

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.

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Frontmatter
[More information](#)

TinyOS Programming

PHILIP LEVIS
Stanford University

and

DAVID GAY
Intel Research



CAMBRIDGE
UNIVERSITY PRESS

Cambridge University Press
978-0-521-89606-1 - TinyOS Programming
Philip Levis and David Gay
Frontmatter
[More information](#)

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

© Cambridge University Press 2009

This publication is in copyright. Subject to statutory exception
and to the provisions of relevant collective licensing agreements,
no reproduction of any part may take place without
the written permission of Cambridge University Press.

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

Cambridge University Press has no responsibility for the persistence or
accuracy of URLs for external or third-party internet websites referred to
in this publication, and does not guarantee that any content on such
websites is, or will remain, accurate or appropriate.

Contents

	<i>List of Code examples</i>	page xi
	<i>Preface</i>	xvii
	<i>Acknowledgements</i>	xix
	<i>Programming hints, condensed</i>	xxi
	Part I TinyOS and nesC	1
1	Introduction	3
	1.1 Networked, embedded sensors	3
	1.1.1 Anatomy of a sensor node (mote)	4
	1.2 TinyOS	5
	1.2.1 What TinyOS provides	6
	1.3 Example application	7
	1.4 Compiling and installing applications	8
	1.5 The rest of this book	8
2	Names and program structure	10
	2.1 Hello World!	10
	2.2 Essential differences: components, interfaces, and wiring	13
	2.3 Wiring and callbacks	15
	2.4 Summary	16
	Part II Basic programming	19
3	Components and interfaces	21
	3.1 Component signatures	21
	3.1.1 Visualizing components	22
	3.1.2 The “as” keyword and clustering interfaces	23
	3.1.3 Clustering interfaces	24
	3.2 Interfaces	24
	3.2.1 Generic interfaces	27
	3.2.2 Bidirectional interfaces	28

vi	Contents	
		<hr/>
	3.3 Component implementations	29
	3.3.1 Modules	30
	3.3.2 A basic configuration	31
	3.3.3 Module variables	32
	3.3.4 Generic components	33
	3.4 Split-phase interfaces	34
	3.4.1 Read	36
	3.4.2 Send	36
	3.5 Module memory allocation, avoiding recursion, and other details	36
	3.5.1 Memory ownership and split-phase calls	38
	3.5.2 Constants and saving memory	41
	3.5.3 Platform-independent types	42
	3.5.4 Global names	44
	3.5.5 nesC and the C preprocessor	46
	3.5.6 C libraries	47
	3.6 Exercises	48
4	Configurations and wiring	49
	4.1 Configurations	50
	4.1.1 The \rightarrow and \leftarrow operators	51
	4.1.2 The = operator	52
	4.1.3 Namespace management	53
	4.1.4 Wiring rules	54
	4.1.5 Wiring shortcuts	56
	4.2 Building abstractions	57
	4.2.1 Component naming	58
	4.2.2 Component initialization	59
	4.3 Component layering	60
	4.3.1 Extensibility	61
	4.3.2 Hardware specificity	61
	4.4 Multiple wirings	63
	4.4.1 Fan-in and fan-out	64
	4.4.2 Uses of multiple wiring	65
	4.4.3 Combine functions	66
	4.5 Generics versus singletons	68
	4.5.1 Generic components, revisited	68
	4.5.2 Singleton components, revisited	70
	4.6 Exercises	70
5	Execution model	71
	5.1 Overview	71
	5.2 Tasks	72
	5.2.1 Task timing	74

	5.2.2 Timing and event handlers	75
5.3	Tasks and split-phase calls	75
	5.3.1 Hardware versus software	75
	5.3.2 Tasks and call loops	76
5.4	Exercises	78
6	Applications	79
6.1	The basics: timing, LEDs, and booting	79
	6.1.1 Deadline-based timing	81
	6.1.2 Wiring AntiTheftC	83
6.2	Sensing	83
	6.2.1 Simple sampling	84
	6.2.2 Sensor components	85
	6.2.3 Sensor values, calibration	86
	6.2.4 Stream sampling	87
6.3	Single-hop networking	89
	6.3.1 Sending packets	90
	6.3.2 Receiving packets	93
	6.3.3 Selecting a communication stack	94
6.4	Multi-hop networking: collection, dissemination, and base stations	95
	6.4.1 Collection	96
	6.4.2 Dissemination	97
	6.4.3 Wiring collection and dissemination	97
	6.4.4 Base station for collection and dissemination	98
6.5	Storage	101
	6.5.1 Volumes	102
	6.5.2 Configuration data	103
	6.5.3 Block and Log storage	105
6.6	Exercises	111
7	Mote-PC communication	112
7.1	Basics	112
	7.1.1 Serial communication stack	113
7.2	Using mig	114
	7.2.1 Sending and receiving mig-generated packets	116
7.3	Using ncg	118
7.4	Packet sources	119
7.5	Example: simple reliable transmission	120
	7.5.1 Reliable transmission protocol	121
	7.5.2 Reliable transmission in Java	121
	7.5.3 Reimplementing TestSerial	125
7.6	Exercises	125

Part III	Advanced programming	127
8	Advanced components	129
	8.1 Generic components review	129
	8.2 Writing generic modules	131
	8.2.1 Type arguments	132
	8.2.2 Abstract data types as generics	133
	8.2.3 ADTs in TinyOS 1.x	134
	8.3 Parameterized interfaces	135
	8.3.1 Parameterized interfaces and configurations	137
	8.3.2 Parameterized interfaces and modules	139
	8.3.3 Defaults	141
	8.4 Attributes	142
	8.5 Exercises	144
9	Advanced wiring	145
	9.1 unique() and uniqueCount()	145
	9.1.1 unique	146
	9.1.2 uniqueCount	147
	9.1.3 Example: HilTimerMilliC and VirtualizeTimerC	147
	9.2 Generic configurations	150
	9.2.1 TimerMilliC	150
	9.2.2 CC2420SpiC	152
	9.2.3 AMSEnderC	156
	9.2.4 BlockStorageC	160
	9.3 Reusable component libraries	162
	9.4 Exercises	165
10	Design patterns	166
	10.1 Behavioral: Dispatcher	166
	10.2 Structural: Service Instance	170
	10.3 Namespace: Keyspace	174
	10.4 Namespace: Keymap	177
	10.5 Structural: Placeholder	180
	10.6 Structural: Facade	183
	10.7 Behavioral: Decorator	186
	10.8 Behavioral: Adapter	189
11	Concurrency	192
	11.1 Asynchronous code	192
	11.1.1 The async keyword	192
	11.1.2 The cost of async	193
	11.1.3 Atomic statements and the atomic keyword	195

	11.1.4 Managing state transitions	197
	11.1.5 Example: CC2420ControlP	197
	11.1.6 Tasks, revisited	199
	11.2 Power locks	200
	11.2.1 Example lock need: link-layer acknowledgements	200
	11.2.2 Split-phase locks	201
	11.2.3 Lock internals	202
	11.2.4 Energy management	203
	11.2.5 Hardware configuration	204
	11.2.6 Example: MSP430 USART	204
	11.2.7 Power lock library	205
	11.3 Exercises	205
12	Device drivers and the hardware abstraction architecture (HAA)	206
	12.1 Portability and the hardware abstraction architecture	206
	12.1.1 Examples	208
	12.1.2 Portability	210
	12.2 Device drivers	210
	12.2.1 Access control	211
	12.2.2 Access control examples	212
	12.2.3 Power management	215
	12.2.4 Microcontroller power management	218
	12.3 Fitting in to the HAA	219
13	Advanced application: SoundLocalizer	221
	13.1 SoundLocalizer design	221
	13.1.1 Time synchronization	222
	13.1.2 Implementing SoundLocalizer in TinyOS	223
	13.2 SynchronizerC	225
	13.3 DetectorC	230
	13.4 MicrophoneC	233
	13.5 Wrap-up	237
	Part IV Appendix and references	239
A	TinyOS APIs	241
	A.1 Booting	241
	A.2 Communication	241
	A.2.1 Single-hop	242
	A.2.2 Multi-hop collection	243
	A.2.3 Multi-hop dissemination	244
	A.2.4 Binary reprogramming	245
	A.3 Time	245
	A.4 Sensing	245

A.5	Storage	246
A.6	Data structures	247
A.6.1	BitVectorC	247
A.6.2	QueueC	247
A.6.3	BigQueueC	248
A.6.4	PoolC	248
A.6.5	StateC	249
A.7	Utilities	249
A.7.1	Random numbers	249
A.7.2	Leds	249
A.7.3	Cyclic redundancy checks	250
A.7.4	Printf	250
A.8	Low power	251
	<i>References</i>	252
	<i>Index</i>	254

Code examples

2.1	Powerup in C	<i>page</i> 10
2.2	PowerupC module in nesC	11
2.3	Simple nesC interfaces	11
2.4	PowerupAppC configuration in nesC	12
2.5	Powerup with blinking LED in C	15
2.6	Powerup with blinking LED in nesC (slightly simplified)	15
2.7	Powerup with blinking LED configuration (slightly simplified)	16
3.1	The signature and implementation blocks	21
3.2	Signatures of PowerupC and LedsC	22
3.3	MainC's signature	22
3.4	The LedsP module	23
3.5	PowerupC and an alternative signature	24
3.6	Interface declarations for Leds and Boot	25
3.7	The Init and Boot interfaces	25
3.8	Signatures of MainC and PowerupC	26
3.9	The Queue interface	27
3.10	Using a queue of 32-bit integers	27
3.11	Providing a 16-bit or a 32-bit queue	27
3.12	The Notify interface	28
3.13	UserButtonC	28
3.14	Simplified Timer interface showing three commands and one event	29
3.15	PowerupC module code	30
3.16	The module PowerupToggleC	30
3.17	The PowerupToggleAppC configuration	31
3.18	Example uses of the components keyword	31
3.19	The Get interface	32
3.20	A self-incrementing counter	32
3.21	Generic module SineSensorC and generic configuration TimerMilliC	33
3.22	Instantiating a generic component	34
3.23	Signature of BitVectorC	34
3.24	QueueC signature	34
3.25	The Read interface	36
3.26	The split-phase Send interface	36
3.27	The Send interface	38

3.28	The Receive interface	39
3.29	The signature of PoolC	41
3.30	CC2420 packet header	42
3.31	The dreaded “packed” attribute in the 1.x MintRoute library	43
3.32	The CC2420 header	44
3.33	TinyError.h, a typical nesC header file	45
3.34	Including a header file in a component	45
3.35	Indirectly including a header file	46
3.36	Fancy.nc: C preprocessor example	46
3.37	FancyModule.nc: C preprocessor pitfalls	47
3.38	Fancy.h: the reliable way to use C preprocessor symbols	47
3.39	Using a C library function	47
4.1	Signature of part of the CC1000 radio stack	49
4.2	The PowerupToggleAppC configuration revisited	51
4.3	C code generated from the PowerupToggleAppC configuration	51
4.4	The LedsC configuration	52
4.5	CC2420ReceiveC’s use of the as keyword	53
4.6	Naming generic component instances	54
4.7	MainC and LedsP	55
4.8	Valid alternate of PowerupToggleAppC	55
4.9	Invalid alternate of PowerupToggleAppC	55
4.10	LedsC revisited	56
4.11	BlinkC signature	56
4.12	The RandomC configuration	57
4.13	The RandomMlcgC signature	58
4.14	Seed initialization in RandomMlcgP	59
4.15	ActiveMessageC for the CC2420	61
4.16	The signature of CC2420ActiveMessageC	62
4.17	Fan-out on CC2420TransmitC’s Init	63
4.18	StdControl and SplitControl initialization interfaces	64
4.19	Why the metaphor of “wires” is only a metaphor	65
4.20	The combine function for error_t	66
4.21	Fan-out on SoftwareInit	67
4.22	Resulting code from fan-out on SoftwareInit	67
4.23	AMSenderC signature	68
4.24	RadioCountToLedsAppC	68
4.25	PoolC	69
4.26	Exposing a generic component instance as a singleton	70
5.1	The main TinyOS scheduling loop from SchedulerBasicP.nc	72
5.2	A troublesome implementation of a magnetometer sensor	76
5.3	Signal handler that can lead to an infinite loop	77
5.4	An improved implementation of FilterMagC	77
6.1	Anti-theft: simple flashing LED	80
6.2	The Leds interface	80

6.3	The Boot interface	81
6.4	The full Timer interface	81
6.5	WarningTimer.fired with drift problem fixed	82
6.6	Anti-Theft: application-level configuration	83
6.7	The Read interface	84
6.8	Anti-theft: detecting dark conditions	84
6.9	Anti-Theft: wiring to light sensor	86
6.10	ReadStream Interface	87
6.11	Anti-theft: detecting movement	88
6.12	The AMSend interface	90
6.13	Anti-Theft: reporting theft over the radio	91
6.14	The SplitControl interface	92
6.15	The Receive interface	93
6.16	Anti-Theft: changing settings	93
6.17	Serial vs Radio-based AM components	94
6.18	The Send interface	96
6.19	Anti-Theft: reporting theft over a collection tree	96
6.20	DisseminationValue interface	97
6.21	Anti-Theft: settings via a dissemination tree	97
6.22	The StdControl interface	97
6.23	The DisseminationUpdate interface	99
6.24	AntiTheft base station code: disseminating settings	99
6.25	The RootControl interface	100
6.26	AntiTheft base station code: reporting thefts	101
6.27	AntiTheft base station wiring	101
6.28	ConfigStorageC signature	102
6.29	Mount interface for storage volumes	103
6.30	ConfigStorage interface	103
6.31	Anti-Theft: reading settings at boot time	104
6.32	Anti-Theft: saving configuration data	105
6.33	BlockStorageC signature	106
6.34	The BlockWrite interface	106
6.35	Simultaneously sampling and storing to flash (most error checking omitted)	108
6.36	The BlockRead interface	108
6.37	LogStorageC signature	108
6.38	The LogWrite interface	109
6.39	The LogWrite interface	110
6.40	The LogRead interface	111
7.1	Serial AM Packet layout	113
7.2	TestSerial packet layout	114
7.3	Backing array methods	115
7.4	Sending packets with mig and MoteIF	117
7.5	Interface for handling received packets	117

7.6	Receiving packets with mig and MoteIF	117
7.7	Constants and packet layout for Oscilloscope application	118
7.8	Class generated by ncg	119
7.9	Simplified code to save received samples	119
7.10	Reliable transmission protocol in Java – transmission	121
7.11	Reliable transmission protocol in Java – transmission	123
7.12	A reliable TestSerial.java	125
8.1	Instantiation within a generic configuration	130
8.2	The fictional component SystemServiceVectorC	131
8.3	QueueC excerpt	131
8.4	A generic constant sensor	132
8.5	Queue interface (repeated)	133
8.6	QueueC implementation	133
8.7	Representing an ADT through an interface in TinyOS 1.x	135
8.8	Timers without parameterized interfaces	135
8.9	Timers with a single interface	136
8.10	HilTimerMilliC signature	137
8.11	ActiveMessageC signature	138
8.12	Signature of TestAMC	138
8.13	Wiring TestAMC to ActiveMessageC	138
8.14	A possible module underneath ActiveMessageC	139
8.15	Parameterized interface syntax	140
8.16	Dispatching on a parameterized interface	140
8.17	How active message implementations decide on whether to signal to Receive or Snoop	140
8.18	Defining a parameter	141
8.19	Wiring full parameterized interface sets	141
8.20	Default events in an active message implementation	142
8.21	nesC attributes	143
9.1	Partial HilTimerMilliC signature	146
9.2	VirtualizeTimerC	148
9.3	Instantiating VirtualizeTimerC	148
9.4	VirtualizeTimerC state allocation	149
9.5	The TimerMilliC generic configuration	151
9.6	TimerMilliP auto-wires HilTimerMilliC to Main.SoftwareInit	151
9.7	The Blink application	151
9.8	The full module-to-module wiring chain in Blink (BlinkC to VirtualizeTimerC)	152
9.9	CC2420SpiC	153
9.10	CC2420SpiP	154
9.11	CC2420SpiC mappings to CC2420SpiP	154
9.12	The strobe implementation	155
9.13	The AMSenderC generic configuration	158
9.14	AMSendQueueEntryP	159

9.15	AMQueueP	159
9.16	AMSendQueueImplP pseudocode	160
9.17	BlockStorageC	161
9.18	The full code of HilTimerMilliC	163
9.19	VirtualizeTimerC virtualizes a single timer	164
10.1	AMReceiverC	169
10.2	VirtualizeTimerC	172
10.3	Telos ActiveMessageC	181
10.4	The Matchbox facade	184
10.5	The CC2420Csmac uses a Facade	185
10.6	AlarmToTimerC implementation	190
11.1	The Send interface	192
11.2	The Leds interface	193
11.3	Toggling a state variable	193
11.4	A call sequence that could corrupt a variable	194
11.5	State transition that is not async-safe	194
11.6	Incrementing with an atomic statement	195
11.7	Incrementing with two independent atomic statements	195
11.8	The first step of starting the CC2420 radio	198
11.9	The handler that the first step of starting the CC2420 is complete	198
11.10	The handler that the second step of starting the CC2420 is complete	198
11.11	The handler that the third step of starting the CC2420 radio is complete	199
11.12	State transition so components can send and receive packets	199
11.13	The Resource interface	201
11.14	Msp430Spi0C signature	202
11.15	Msp320Adc12ClientC signature	202
11.16	The ResourceDefaultOwner interface	203
11.17	The ResourceConfigure interface	204
12.1	ActiveMessageC signature	212
12.2	Arbitration in Stm25pSectorC	215
12.3	McuSleepC: platform-specific sleep code	218
13.1	SynchronizerC: time synchronization for SoundLocalizer	225
13.2	The Counter interface	226
13.3	DetectorC: loud sound detection for SoundLocalizer	231
13.4	The Alarm interface	231
13.5	Atm128AdcSingle: low-level single-sample ATmega128 A/D converter interface	232
13.6	The GeneralIO digital I/O pin interface	235
13.7	The I2Cpacket interface for bus masters	236

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)