Programming Tools for WSNs

Crnjin, Aleksandar

Abstract - *As with all other computing devices, sensor nodes have to be programmed in order to be able to do something useful. The similarity, for the most part, ends there. There is scant similarity between programming a home computer with 2 gigahertz CPU and gigabytes of RAM and storage and programming a tiny device with just a dozen of kilobytes of RAM and an 8-bit processor running on just a few kilohertz. Still, programming sensor networks isn't all that similar to embedded devices either: Programming embedded systems typically amounts to writing an assembly program and loading it into programmable ROM of the embedded device, while a sensor node usually has an operating system (such as TinyOS or ConTiki) which provides support for basic I/O operations, operation of the radio transceiver and so on. This article introduces the reader to general issues in sensor network programming and focuses on two different programming paradigms: NesC programming, for TinyOS compatible sensor nodes, and the Java Squawk virtual machine, created for Sun SPOTs.*

1. General Issues in Sensor Network Programming

In personal computers, OS primitives are used to achieve hardware abstraction, thus lifting the burden of operating the peripherals directly from the programmer. Clearly, that approach is desirable in sensor nodes as well. Still, due to scarce, *scaled* resources of a sensor node, some compromises are required, most notably the need to conserve energy resources by providing an implicitly energy efficient programming model.

The first obstacle we have to overcome is to choose the right operating system for the sensor node. All conventional OSes for embedded systems (such as Windows CE or PalmOS) require a ROM capacity of a hundred kilobytes or more; sensor nodes, however, typically have only a few kilobytes. The most popular solution today is the TinyOS, developed by U.C. Berkeley. Another option is the Java Squawk virtual machine, developed by Sun Microsystems for their Sun SPOT project. This article focuses on these two paradigms.

2. The TinyOS Operating System

For sensor nodes, a specialized operating system had to be devised; one that would provide the necessary primitives to operate the sensor node hardware, but would also cope with the quite limited resources of a sensor node. These limitations include:

Not enough memory for stack. All TinyOS programs have to operate within a single context, as it is impossible to perform traditional context switching, due to a very small amount of memory available for stack. This also means that TinyOS programs can't rely on registers to save state.

Limited amount of memory. This means that what available memory there is has to be allocated carefully. Dynamic allocation of memory is prohibited; the TinyOS *components* are arranged into a *configuration* at compile-time, and individual components each get their preassigned portion of memory, the *memory frame.*

Limited amount of energy. Special care is taken to ensure that the battery power is conserved as much as possible. Busy waiting and interrupt polling is prohibited in all TinyOS compatible devices. TinyOS programs execute only in response to *events*; this is called the *event-driven* programming model.

2.1. TinyOS Component Model

To address these issues, the *TinyOS Component Model* was devised. The parts of code in this model are organized into distinct entities based on functionality. For example, there may be an entity for operating the integrated radio-transceiver unit. (In fact, there is such an entity in TinyOS, it's the **rfm** system component.) These entities are associated with a statically allocated memory portion (called *memory frame)* and interconnected with other such entities through *interfaces.* Such completely described entities are called *components.*

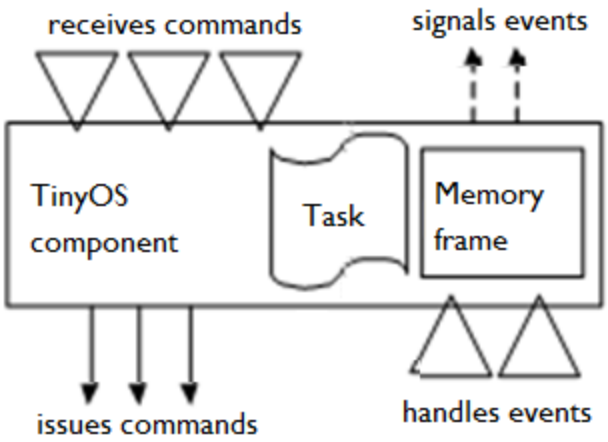


Figure 1: Block view of a TinyOS component

The *memory frame* is allocated at compile-time, based on total memory requirements of the component.
The *interface* consists of registered commands and events, through which components communicate. The main difference between the two is in the fact that events propagate upward in the connection chain (from device driver components, such as aforementioned **rfm**, through bottom level user components up to the top level user components) and commands propagate downward in the connection chain. One component may choose to implement several interfaces, each consisting of a number of commands and events. More on this will be mentioned later, in the chapter on component interconnection.
The *code* contained in an individual TinyOS component consists of:
Command handling routines, which they execute in response to *commands* issued from other connected components;

Event handling routines**,** which they execute in response to *events* signalled by connected components;
Tasks, which components themselves schedule for later execution.

System Components and Pre-defined Events

TinyOS comes with a certain number of *system components* which perform the function of device drivers. Examples include the already mentioned **rfm** transceiver-operating component, **Photo** and **Temperature** – the components for manipulating photo and temperature sensors, and so on.
In addition to pre-defined system components, there is also a pre-defined interface **StdControl,** consisting of events **Init, Start** and **Stop.** Each TinyOS component has to implement this interface by providing handlers. For example, an **Init** event handler defines the response of the component to the event of system initialization. A component could use this handler in order to initialize it's state to initial values. Actual execution of a program begins with handling of the **Start** event, in the mandatory **Main** component.

Component Configurations

Components are either **modules** or **configurations.** Each TinyOS "program" is a configuration of interconnected modules; each module is an encapsulation of code based on functionality, reminiscent of classes in object oriented programming languages. Each configuration and module is described in a separate **.nc** file. Modules are connected through *interfaces.* An interface is a collection of events a given component can signal or be notified of, and commands that a component can issue or obey. Interfaces are also defined in their separate **.nc** files. A configuration is formed by connected by "software wiring" of the interfaces. This is accomplished in two steps:

1. each module lists the desired interface in either it's **uses {…}** block, or it's **provides {…} block;**
2. a special line in the configuration code is added, which establishes an unidirectional link between an *interface provider* and *interface user***.** This is achieved with following line of code:

User.UserInterfaceName ->
Provider.ProvInterfaceName
or, alternatively:
Provider.ProvInterfaceName <-
User.UserInterfaceName

(The UserInterfaceName part may be omitted, if the interface names are the same.)

One example of an interface is the Timer interface, through which the system component for the on-board timer notifies the user component of regularly spaced *tick events.* The description of this interface is given in Timer.nc:

```
interface Timer {
  command result_t start(char type, uint32_t
interval);
  command result_t stop();
  event result_t fired();
}
```

From the source code, it should be clear that the interface specifies one event – fired() – through which the Timer provider notifies the user of timer ticks, and accepts two commands – start, which accepts the type and interval parameters and starts the timer, and stop(), which stops the timer.
The actual semantics of the words *provider* and *user* might not be immediately clear: The events and commands which are the part of the interface are specified in the interface's **.nc** file. If a module declares itself to be a provider of a given interface, then it's responsible to implement all commands listed by the interface, and acquires the right (which it may or may not use) to signal events to those users of said interface to which it is connected through wiring. Likewise, if a module declares itself to be a user of an interface, then it is responsible to implement event handlers for every single event specified by the interface, and it attains the right to issue commands to the provider of the used interface.
In other words, the provider can signal events to users, and users can issue commands to the provider.

NesC "Hello World" program: Blink

To give an overview of the issues involved, we consider a simple application that comes bundled with the TinyOS distribution, called **Blink.** As it name suggests, the Blink application uses the on-board timer to periodically change the state of a sensor node's LED, producing a blinking effect. The Blink application consists of four connected components: Main, BlinkM, SingleTimer and Leds, as seen in the configuration declaration (Blink.nc):

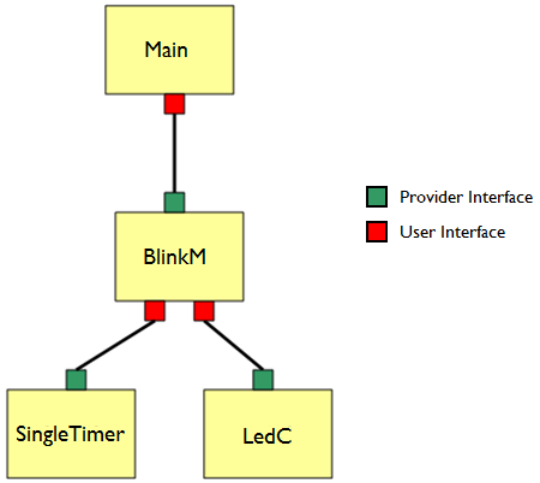
```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  // connecting Main to SingleTimer and BlinkM through
StdControl
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Timer -> SingleTimer.Timer; // connecting BlinkM to
SingleTimer
  BlinkM.Leds -> LedsC;          // connecting BlinkM to LedC
}
```

In this example, we can note the following:
Components Main, BlinkM, SingleTimer and LedC are used. Code for these components is elaborated in separate files (for example, BlinkM code is found in BlinkM.nc)
In the last four lines before the ending bracket, we can see how these components are interconnected – through "wiring" of their interfaces, using the notation User.Interface -> Provider.Interface.
BlinkM, the central module of the application, provides the StdControl interface, of which the implicit component Main is the user. The execution of the program "begins" with Main issuing the commands "Init" and "Start" to BlinkM. In response to these commands, BlinkM will start the timer, through the Timer interface. As BlinkM is notified of timer tick events, it toggles the LED, by issuing commands to the LedC component through the Leds interface.

```
//Implementation for Blink application:
// the red LED is toggled whenever Timer fires.
module BlinkM {
 provides {
   interface StdControl;
 }
 uses {
   interface Timer;
   interface Leds;
 }
}

implementation {
 //Handling of the Init command, issued by Main
 // just pass the Init command on to the Leds component
 command result_t StdControl.init() {
   call Leds.init();
   return SUCCESS;  // SUCCESS is always returned
 }
 // Handling of the Start command, issued by Main:
 // set the rate for the clock component.
 command result_t StdControl.start() {
 // Start a repeating timer that fires every 1000ms
   return call Timer.start(TIMER_REPEAT, 1000);
 }
 // Handling of the Stop command, issued by Main:
 // stop the timer
 command result_t StdControl.stop() {
   return call Timer.stop();
 }
 // Handling the Fired event, issued by Timer:
 // issue redToggle command to Leds
 event result_t Timer.fired(){
   call Leds.redToggle();
   return SUCCESS;
 }
}
```

The configuration of the Blink application can be summarized in the following diagram:



TinyOS: Conclusion

Programming in TinyOS is much simpler and quicker compared to the Full Custom model of assembly programming, and arguably easier, too. Still, the TinyOS programming model has suffered criticism for its still steep learning curve, especially for programmers accustomed to programming in established languages (such as Java).

3. Sun SPOTs and the Squawk virtual machine

The Squawk virtual machine, created by Sun Microsystems for their Sun SPOT technology, is one attempt at providing easier-to-use programming model to developers for sensor networks. Using Sun SPOTs and Squawk, a sensor network developer can write applications in a slightly modified versionof Java.
Squawk itself is a Java virtual machine running directly on SPOT hardware, without an underlying operating system. To facilitate execution of such sophisticated software, Sun SPOTs had to be designed with significantly more powerful hardware, compared to the TinyOS nodes (a Sun SPOT has a 180MHz 32-bit processor). This might mean greater energy consumption and less battery life, compared to TinyOS nodes.

Programming Sun SPOTs

As mentioned already, Sun SPOTs are programmed using a variation of Java language. SunSPOT applications are MIDlets; the main class of the SunSPOT application extends the javax.microedition.midlet.MIDlet class, so constructs typical for MIDlets, such as startApp(), pauseApp() and destroyApp(), form the skeleton of a Sun SPOT application. Therefore, an "entry point" for a Sun SPOT application is the startApp() method.
The whole functionality of the SPOT is abstracted using the EDemoBoard class. Typically, a programmer will obtain an instance of this class using EDemoBoard.getInstance() in the body of the startApp() method. Inputs (sensors and switches) and outputs (LEDs) are then reachable through Java interfaces, which are obtained using demoBoardInstance.get*xxx* () methods. For example, a temperature reading can be obtained using:

```
ITemperatureInput ourTempSensor =
EDemoBoard.getADCTemperature();
double celsiusTemp = ourTempSensor.getCelsius();
double fahrTemp = ourTempSensor.getFahrenheit();
```

A SunSPOT "Blink" application

As for TinyOS/nesC, we now present a Blink (Hello World) application for Sun SPOTs.

Sun SPOTs/Squawk: A Conclusion

Sun SPOTs certainly provide an easier programming model and as a result they are very popular within certain educational circles. However, their bulky size (compared to TinyOS-based platforms), higher price (for non-educational use) and higher battery consumption still make them somewhat impractical for some industrial applications.

```
package org.sunspotworld;
import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.peripheral.ISwitch;
import com.sun.spot.senso
            rboard.peripheral.ITriColorLED;
import com.sun.spot.util.*;
import javax.microedition.midlet.MIDlet;
import
javax.microedition.midlet.MIDletStateChangeException;

public class Blink extends MIDlet {
  private ITriColorLED [] leds =
EDemoBoard.getInstance().getLEDs();
  protected void startApp() throws
MIDletStateChangeException {
    System.out.println("Hello, world");
    ISwitch sw1 =
EDemoBoard.getInstance().getSwitches()[EDemoBoard.SW
1];
    leds[0].setRGB(100,0,0);       // set color to moderate red
    while (sw1.isOpen()) {         // done when switch is
pressed
      leds[0].setOn();           // Blink LED
      Utils.sleep(250);          // wait 1/4 seconds
      leds[0].setOff();
      Utils.sleep(1000);         // wait 1 second
    }
    notifyDestroyed();           // cause the MIDlet to exit
  }
  protected void pauseApp() {
    // This is not currently called by the Squawk VM
  }

protected void destroyApp(boolean unconditional) throws
MIDletStateChangeException {
    for (int i = 0; i < 8; i++) {
      leds[i].setOff();
    }
  }
}
```

4. References

[1] Philip Lewis, TinyOS Programming, 2006.
[2] TinyOS Documentation Wiki
[3] SunSPOT Developer's Guide, SUN Microsystems, 2009.