TinyOS Programming

Do you need to know how to write systems, services, and applications using the TinyOS operating system? Learn how to write nesC code and efficient applications with this indispensable guide to TinyOS programming.

Detailed examples show you how to write TinyOS code in full, from basic applications right up to new low-level systems and high-performance applications. Two leading figures in the development of TinyOS also explain the reasons behind many of the design decisions made and explain for the first time how nesC relates to and differs from other C dialects. Handy features such as a library of software design patterns, programming hints and tips, end-of-chapter exercises, and an appendix summarizing the basic application-level TinyOS APIs make this the ultimate guide to TinyOS for embedded systems programmers, developers, designers, and graduate students.

Philip Levis is Assistant Professor of Computer Science and Electrical Engineering at Stanford University. A Fellow of the Microsoft Research Faculty, he is also Chair of the TinyOS Core Working Group and a Member of the TinyOS Network Protocol (net2), Simulation (sim), and Documentation (doc) Working Groups.

David Gay joined Intel Research in Berkeley in 2001, where he has been a designer and the principal implementer of the nesC language, the C dialect used to implement the TinyOS sensor network operating system, and its applications. He has a diploma in Computer Science from the Swiss Federal Institute of Technology in Lausanne and a Ph.D. from the University of California, Berkeley.

TinyOS Programming

PHILIP LEVIS
Stanford University

and

DAVID GAY
Intel Research

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo, Delhi

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521896061

First published 2009

Printed in the United Kingdom at the University Press, Cambridge

A catalogue record for this publication is available from the British Library

ISBN 978-0-521-89606-1 paperback

Contents

Code examples

Preface

This book provides an in-depth introduction to writing nesC code for the TinyOS 2.0 operating system. While it goes into greater depth than the TinyOS tutorials on this subject, there are several topics that are outside its scope, such as the structure and implementation of radio stacks or existing TinyOS libraries. It focuses on how to write nesC code, and explains the concepts and reasons behind many of the nesC and TinyOS design decisions. If you are interested in a brief introduction to TinyOS programming, then you should probably start with the tutorials. If you're interested in details on particular TinyOS subsystems you should probably consult TEPs (TinyOS Enhancement Proposals), which detail the corresponding design considerations, interfaces, and components. Both of these can be found in the `doc/html` directory of a TinyOS distribution.

While some of the contents of this book are useful for 1.x versions of TinyOS , they do have several differences from TinyOS 2.0 which can lead to different programming practices. If in doubt, referring to the TEP on the subject is probably the best bet, as TEPs often discuss in detail the differences between 1.x and 2.0.

For someone who has experience with C or C++, writing simple nesC programs is fairly straightforward: all you need to do is implement one or two modules and wire them together. The difficulty (and intellectual challenge) comes when building larger applications. The code inside TinyOS modules is fairly analogous to C coding, but configurations – which stitch together components – are not.

This book is a first attempt to explain how nesC relates to and differs from other C dialects, stepping through how the differences lead to very different coding styles and approaches. As a starting point, this book assumes that

1. you know C, C++, or Java reasonably well, understand pointers and that
2. you have taken an undergraduate level operating systems class (or equivalent) and know about concurrency, interrupts, and preemption.

Of course, this book is as much a description of nesC as it is an argument for a particular way of using the language to achieve software engineering goals. In this respect, it is the product of thousands of hours of work by many people, as they learned and explored the use of the language. In particular, Cory Sharp, Kevin Klues, and Vlado Handziski have always pushed the boundaries of nesC programming in order to better understand which practices lead to the simplest, most efficient, and robust code. In particular, Chapter 10

is an edited version of a paper we wrote together, while using structs as a compile-time checking mechanism in interfaces (as Timer does) is an approach invented by Cory.

This book is divided into four parts. The first part, Chapters 1–2, gives a high-level overview of TinyOS and the nesC language. The second part, Chapters 3–7 goes into nesC and TinyOS at a level sufficient for writing applications. The third part, Chapters 8–13 goes into more advanced TinyOS and nesC programming, as is sometimes needed when writing new low-level systems or high performance applications. The book ends with an appendix summarizing the basic application-level TinyOS APIs.

Acknowledgements

We'd like to thank several people for their contributions to this book. First is Mike Horton, of Crossbow, Inc., who first proposed writing it. Second is Pablo Guerrero, who gave detailed comments and corrections. Third is Joe Polastre of Moteiv, who gave valuable feedback on how to better introduce generic components. Fourth, we'd like to thank Phil's father, who although he doesn't program, read the entire thing! Fifth, John Regehr, Ben Greenstein and David Culler provided valuable feedback on this expanded edition. Last but not least, we would like to thank the TinyOS community and its developers. Many of the concepts in this book – power locks, tree routing, and interface type checking – are the work and ideas of others, which we merely present.

Chapter 10 of this book is based on: Software design patterns for TinyOS, in ACM Transactions on Embedded Computing Systems (TECS), Volume 6, Issue 4 (September 2007), ©ACM, 2007. http://doi.acm.org/10.1145/1274858.1274860

Programming hints, condensed

Programming Hint 1 Use the "as" keyword liberally. (page 24)

Programming Hint 2 Never write recursive functions within a module. In combination with the TinyOS coding conventions, this guarantees that all programs have bounded stack usage. (page 38)

Programming Hint 3 Never use malloc and free. Allocate all state in components. If your application requirements necessitate a dynamic memory pool, encapsulate it in a component and try to limit the set of users. (page 38)

Programming Hint 4 When possible, avoid passing pointers across interfaces; when this cannot be avoided only one component should be able to modify a pointer's data at any time. (page 39)

Programming Hint 5 Conserve memory by using enums rather than const variables for integer constants, and don't declare variables with an enum type. (page 42)

Programming Hint 6 Never, ever use the "packed" attribute in portable code. (page 43)

Programming Hint 7 Use platform-independent types when defining message structures. (page 44)

Programming Hint 8 If you have to perform significant computation on a platform-independent type or access it many (hundreds or more) times, temporarily copy it to a native type. (page 44)

Programming Hint 9 Interfaces should #include the header files for the types they use. (page 46)

Programming Hint 10 Always #define a preprocessor symbol in a header file. Use #include to load the header file in all components and interfaces that use the symbol. (page 47)

Programming Hint 11 If a component is a usable abstraction by itself, its name should end with C. If it is intended to be an internal and private part of a larger abstraction, its name should end with P. Never wire to P components from outside your package (directory). (page 58)

Programming Hint 12 Auto-wire Init to MainC in the top-level configuration of a software abstraction. (page 60)

Programming Hint 13 When using layered abstractions, components should not wire across multiple abstraction layers: they should wire to a single layer. (page 63)

Programming Hint 14 Never ignore combine warnings. (page 68)

Programming Hint 15 Keep tasks short. (page 74)

Programming Hint 16 If an event handler needs to make possibly long-executing command calls, post a task to make the calls. (page 75)

Programming Hint 17 Don't signal events from commands – the command should post a task that signals the event. (page 77)

Programming Hint 18 Use a parameterized interface when you need to distinguish callers or when you have a compile-time constant parameter. (page 141)

Programming Hint 19 If a component depends on unique, then #define the string to use in a header file, to prevent bugs due to string typos. (page 149)

Programming Hint 20 Whenever writing a module, consider making it more general-purpose and generic. In most cases, modules must be wrapped by configurations to be useful, so singleton modules have few advantages. (page 165)

Programming Hint 21 Keep code synchronous when you can. Code should be async only if its timing is very important or if it might be used by something whose timing is important. (page 195)

Programming Hint 22 Keep atomic statements short, and have as few of them as possible. Be careful about calling out to other components from within an atomic statement. (page 199)