Getting NesC and TinyOS Ready for Prime Time

William P. McCartney
Electrical and Computer Engineering
Cleveland State University
2121 Euclid Ave, Cleveland OH 44115 USA
w.p.mccartney@csuohio.edu

Nigamanth Sridhar
Electrical and Computer Engineering
Cleveland State University
2121 Euclid Ave, Cleveland OH 44115 USA
n.sridhar1@csuohio.edu

Abstract

NesC provides us with many useful additions to the standard C language. Much of these contributions would be beneficial to embedded systems at large, and not just sensor networks. Many papers talk about sensor networks becoming ubiquitous in the next few years. However, without leveraging existing embedded systems developers, this seems far fetched. There are several problems which must be overcome in both the development focus and tools available before TinyOS, or any future platform, is ready for prime time. We discuss some of these problems, and suggest techniques to address them.

Categories and Subject Descriptors

C.3 [Special purpose and Application-based systems]: Real-time and embedded systems; D.1 [Programming Techniques]; D.3 [Language Constructs and Features]

General Terms

Design, Languages

Keywords

Wireless sensor networks, embedded systems, programming methodology

1 Introduction

Wireless sensor networks (sensornets) are an important class of embedded networked systems that hold the promise of ubiquitous deployment of nodes in our physical environment. Advances in the area of developing small and inexpensive devices capable of capturing sensory input, and processing such input, have provided a rich set of technologies and platforms for use in building large-scale sensornets [4, 7, 13]. Along with the development of these hardware platforms, the research community has also been actively engaged in developing protocols and algorithms aimed at combating the challenges presented by the extremes that are a consequence of the hardware being small and inexpensive — high rate of failure, limited resources, very large numbers — to name a few [1, 14, 17, 18]. In addition, the community has produced a number of competing software platforms for programming sensornet applications [2, 3, 5, 11].

Yet, building sensornet applications is still hard. There is still no consensus among sensornet developers on which software system is the standard. Even worse, there is very little adoption of the languages and tools developed for the sensornet community, by the larger superset of embedded systems developers. Why is this? Unless we ask this question seriously, and engage the embedded systems developer community properly, the vision of sensor network ubiquity is going to be difficult to achieve.

This is the position we take in this paper. We argue that while nesC and TinyOS have become the de facto standard of the sensornet programmer community, they require some changes, albeit small and localized, before they can be popularly embraced by embedded systems programmers outside the context of sensornets. We show examples of such problems, and propose ideas for solving them.

2 What’s Holding TinyOS Back?

2.1 Hardware Abstraction Layers

To achieve the goal of ubiquity, we require platform-independent hardware interfaces. To further explain this, we define a platform as a specific board. This platform contains several different chips, following the TinyOS definitions of platforms. We need interfaces which are less specific than even a single chip. This means we need to make an interface or collection of interfaces which actually support a superset of any one given piece of hardware’s implementation. While this may make it slightly more complicated at directly interfacing a given layer (since multiple code configurations...
must be supported), such an intermediate layer can simplify application development through code reuse even though some portions may become more complicated.

For example, consider the RS-232 port of any microcontroller. And to help simplify the example even further, let us only discuss the transmission side. The practical implementation of a serial port is typically irrelevant to almost all applications. These majority of applications which do not care about the specific detail of the implementation of the serial port should be able to use higher-level constructs to interface it. Such a higher-level interface is already done very well inside of TinyOS. To interface a lower layer, however, a hardware-specific interface must be used. We need to try to describe how all possible interfaces could be implemented in hardware.

Most notably, different microcontrollers provide different interrupts to help support their peripheral’s architecture. Since we are trying to describe how to interface different architectures, we need to provide all of the possible interrupts, and then, preferably at compile time, enable developers to figure out which interrupts are actually available on the platforms they are currently compiling on. Another feasible approach would be to have a handful of platform-independent interfaces which describe most of the common mappings of interrupts, and have the developer program interfaces to one (or several) of these. A very good example of this is: a DMA-driven RS-232 port has a very different interface than a simple register interface.

It can be argued that some of this is already solved in the current architecture of TinyOS. TinyOS already provides a pair of platform-agnostic UART interfaces, namely UARTbyte and UARTstream. While these interfaces fully describe the needs from a serial port that sensornet applications currently require, they do not (at least, not fully) map out the underlying hardware. For example, the hardware presentation layer does not actually expose the ATMega’s data ready interrupt signal. There is no way to get at this signal or turn it on, or disable it through that interface. And that is not to say that the hardware independent layers do not expose it but actually, the lowest level interface in TinyOS does not expose this interrupt. Even assuming that the hardware presentation layer fully exposes all the underlying hardware, all this enables developers to do is to make more platform-specific code.

By allowing developers to only wire to platform-specific interfaces, developers will know which interrupts are available with each different platform. This argument really detracts from some of the beauty of TinyOS, where we can write our applications once and retarget it to a myriad of platforms.

What about the project that has to interface with a GPS system, or another specialized sensor, or a barcode scanner, or any other physical device in the world?

With the exception of the ADCs, the current state-of-the-art in TinyOS is to require the user to step down into the platform-specific interface. Now that is not to say that the wiring files are not platform-specific. Since the user will most definitely have specified (at the very least), which hardware components they are using (ports, pins, etc). The realistic effects of this kind of separation could, through other ongoing research efforts, create a large number of platform-agnostic drivers. Currently, the TinyOS contrib is filled with several platform-specific drivers for various pieces of hardware.

All of this hardware-specific nature of the drivers is not limited to the wiring files. Developers need to be notified of the underlying hardware at compile time. If the system were written in C, it would be convenient to have it as a pre-processing definition. So platform-specific code that required, or optimized based upon, the availability of an interrupt could be enabled or disabled at compile time, without requiring the assistance of the user of the module.

This leads to pointing out the problems with the current layers in the TinyOS Hardware Abstraction Architecture. It is not so much that these layers were not designed correctly. In fact, they were designed very well with sensornet development in mind. However, to go beyond sensornet development alone, TinyOS needs a hardware abstraction layer which can represent a myriad of hardware. Currently in TinyOS, there are three layers in the hardware abstraction architecture [6]. These layers are:

- the hardware presentation layer, which simply maps out the underlying hardware in a platform-specific manner;
- the hardware abstraction layer, which builds on top of the presentation layer and provides a simplified interfaces, while still exposing all of the underlying hardware; and
- the hardware independent layer, which uses the hardware abstraction layer to provide a hardware independent services.

These layers are well-designed, for the task of developing sensornet applications on existing hardware platforms. In order to be more conducive to developing general microcontroller-based embedded systems applications, there needs to be another layer, which is hardware-centric, but platform-independent. Semantically, this falls inside of the hardware independent layer but practically, it does not quite make sense. This is mainly due to what currently populates the hardware independent layer: components such as the CC1000 driver itself, which hypothetically could be made platform-independent by building on top of some new interim layer. The CC1000 basically requires a handful of specific interfaces to be implemented. After digging through the comments, these can be found, and possi-
bly implemented for another platform, but this creates an extra burden upon people wanting to utilize this in an embedded application. There should be another chip-agnostic layer which includes these interfaces which are used by the CC1000.

This layer, called the hardware access layer, may have multiple interfaces which fully describe a single type of hardware interface. This should only happen when the hardware interface differences are not reconcilable. This may mean that each processor core has a different interface, but it is not specific to a single processor. For instance, the differences between an ATMega128 and an MSP430F149 interface for a specific peripheral may be irreconcilable. That does not mean that there are significant differences between an ATMega 128 and an ATMega 8. This distinction between families of chips must be made to developers. To convey this, we need to change some of the Hardware Presentation Layer interfaces to represent their non-platform-specific nature. Much of this could simply be a documentation/nomenclature change, keeping in mind that other platforms may be conducive to existing interfaces. For instance, a PIC microcontroller’s serial interfaces may be similar enough that it could be expressed through an ATMega-style interface.

2.2 What’s in a .platform

The new layer that exposes the hardware alone is not enough to enable people to use TinyOS for other embedded applications. The current focus of TinyOS is on making sensornet applications, on a handful of predetermined platforms — very simple. In most microcontroller-centric embedded systems, each new application is often a new platform. Now making a platform in TinyOS is not such a simple task. Even though many new platforms have been added to TinyOS\(^1\), this involves a fairly extensive knowledge of TinyOS, and is not a task for a novice nesC user.

\(^1\)Documentation at http://www.tinyos.net/tinyos-2.x/doc/nesdoc/.

For argument’s sake, let us assume that the specific processor, which a developer is using, is fully supported through existing modules. Let us also assume that there is no radio or stack even required in this application. Creating a platform involves the developer modifying not only the build system, but also creating a platform directory and placing all the required source files in that platform directory, along with a .platform file. This .platform file is not a makefile, or a C file, or a header file, or a nesC file — it is actually a Perl file. This file is included by the nesC compiler, and determines the included parse order to the nesC components. It also targets the nesC compiler to specific C compiler.

The inability to rapidly create platforms of even existing components is an obvious barrier to any general embedded systems development. A simple solution to this problem to is to add a “special target”. This special target would force the build system to rely on the local Makefile for the build path and target compiler. This would allow developers to use modules from one of the chips directories without having to actually create a platform directory and a .platform file.

2.3 NesC Syntax and Wiring

NesC’s object model could be incredibly empowering to embedded systems developers. It could allow a new level of call-backs, without the requirements of function pointers and complex preprocessor macros. By design, it can help developers create safe, efficient, event-driven applications. Even the consistent scheduler framework can help standardize many different applications across many platforms. Currently, there are many different ways task scheduling is handled. Using post statements could standardize how different main-loop tasks are written.

However, the benefits of nesC are not without practical drawbacks. The most notable drawback is that nesC is not exactly C. It has been called “a dialect of C” [10]. The truth of the matter is somewhere in the middle. The code portions of the applications are extremely similar to ANSI C. The most notable additions are:

- atomic statements — which cleanly disable interrupts in a platform-independent manner.
- post statements — which clearly mark a task to be executed by the scheduler.
- call and signal statements — which invoke commands and events respectively.

The atomic and post statements alone can be happily digested by even a novice embedded programmer, since they are analogous for commonly-used routines and practices. However, the call and signal statements are not analogous to any existing practices. Rather, while they may be similar to calling a routine and parameterizing it with a callback, syntactically the plain C code is shorter, and much different. Most notably, the addition of the keywords call and signal
is not so natural. One way to make nesC look closer to traditional C is to make the call and signal keywords optional. This would allow the existing code base and coding standards to remain unchanged, while enticing other developers with a simpler syntax and possibly simpler examples.

In some senses, call and signal keywords are redundant (ignoring any namespace separation). Interfaces are declared as provided or used at the top of an implementation. But inside of the implementation, only interfaces which are used can be called and only interfaces which are provided can be signaled.

The differences between nesC and standard C do not end with the syntax alone. The wiring files and their corresponding component sections create an added barrier for developers switching from standard C to nesC. Our approach to enable more developers to understand, and be able to create wiring files, is to have a development environment which fully supports graphically creating and modifying existing wiring files. The graphical support cannot simply end with the wiring. It has to also enable developers to create new components with empty prototypes automatically generated for any desired interface. Such a development environment would also require tools for rapidly creating interfaces.

If such a development environment were to be produced, the abstraction of wiring could be maintained. Without such a development environment, a new user is subjected to learning a whole host of new syntax. Arguably, users have to learn an entire new declarative language in the wiring files alone. This is a decent sized roadblock to anyone who wants to do a quick evaluation of nesC. For instance, it would take a first-time nesC user (but experienced in C), an unnecessarily long time to reproduce Blink without copying examples.

In fact, most TinyOS papers do not include much code from wiring files. Instead, they show graphical representations of said wiring files [5]. Obviously, the graphical form of these wiring files rapidly conveys a significant amount of information versus the source code files. In our previous work, we have presented a development environment (TOSDev) that takes this view [12]. Wiring diagrams in TOSDev are displayed only in their graphical form. The development environment does not allow a developer to edit the wiring diagram in text at all\(^2\).

This is not to say that the graphical editing of the wiring file can immediately be as expressive as the textual form. There are several important details, which must be fully supported in the graphical wiring editor. First, parameterized interfaces must be fully supported. This means interfaces which not only have one parameter, but also multiple parameters. Second, conditional wires must be allowed. Currently, nesC wiring files use #ifdef statements to conditionally wire interfaces. It is also important to fully support conditional components, and conditional interfaces. Our current work on TOSDev is focused on including these concepts. There have been many other development environments for TinyOS, but none of the others included a graphical wiring editor. [8, 15, 16]

3 Conclusion

In this paper, we have expressed our position that while TinyOS and NesC are popular among sensornet application developers, that they are not ready for widespread adoption in the larger embedded systems community. It is essential for the health and growth of sensornets as a strong force, that embedded systems developers outside of the sensornet context embrace, use, and contribute to the TinyOS vision. There is a lot of momentum among the TinyOS community to grow the system and make it attractive to developers at large. There are other efforts focused on making the programming system more open to a wider variety of different higher-level and lower-level components. As an example, the MAC Layer Architecture presented in [9] provides a unified component-based view to implement MAC protocols for TinyOS.

In our view, the three major needs from TinyOS are:
- A hardware access layer
- A build system with a platform per application
- A solid graphical development environment

All things considered, it would be incredibly beneficial to have embedded systems developers contributing to TinyOS. Imagine TinyOS having standard components to interface LCDs, keypads or capacitive sensing switches. Exploring TinyOS in an industrial control environment, communicating over Modbus or another protocol. Or possibly in homes communicating over X10, or even in a more conventional way over TCP/IP or UDP.

Also remember, TinyOS and nesC provide a solid framework for doing event-driven programming — which can enable people to write more complicated state machines. For instance, people could possibly implement software RS-232 ports or software USB ports. These applications have been implemented on microcontrollers before, but that is the exception, not the rule. These could be standard components in TinyOS enabling many other applications from hardware with lesser capability.

Not to mention the hundreds of possible processors supported — along with many more drivers being implemented. Imagine being able to transparently switch between hundreds of platforms, instead of only a few.

If these three needs are met, the embedded developers will trickle into trying TinyOS. This can help build the TinyOS community, while growing the number of possible sensornet applications. The sensornet applications of the future will not come from academia, but

\(^2\)Of course, the developer is free to open up the file in a different text editor and modify it, but that is not the point.
from industry and indirectly from general embedded systems. It is only natural to entice embedded systems developers into the folds of TinyOS.

Acknowledgment

This work was supported by a CAREER grant from the NSF (CNS-0746632). The authors gratefully acknowledge the NSF for its support.

References


