CAMBRIDGE

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Excerpt
More information

Part I

TinyOS and nesC

© Cambridge University Press www.cambridge.org

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Excerpt
More information

1 Introduction

This book is about writing TinyOS systems and applications in the nesC language. This chapter gives a brief overview of TinyOS and its intended uses. TinyOS is an open-source project which a large number of research universities and companies contribute to. The main TinyOS website, `www.tinyos.net`, has instructions for downloading and installing the TinyOS programming environment. The website has a great deal of useful information which this book doesn't cover, such as common hardware platforms and how to install code on a node.

1.1 Networked, embedded sensors

TinyOS is designed to run on small, wireless sensors. Networks of these sensors have the potential to revolutionize a wide range of disciplines, fields, and technologies. Recent example uses of these devices include:

Golden Gate Bridge safety High-speed accelerometers collect synchonized data on the movement of and oscillations within the structure of San Francisco's Golden Gate Bridge. This data allows the maintainers of the bridge to easily observe the structural health of the bridge in response to events such as high winds or traffic, as well as quickly assess possible damage after an earthquake [10]. Being wireless avoids the need for installing and maintaining miles of wires.

Volcanic monitoring Accelerometers and microphones observe seismic events on the Reventador and Tungurahua volcanoes in Ecuador. Nodes locally compare when they observe events to determine their location, and report aggregate data to a camp several kilometers away using a long-range wireless link. Small, wireless nodes allow geologists and geophysicists to install dense, remote scientific instruments [30], obtaining data that answers other questions about unapproachable environments.

Data center provisioning Data centers and enterprise computing systems require huge amounts of energy, to the point at which they are placed in regions that have low power costs. Approximately 50% of the energy in these systems goes into cooling, in part due to highly conservative cooling systems. By installing wireless sensors across machine racks, the data center can automatically sense what areas need cooling and can adjust which computers do work and generate heat [19]. Dynamically adapting

CAMBRIDGE

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Excerpt
More information

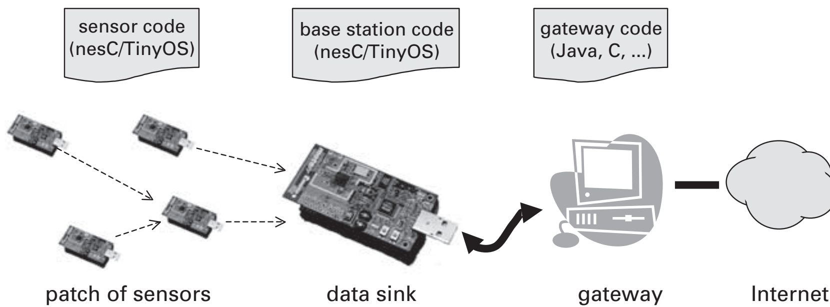

Figure 1.1 A typical sensor network architecture. Patches of ultra-low power sensors, running nesC/TinyOS, communicate to gateway nodes through data sinks. These gateways connect to the larger Internet.

these factors can greatly reduce power consumption, making the IT infrastructure more efficient and reducing environmental impact.

While these three application domains are only a small slice of where networks of sensors are used, they show the key differences between these networks and most other computing systems. First, these "sensor networks" need to operate unattended for long periods of time. Second, they gather data from and respond to an unpredictable environment. Finally, for reasons of cost, deployment simplicity, and robustness, they are wireless. Together, these three issues – longevity, embedment, and wireless communication – cause sensor networks to use different approaches than traditional, wired, and human-centric or machine-centric systems.

The sheer diversity of sensor network applications means that there are many network architectures, but a dominant portion of deployments tend to follow a common one, shown in Figure 1.1 [21, 26, 30] of ultra-low power sensors self-organized to form an ad-hoc routing network to one or more data sink nodes. These sensor sinks are attached to gateways, which are typically a few orders of magnitude more powerful than the sensors: gateways run an embedded form of Linux, Windows, or other multitasking operating system. Gateways have an Internet connection, either through a cell phone network, long-distance wireless, or even just wired Ethernet.

Energy concerns dominate sensor hardware and software design. These nodes need to be wireless, small, low-cost, and operate unattended for long periods. While it is often possible to provide large power resources, such as large solar panels, periodic battery replacement, or wall power, to small numbers of gateways, doing so to every one of hundreds of sensors is infeasible.

1.1.1 Anatomy of a sensor node (mote)

Since energy consumption determines sensor node lifetime, sensor nodes, commonly referred to as *motes*, tend to have very limited computational and communication resources. Instead of a full-fledged 32-bit or 64-bit CPU with megabytes or gigabytes of RAM, they have 8-bit or 16-bit microcontrollers with a few kilobytes of RAM. Rather than gigahertz, these microcontrollers run at 1–10 megahertz. Their low-power radios

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Excerpt
More information

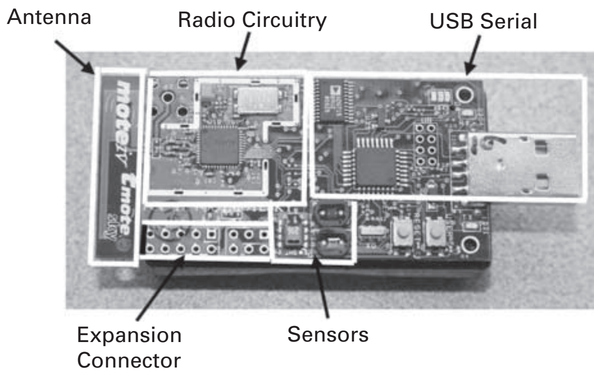

Figure 1.2 A Telos sensor produced by Moteiv. The top of the node has the radio, sensors, and circuitry for the USB connector. The bottom, not shown, has the processor and flash storage chip. The antenna is part of the printed circuit board (PCB).

can send tens to hundreds of kilobits per second (kbps), rather than 802.11's tens of megabits. As a result, software needs to be very efficient, both in terms of CPU cycles and in terms of memory use.

Figure 1.2 shows a sample node platform, the Telos , which is designed for easy experimentation and low-power operation. It has a TI MSP430 16-bit microcontroller with 10 kB of RAM and 48 kB of flash program memory. Its radio, a TI CC2420 which follows the IEEE 802.15.4 standard, can send up to 250 kbps. In terms of power, the radio dominates the system: on a pair of AA batteries, a Telos can have the radio on for about four days. Lasting longer than four days requires keeping the node in a deep sleep state most of the time, waking only when necessary, and sleeping as soon as possible.

The other mote discussed in this book, the micaz from Crossbow Technology is similar: it has an Atmel ATmega128 8-bit microcontroller with 4 kB of RAM, 128 kB of flash program memory, uses the same CC2420 radio chip, also runs off a pair of AA batteries and has a similar power consumption profile.

Networks, once deployed, gather data uninterrupted for weeks, months, or years. As the placement of sensors is very application-specific, it is rare for networks to need to support multiple concurrent applications, or even require more than the occasional reprogramming. Therefore, unlike general-purpose computing systems, which emphasize run-time flexibility and composability, sensor network systems tend to be highly optimized. Often, the sensor suite itself is selected for the specific application: volcanic monitoring uses accelerometers and microphones, while data center provisioning uses temperature sensors.

1.2 TinyOS

TinyOS is a lightweight operating system specifically designed for low-power wireless sensors. TinyOS differs from most other operating systems in that its design focuses on ultra low-power operation. Rather than a full-fledged processor, TinyOS is designed

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Excerpt
More information

for the small, low-power microcontrollers motes have. Furthermore, TinyOS has very aggressive systems and mechanisms for saving power.

TinyOS makes building sensor network applications easier. It provides a set of important services and abstractions, such as sensing, communication, storage, and timers. It defines a concurrent execution model, so developers can build applications out of reusable services and components without having to worry about unforeseen interactions. TinyOS runs on over a dozen generic platforms, most of which easily support adding new sensors. Furthermore, TinyOS's structure makes it reasonably easy to port to new platforms.

TinyOS applications and systems, as well as the OS itself, are written in the nesC language. nesC is a C dialect with features to reduce RAM and code size, enable significant optimizations, and help prevent low-level bugs like race conditions. Chapter 2 goes into the details on how nesC differs significantly from other C-like languages, and most of this book is about how to best use those features to write robust, efficient code.

1.2.1 What TinyOS provides

At a high level, TinyOS provides three things to make writing systems and applications easier:

- a component model, which defines how you write small, reusable pieces of code and compose them into larger abstractions;
- a concurrent execution model, which defines how components interleave their computations as well as how interrupt and non-interrupt code interact;
- application programming interfaces (APIs), services, component libraries and an overall component structure that simplify writing new applications and services.

The component model is grounded in nesC. It allows you to write pieces of reusable code which explicitly declare their dependencies. For example, a generic user button component that tells you when a button is pressed sits on top of an interrupt handler. The component model allows the button implementation to be independent of which interrupt that is – e.g. so it can be used on many different hardware platforms – without requiring complex callbacks or magic function naming conventions. Chapter 2 and Chapter 3 describe the basic component model.

The concurrent execution model enables TinyOS to support many components needing to act at the same time while requiring little RAM. First, every I/O call in TinyOS is *split-phase*: rather than block until completion, a request returns immediately and the caller gets a callback when the I/O completes. Since the stack isn't tied up waiting for I/O calls to complete, TinyOS only needs one stack, and doesn't have threads. Instead, Chapter 5 introduces *tasks*, which are lightweight deferred procedure calls. Any component can post a task, which TinyOS will run at some later time. Because low-power devices must spend most of their time asleep, they have low CPU utilization and so in practice tasks tend to run very soon after they are posted (within a few milliseconds). Furthermore, because tasks can't preempt each other, task code doesn't need to worry about data races. Low-level interrupt code (discussed in the advanced concurrency

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Excerpt
More information

chapter, Chapter 11) can have race conditions, of course: nesC detects possible data races at compile-time and warns you.

Finally, TinyOS itself has a set of APIs for common functionality, such as sending packets, reading sensors, and responding to events. Uses of these are sprinkled throughpout the entire book, and presented in more detail in Chapter 6 and Appendix 1. In addition to programming interfaces, TinyOS also provides a component structure and component libraries. For example, Chapter 12 describes TinyOS's Hardware Abstraction Architecture (HAA), which defines how to build up from low-level hardware (e.g. a radio chip) to a hardware-independent abstraction (e.g. sending packets). Part of this component structure includes resource locks, covered in Chapter 11, which enable automatic low-power operation, as well as the component libraries that simplify writing such locks.

TinyOS itself is continually evolving. Within the TinyOS community, "Working Groups" form to tackle engineering and design issues within the OS, improving existing services and adding new ones. This book is therefore really a snapshot of the OS in time. As Chapter 12 discusses and Appendix 1 presents, TinyOS has a set of standard, stable APIs for core abstractions, but this set is always expanding as new hardware and applications emerge. The best way to stay up to date with TinyOS is to check its web page `www.tinyos.net` and participate in its mailing lists. The website also covers advanced TinyOS and nesC features which are well beyond the scope of this book, including binary components, over-the-air reprogramming services, debugging tools, and a nesC reference manual.

1.3 Example application

To better understand the unique challenges faced by sensor networks, we walk through a basic data-collection application. Nodes running this application periodically wake up, sample some sensors, and send the data through an ad hoc collection tree to a data sink (as in Figure 1.1). As the network must last for a year, nodes spend 99% of their time in a deep sleep state.

In terms of energy, the radio is by far the most expensive part of the node. Lasting a year requires telling the radio to be in a low power state. Low power radio implementation techniques are beyond the scope of this book, but the practical upshot is that packet transmissions have higher latency. [23]

Figure 1.3 shows the four TinyOS APIs the application uses: low power settings for the radio, a timer, sensors, and a data collection routing layer. When TinyOS tells the application that the node has booted, the application code configures the power settings on the radio and starts a periodic timer. Every few minutes, this timer fires and the application code samples its sensors. It puts these sensor values into a packet and calls the routing layer to send the packet to a data sink. In practice, applications tend to be more complex than this simple example. For example, they include additional services such as a management layer which allows an administrator to reconfigure parameters and inspect the state of the network, as well as over-the-air programming so the network

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Excerpt
More information

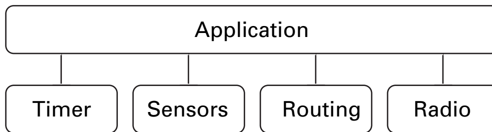

Figure 1.3 Example application architecture. Application code uses a timer to act periodically, sensors to collect data, and a routing layer to deliver data to a sink.

can be reprogrammed without needing to collect all of the nodes. However, these four abstractions – power control, timers, sensors, and data collection – encompass the entire datapath of the application.

1.4 Compiling and installing applications

You can download the latest TinyOS distribution, the nesC compiler, and other tools at `www.tinyos.net`. Setting up your programming environment is outside the scope of this book; the TinyOS website has step-by-step tutorials to get you started. One part of TinyOS is an extensive build system for compiling applications. Generally, to compile a program for a sensor platform, one types `make <platform>`, e.g. `make telosb`. This compiles a binary. To install that binary on a node, you plug the node into your PC using a USB or serial connection, and type `make <platform> install`. The tutorials go into compilation and installation options in detail.

1.5 The rest of this book

The rest of this book goes into how to program in nesC and write TinyOS applications. It is divided into three parts. The first is a short introduction to the major programming concepts of nesC. The second part addresses basic application programming using standard TinyOS APIs. The third part digs a little deeper, and looks into how those TinyOS APIs are implemented. For example, the third part describes how TinyOS abstracts hardware, so you can write a driver for a new sensor.

Chapter by chapter, the book is structured as follows:

- **Chapter 1** is this chapter.
- **Chapter 2** describes the major way that nesC breaks from C and C-like languages: how programs are built out of components, and how components and interfaces help manage programs' namespaces.
- **Chapter 3** presents components and how they interact via interfaces.
- **Chapter 4** goes into greater detail into configurations, components which connect other components together.
- **Chapter 5** covers the basic TinyOS execution model and gives guidance on how and when to use tasks.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Excerpt
More information

- **Chapter 6** takes the material from the prior three chapters and brings it together into an example of writing a fully-fledged application that sends an alarm when a node observes a change in its environment. In the process, it covers the major TinyOS APIs (timing, sensing, communication, and storage).
- **Chapter 7** details the PC-side tools for communicating with nodes connected over the serial port, and covers the TinyOS serial stack and packet formats.
- **Chapter 8** introduces more advanced component topics, such as parameterized interfaces and attributes. While applications typically don't use these mechanisms, they are indispensable when writing reusable libraries and systems.
- **Chapter 9** goes into wiring parameterized interfaces, which form the basis for most reusable systems. After describing the basic mechanisms for managing interface keys, it goes through four examples of increasing complexity.
- **Chapter 10** presents eight common TinyOS design patterns: commonly useful and powerful ways of organizing components.
- **Chapter 11** concludes the advanced programming topics by covering concurrency. It describes asynchronous code, the TinyOS task model, and power locks.
- **Chapter 12** describes the Hardware Abstraction Architecture (HAA), the three-level hierarchy TinyOS uses to raise low-level hardware abstractions to hardware-independent, application-level services.
- **Chapter 13** goes step-by-step through an advanced application that identifies the location of an event based on when nodes sense it. It uses lower-layer interfaces to implement time synchronization and high-frequency sampling.
- **Appendix** gives a concise overview of major TinyOS application interfaces.

Throughout the book, you will find *programming hints*: general best-use practices that we've distilled from the many users of TinyOS and nesC. These are all listed on the sheet at the front of the book.

Finally, the complete source code for example applications presented in this book (in Chapters 6, 7 and 13) is available in TinyOS's contributed code directory, under the name "TinyOS Programming" – see `www.tinyos.net` for details.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Excerpt
More information

2 Names and program structure

Program structure is the most essential and obvious difference between C and nesC. C programs are composed of variables, types, and functions defined in files that are compiled separately and then linked together. nesC programs are built out of components that are connected ("wired") by explicit program statements; the nesC compiler connects and compiles these components as a single unit. To illustrate and explain these differences in how programs are built, we compare and contrast C and nesC implementations of two very simple "hello world"-like mote applications, Powerup (boot and turn on a LED) and Blink (boot and repeatedly blink a LED).

2.1 Hello World!

The closest mote equivalent to the classic "Hello World!" program is the "Powerup" application that simply turns on one of the motes LEDs at boot, then goes to sleep.

A C implementation of Powerup is fairly simple:

```
#include "mote.h"

int main()
{
  mote_init();
  led0_on();
  sleep();
}
```

Listing 2.1 Powerup in C

The Powerup application is compiled and linked with a "mote" library which provides functions to perform hardware initialization (mote_init), LED control (led0_on) and put the mote in to a low-power sleep mode (sleep). The "mote.h" header file simply provides declarations of these and other basic functions. The usual C main function is called automatically when the mote boots.[1]

[1] The C compiler, library, and linker typically arrange for this by setting the mote's hardware reset vector to point to a piece of assembly code that sets up a C environment, then calls main.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Excerpt
More information

The nesC implementation of Powerup is split into two parts. The first, the PowerupC *module*, contains the executable logic of Powerup (what there is of it …):

```
module PowerupC {
  uses interface Boot;
  uses interface Leds;
}
implementation {
  event void Boot.booted() {
   call Leds.led0On();
  }
}
```

Listing 2.2 PowerupC module in nesC

This code says that PowerupC interacts with the rest of the system via two *interfaces*, Boot and Leds, and provides an implementation for the booted *event* of the Boot interface that calls the led0On[2] *command* of the Leds interface. Comparing with the C code, we can see that the booted event implementation takes the place of the main function, and the call to the led0On command the place of the call to the led0_on library function.

This code shows two of the major differences between nesC and C: where C programs are composed of functions, nesC programs are built out of *components* that implement a particular service (in the case of PowerupC, turning a LED on at boot-time). Furthermore, C functions typically interact by calling each other directly, while the interactions between components are specified by interfaces: the interface's *user* makes requests (*calls commands*) on the interface's *provider*, the provider makes callbacks (*signals events*) to the interface's user. Commands and events themselves are like regular functions (they can contain arbitrary C code); calling a command or signaling an event is just a function call. PowerupC is a user of both Boot and Leds; the booted event is a callback signaled when the system boots, while the led0On is a command requesting that LED 0 be turned on.

nesC interfaces are similar to Java interfaces, with the addition of a `command` or `event` keyword to distinguish requests from callbacks:

```
interface Boot} {
  event void} booted();
  }

interface Leds {
  command void led0On();
  command void led0Off();
```

[2] LEDs are numbered in TinyOS, as different platforms have different color LEDs.

© Cambridge University Press www.cambridge.org