInit();     // Initialization tasks
  while(TRUE) {                                // Event Loop
     halResetWatchdog(); emberTick();          // Static API calls in the event loop
     if (formed_network) applicationTick();    // Procedure that contains main logic
        else formNetwork();           // Only done by the coordinator, other nodes will join
  }                                   // in the network
}

void state_Machine(){.. code shown in the example 1..  }

void clock_event(void)          {  incoming_event = event_CLK;  state_Machine(); }
void incoming_pkt_event(void)   {  incoming_event = event_PKT;  state_Machine(); }

void emberIncomingMessageHandler(...) // Call back function for message handling
{...
  switch (Message_Type) {
    ...
    case MSG_DATA:
            incoming_pkt_event();  // Passing PKT event to State Machine for the next transition
            break; }
}

void receivePacket(int16t receivingTimeStamp){
   emberStackPowerUp();      // Start the radio
   emberEventControlSetDelayMS(RXwindowTimer, receivingTimeStamp);   // Setting an event generator for 30 ms
}

void RXwindowTimerHandler(void)            // Event handler of RXwindowTimer
{  emberStackPowerDown() ;                 // Receiving time expires, stop the radio
   emberEventControlSetInactive(RXwindowTimer);    //  Disable the event generator
}

void emberMessageSentHandler(...)
{ emberStackPowerDown() ;}    // Call back function after sending a message Stop the radio

void sendPacket (int8u* packet_payload ){
   emberStackPowerUp();        // Start the radio
   // Construct the packet
   emberSendUnicast (...)      // Sending packet
}

void led_toggle(int8t ledN)  {  halToggleLed(..) }      // Toggling the leds

static void applicationTick(void) {
   if ( duration_of_lastCLKTime > 10){
       clock_event(); }      // Passing CLK event to State Machine for the next transition
}
```

Appendix F: ZigBee (hand written non-portable code)

```
EmberEventControl RXwindowTimer;
...
int main(void)
{  // Initialization code, same as Appendix E
   tNextRX=getRandNumber();
   tNextTX=getRandNumber();
   packetCount=0;
   while(TRUE) {// same as Appendix E  }
}

void emberIncomingMessageHandler(...)
{  // same as Appendix E
   case MSG_DATA:
          if (receiving)
               packetCount++;
          break;
}

static void applicationTick(void) {
   while (packetCount<=5)
   {    if (tNextRX>=tNextTX){
             halIdleForMilliseconds(tNextTX); // Sleep until transmit time expires
             send_packet(..); // Sending packet
        }else {
             halIdleForMilliseconds(tNextRX); // Sleep until receive time expires
             receiving=TRUE;
             emberEventControlSetDelayMS(RXwindowTimer, 30); // setting an event RXwindowTimer
         }
    }
   // Received 6 packets. Turn on all the leds
}

void send_packet()
{  ....emberSendUnicast(..);   }

void emberMessageSentHandler(..)  // Call back function from emberSendUnicast
```

```
{  tNextTX=getRandNumber();       // Next transmission time
   tNextRX-=tNextTX;              // Adjusting the receiving time
}

void RXwindowTimerHandler(void) // Event handler of RXwindowTimer
{ receiving=FALSE;
   tNextRX=getRandNumber();                      // Next receiving time
   tNextTX-=tNextRX;                             // Adjusting the transmission time
   emberEventControlSetInactive(RXwindowTimer);    //Disable this event until its next use.
}
```