Faster code generation for wireless sensor networks

Stefano Olivieri, Mostafizur Mozumdar, Luciano Lavagno, & Laura Vanzago - April 16, 2009

Wireless sensor networks (WSN) have become a hot research topic and show much promise to become a driver of current and future microelectronic technologies. This emerging technology offers exciting potential for numerous application areas including environmental, medical, military, transportation, entertainment and crisis management.

In the main scenario, users retrieve information from a WSN by injecting queries and gathering results from sensor nodes, called “motes.” A single mote might consist of sensing, computing, communication, actuation, and power components, which are integrated and packaged into a miniaturized mote. A WSN usually consists of tens to thousands of such nodes, communicating through wireless channels for information sharing and cooperative processing.

Because of the requirement of unattended operation in remote or even potentially hostile locations, sensor networks are extremely energy-limited.

The major challenges in the WSN domain include sensing and collecting data from sensors and then evaluating it to formulate meaningful information such as generating alarms or decision-making while minimizing energy consumption.

Currently, researchers are developing software applications at different stack levels for WSNs. However, several serious barriers to the success of WSN in commercial applications remain:

Two primary types of functional analysis packages are currently available. One is very platform- and operating system (OS)-specific (for example, TOSSIM), which provides a binary API to model the OS and the motes, with limited capabilities for reusing existing channel models, tracing, collecting statistics, and so on. The other set includes generic network simulators (such as OmNet, NS, etc.), sometimes enhanced with models tailored to the radios and channel used by WSNs.

Both show significant drawbacks in the application development phase. The first group makes it quite difficult to port an application to a different platform, for example from TinyOS to MANTIS, or to a ZigBee-compliant platform, without very extensive code rewriting and a significant amount of debugging. The second group still leaves a lot of detailed platform-dependent code to be developed. 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 different environments. Moreover, if one finds a bug at the end of the development and integration phase, the only solution is led-based debugging, which is extremely time-consuming.

To solve these issues, system engineers need to be able to model applications using high-level abstractions, and to simulate them using configurable and realistic topologies for the network itself. Moreover, suitable tools to automatically generate code for several target operating systems would
be desirable. As a result of a joint research project of STMicroelectronics, Politecnico di Torino, and The MathWorks, we prototyped a framework for modeling, simulating, and generating code for sensor network applications which has been defined based on MathWorks tools.

At the top level, a sensor network model with Simulink can be built and serves as a time-based simulator for dynamic systems. Such a model has two major components: the wireless channel model (called communication medium in the following) and the node block.

One can implement the communication Medium block in C, so that it can be modified to reuse any existing channel and connectivity models. We are now working, for example, on refining it using the ZigBee physical and medium access control standards.

The node block contains a number of motes; each of them is fully parameterized and contains a model of the hardware and software platform, including for example a timer and the LEDs, as well as a parameterized Stateflow block representing the application. Stateflow is a finite-state machine language used to model and simulate control logic and flow charts. Here, it implements the algorithm (application, middleware, or device drivers) running inside each mote. The timer, for example, is used for generating time events for the Stateflow algorithm. It is possible to model sensor networks running either identical or different applications in each node. To model a new sensor network application based on this framework, an application developer only needs to modify the template algorithm implementation (Stateflow library object) and set the connectivity parameters of the nodes in the communication medium block.

A simple simulation framework for wireless sensor networks.

To see a bigger version of this graphic click [here](#).

**A refined model**

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 that represents the desired behavior.

After functional analysis of the algorithm, the next step is to generate code automatically for any supported WSN platforms from the Stateflow representation of the algorithm, using a customization of RealTime Workshop Embedded Coder software, which can generate embeddable ANSI C code for Simulink and Stateflow blocks.
TinyOS and Mantis are two operating systems that represent exemplary paradigms for developing embedded applications in wireless sensor networks.

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 commands and events that must be implemented by the provider and user of the interface respectively. Events are the software abstractions of hardware or software events such as the reception of a 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 applications are written in nesC, an extension of the C language. MANTIS is a light-weight, multithreaded operating system that is capable of multitasking on energy-constrained distributed sensor networks. The scheduler of MANTIS supports thread pre-emption, allowing the operating system to switch between active threads without waiting.

In this way, the responsiveness of the operating system to critical events can be faster than in TinyOS, which is non pre-emptive. The scheduler of MANTIS is priority-based with round robin. The kernel and APIs of MANTIS are written in standard C.

The computational bodies of both MANTIS threads and tasks, and commands of TinyOS are essentially written in C. In MANTIS, integration is straightforward, because ANSI C output of Real-Time Workshop Embedded Coder can be used directly as a user thread.

In TinyOS, programs are composed of high-level abstractions such as components, modules, and interfaces which are absent in ANSI C.

The bodies of these abstractions, however, are written in C, so all that is needed is a code converter, which takes C code as an input and splits the code into sections (includes, defines, functions, etc.) and uses these sections to generate nesC code.

The Target Language Compiler (TLC) from MathWorks may be used to implement such a code converter. It provides mechanisms to 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. When working with Stateflow, application developers do not have to think about the actual implementation of these generic functions in TinyOS or in MANTIS, since they have been implemented in the TLC library and can be targeted to any operating system.

By using TLC scripts, a developer can automatically generate a TinyOS or a MANTIS application, and then can compile and execute these applications for the target platform without modification.

This is more powerful than a pure hardware independent and operating system-independent abstraction API, because TLC can actually rewrite and modify the source code to split it and suit it to the underlying platform.

This enables much more optimization than, for example, a compiler can do, and allows one to approach the amount of manual code restructuring that a programmer would do to port an application between very different hardware and software platforms. Results from the implementation of a typical WSN algorithm on a TinyOS platform show that the increment of code size due to automatic code generation versus manual code writing is as little as 5 percent.
Framework for modeling, simulation and code generation of a WSN application.

To see a bigger version of this graphic click here.

The advantage of using this framework, as opposed to manually writing WSN application code, is that it takes only a few hours to get a new application running. Thus, developers can concentrate on spending the time modeling and simulating at the functional level, and then code generation, compilation, and execution on different platforms are automated and extremely fast.

A complete framework for modeling, simulation, and multiplatform code generation based on MathWorks tools enables system engineers to effectively design WSN applications and map them on a wireless network. These tools offer applications developers rich libraries for digital signal processing and control algorithm behavior simulation, along with a broad variety of debugging and analysis tools, such as animated state chart displays, scopes, and plots.

Moreover, with this framework users can automatically generate the complete application code for several target operating systems from the same simulated and debugged model, without thinking about the details of the target platform implementation.

Finally, reimplementing a new application (for example, code generation, compilation, and execution) on a specific platform is automated and very fast, at the cost of a negligible increase in code size compared to hand-written code.

Stefano Olivieri is a senior application engineer for The MathWorks.

Mostafizur Mozumdar is a PhD student, Politecnico di Torino, Italy.

Luciano Lavagno is an associate professor, Politecnico di Torino, Italy.

Laura Vanzago works for STMicroelectronics.
This story appeared in the April 2009 *EE Times Europe* special issue on Industrial Automation and Test & Measurement. European residents who wish to receive regular copies of EE Times Europe, subscribe [here](#).

See other stories from this issue [here](#).

You can view and download a digital edition of this special issue of *EE Times Europe* [here](#).