

BLUETOOTH APPLICATION PROGRAMMING

WITH THE **JAVA APIS**



C Bala Kumar Paul J. Kline Timothy J. Thompson

BLUETOOTH® APPLICATION PROGRAMMING WITH THE JAVA™ APIs

The Morgan Kaufmann Series in Networking

Series Editor: David Clark, M.I.T.

Bluetooth Application Programming with the Java APIs
C Bala Kumar, Paul J. Kline, and Timothy J. Thompson

Modern Cable Television Technology: Video, Voice, and Data Communications, Second Edition
Walter Ciciora, James Farmer, David Large, and Michael Adams

Policy-Based Network Management: Solutions for the Next Generation
John Strassner

Computer Networks: A Systems Approach, Third Edition
Larry L. Peterson and Bruce S. Davie

Network Architecture, Analysis, and Design, Second Edition
James D. McCabe

MPLS Network Management: MIBs, Tools, and Techniques
Thomas D. Nadeau

Developing IP-Based Services: Solutions for Service Providers and Vendors
Monique Morrow and Kateel Vijayananda

Telecommunications Law in the Internet Age
Sharon K. Black

Optical Networks: A Practical Perspective, Second Edition
Rajiv Ramaswami and Kumar N. Sivarajan

Internet QoS: Architectures and Mechanisms
Zheng Wang

TCP/IP Sockets in Java: Practical Guide for Programmers
Michael J. Donahoo and Kenneth L. Calvert

TCP/IP Sockets in C: Practical Guide for Programmers
Kenneth L. Calvert and Michael J. Donahoo

Multicast Communication: Protocols, Programming, and Applications
Ralph Wittmann and Martina Zitterbart

MPLS: Technology and Applications
Bruce Davie and Yakov Rekhter

High-Performance Communication Networks, Second Edition
Jean Walrand and Pravin Varaiya

Internetworking Multimedia
Jon Crowcroft, Mark Handley, and Ian Wakeman

Understanding Networked Applications: A First Course
David G. Messerschmitt

Integrated Management of Networked Systems: Concepts, Architectures, and their Operational Application
Heinz-Gerd Hegering, Sebastian Abeck, and Bernhard Neumair

Virtual Private Networks: Making the Right Connection
Dennis Fowler

Networked Applications: A Guide to the New Computing Infrastructure
David G. Messerschmitt

Switching in IP Networks: IP Switching, Tag Switching, and Related Technologies
Bruce S. Davie, Paul Doolan, and Yakov Rekhter

Wide Area Network Design: Concepts and Tools for Optimization
Robert S. Cahn

*For further information on these books and for a list of forthcoming titles,
please visit our website at <http://www.mkp.com>*

BLUETOOTH® APPLICATION PROGRAMMING WITH THE JAVA™ APIs

C BALA KUMAR

PAUL J. KLINE

TIMOTHY J. THOMPSON

MOTOROLA SEMICONDUCTOR PRODUCTS SECTOR



MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER SCIENCE

AMSTERDAM BOSTON LONDON NEW YORK
OXFORD PARIS SAN DIEGO SAN FRANCISCO
SINGAPORE SYDNEY TOKYO

<i>Acquisitions Editor:</i>	Rick Adams
<i>Associate Editor:</i>	Karyn P. Johnson
<i>Publishing Services Manager:</i>	Simon G. Crump
<i>Project Manager:</i>	Mamata N. Reddy
<i>Designer:</i>	Eric DeCicco
<i>Production Services:</i>	Graphic World Publishing Services
<i>Composition:</i>	Expo
<i>Illustration:</i>	Dartmouth Publishing
<i>Printer:</i>	The Maple-Vail Book Manufacturing Group
<i>Cover Printer:</i>	Phoenix Color

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK. Telephone: (+44) 1865 843830; FAX: (+44) 1865 853333; E-mail: permissions@elsevier.com.uk. You may also complete your request online via the Elsevier homepage (www.elsevier.com), by selecting "Customer Support" and then "Obtaining Permissions."

Morgan Kaufmann Publishers
An imprint of Elsevier Inc.
500 Sansome Street, Suite 400
San Francisco, CA 94111
www.mkp.com

© 2004 by Elsevier Inc.
All rights reserved
Printed in the United States of America

08 07 06 05 04 5 4 3 2 1

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher.

MOTOROLA, the Stylized M Logo and all other trademarks indicated as such herein are trademarks of Motorola, Inc. registered in the U.S. Patent and Trademark Office. The Bluetooth trademark is owned by its proprietor and used by Motorola, Inc. under license. Java and all other Java-based terms are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All other product or service names are the property of their respective owners. © 2003 Motorola, Inc. All rights reserved.

Library of Congress Control Number: 2003107476
International Standard Book Number: 1-55860-934-2

This book is printed on acid-free paper.

To my wife, Sundari, and sons, Sailesh and Shiva
—Bala

To my wife, Dianne
—Paul

To my wife, Karmen
—Tim

This Page Intentionally Left Blank

Contents

LIST OF TABLES	xiii
LIST OF FIGURES	xv
PREFACE	xvii
ACKNOWLEDGMENTS	xxi

CHAPTER 1	Introduction.	1
	1.1 Wireless Connectivity	1
	1.2 What is Bluetooth Wireless Technology?	3
	1.2.1 History of Bluetooth Wireless Technology	4
	1.2.2 Bluetooth Vision	5
	1.2.3 Bluetooth Specification	7
	1.3 Overview of the Bluetooth Stack Architecture.	8
	1.3.1 Bluetooth Protocols.	9
	1.3.2 Bluetooth Profiles	11
	1.3.3 Bluetooth Qualification	13
	1.4 What is J2ME?	14
	1.4.1 Configurations	15
	1.4.2 Profiles	16
	1.4.3 Optional Packages.	18
	1.5 Why Java Technology for Bluetooth Devices?	18
	1.5.1 Java Community Process and JSR-82.	19
	1.5.2 What about J2SE and PersonalJava?	20
	1.5.3 Jini, JXTA, and Bluetooth Networks.	20
	1.6 Summary.	22
CHAPTER 2	An Overview of JABWT	23
	2.1 Goals	23
	2.1.1 Target Devices	24
	2.1.2 Keeping Up with the Bluetooth Profiles	24

	2.1.3 JABWT Use Cases.	26
	2.2 API Characteristics and Hardware Requirements.	27
	2.2.1 JABWT Specification Characteristics	27
	2.2.2 J2ME Device Requirements	28
	2.2.3 Bluetooth System Requirements.	29
	2.3 Scope	30
	2.4 Summary.	33
CHAPTER 3	High-Level Architecture	35
	3.1 Architecture of JABWT.	35
	3.1.1 CLDC, MIDP, and JABWT	35
	3.1.2 Java Packages	37
	3.1.3 Client and Server Model	37
	3.1.4 Device Properties	41
	3.2 Bluetooth Control Center	41
	3.2.1 Conflict Resolution	42
	3.2.2 Modifying Device Properties	43
	3.2.3 User Interaction.	43
	3.2.4 BCC on Devices with No User Interface	43
	3.3 Simple JABWT Application	43
	3.4 Summary.	50
CHAPTER 4	RFCOMM	51
	4.1 Overview.	51
	4.2 API Capabilities	56
	4.3 Programming with the API	58
	4.3.1 Establishing a Server Connection	60
	4.3.2 Establishing a Client Connection	65
	4.4 Summary.	75
CHAPTER 5	OBEX	77
	5.1 Overview.	77
	5.1.1 Use Cases	78
	5.1.2 Protocol Description.	79
	5.1.3 Example Session.	81
	5.2 API Capabilities	84
	5.3 Programming with the API	87
	5.3.1 Establishing a Connection	88
	5.3.2 Manipulating OBEX Headers	90

	5.3.3 Sending a Request to the Server	92
	5.3.4 Receiving a Request from a Client	98
	5.3.5 Using OBEX Authentication	102
	5.4 Summary	107
CHAPTER 6	Device Discovery	109
	6.1 Overview	109
	6.2 API Capabilities.	111
	6.3 Programming with the API.	113
	6.3.1 Accessing the Local Device	114
	6.3.2 Device Discovery via Inquiry	125
	6.3.3 Retrieving Information from a Remote Device.	132
	6.3.4 Using the DeviceClass Class	135
	6.4 Summary	138
CHAPTER 7	Service Discovery.	139
	7.1 Overview	139
	7.1.1 Key Concepts	141
	7.1.2 Services Delivered by Bluetooth Profiles	142
	7.1.3 Custom Services	142
	7.2 API Capabilities.	142
	7.2.1 Run-before-Connect Services	143
	7.2.2 Servers Are Usually Discoverable	144
	7.2.3 Register a Service	146
	7.2.4 Modifications to Service Records	158
	7.2.5 Device Service Classes	160
	7.3 Programming with the API.	161
	7.3.1 Automatic Generation of Service Records	161
	7.3.2 Modifying a Service Record	167
	7.3.3 Support for String Attributes in Several Languages	170
	7.3.4 Service Records for Bluetooth Profiles	173
	7.3.5 Service Discovery	181
	7.3.6 Working with Service Records	191
	7.3.7 Retrieving Additional Attributes after Service Discovery	195
	7.3.8 Simple Device and Service Discovery	199
	7.3.9 Connect-Anytime Services	202
	7.4 Summary	204

CHAPTER 8	L2CAP	205
	8.1 Overview	205
	8.1.1 L2CAP Channels and L2CAP Packets	207
	8.1.2 Reasons for Using L2CAP	208
	8.2 API Capabilities	211
	8.2.1 JABWT for L2CAP	211
	8.2.2 Open an L2CAP Connection	212
	8.2.3 L2CAP Channel Configuration	215
	8.2.4 No Flow Control in L2CAP	218
	8.2.5 Types of Applications Using L2CAP	220
	8.3 Programming with the API	221
	8.3.1 Example: Echo L2CAP Packets	221
	8.3.2 User Interface for MTU Values	230
	8.3.3 L2CAP Clients and Servers Have the Same Capabilities	233
	8.3.4 Flow Control	234
	8.4 Summary	242
CHAPTER 9	Example Applications	243
	9.1 Overview	243
	9.2 Tic-Tac-Toe MIDlet	244
	9.2.1 Defining the TicTacToeMIDlet	244
	9.2.2 The PauseCanvas and PauseTimerTask Helper Classes	251
	9.2.3 Creating the Game Thread	257
	9.2.4 Playing the Game	267
	9.3 OBEX Application Download	288
	9.3.1 The Message Server Application	289
	9.3.2 The Message Client Application	301
	9.4 Summary	318
CHAPTER 10	Implementing JABWT on a Device	319
	10.1 Porting Process	320
	10.2 Steps 1 and 2: Adding J2ME and Bluetooth Support ...	322
	10.3 Step 3: Implementing JABWT	325
	10.3.1 KVM-Stack Interface	327
	10.3.2 Implementing the BCC	330
	10.3.3 OBEX Implementation	331

	10.4 Step 4: TCK Compliance	332
	10.5 Summary	334
CHAPTER 11	Closing Remarks	337
APPENDIX A	Complete Code Examples	341
APPENDIX B	javax.bluetooth API	391
APPENDIX C	javax.obex API	453
REFERENCES		491
INDEX.		493

This Page Intentionally Left Blank

List of Tables

Table 1.1	Comparison of Wireless Communication Technologies	3
Table 4.1	Valid Parameters for RFCOMM Connection Strings	59
Table 5.1	OBEX Header Constants in the HeaderSet Interface, Their Meaning, and Their Type	91
Table 5.2	How OBEX Requests Are Translated to Methods by the JSR-82 Implementation	99
Table 6.1	Properties Available through LocalDevice.getProperty()	116
Table 6.2	Major Service Classes Defined by the Bluetooth SIG	136
Table 6.3	Major Device Classes Defined by the Bluetooth SIG.	137
Table 7.1	Service Record Created by Connector.open("btspp: //localhost:...")	148
Table 7.2	Some Service Record Attribute IDs Defined by Bluetooth Assigned Numbers.	150
Table 7.3	Examples of Conversions from a 16-Bit UUID to a 128-Bit UUID	151
Table 7.4	Service Record Created by Connector.open("bt12cap: //localhost:...")	154
Table 7.5	Service Record Created by Connector.open("btgoep: //localhost:...")	156
Table 7.6	Methods That Create a Service Record.	157
Table 7.7	Methods That Add Service Records to the SDDB.	158
Table 7.8	Methods That Remove Service Records from the SDDB	158
Table 7.9	Service Record for a Bluetooth Game.	168
Table 7.10	Selected Attributes from a Service Record with English and French Strings	172
Table 7.11	Service Record Defined by Bluetooth Object Push Profile	174
Table 7.12	Service Record for an OBEX Object Push Server.	176
Table 7.13	Status Codes for Service Searches	183
Table 7.14	Bluetooth Data Element Types and Their Associated Java Types.	192
Table 8.1	Protocols and Profiles above Line Are Candidates for L2CAP API.	209

This Page Intentionally Left Blank

List of Figures

Figure 1.1	Bluetooth SIG timeline	5
Figure 1.2	Bluetooth use cases	6
Figure 1.3	Bluetooth host and device classification.	8
Figure 1.4	Bluetooth protocol stack.	9
Figure 1.5	Bluetooth profile hierarchy	13
Figure 1.6	Java 2 platforms	15
Figure 1.7	Components of J2ME architecture.	16
Figure 1.8	Jini and JXTA on a Bluetooth device with JABWT.	21
Figure 2.1	Target devices for JABWT	25
Figure 2.2	JABWT provides access to selected functionality of the Bluetooth stack	32
Figure 3.1	CLDC + MIDP + Bluetooth architecture diagram	36
Figure 3.2	Package structure	37
Figure 3.3	Bluetooth components involved in service discovery	38
Figure 3.4	How the BCC fits into a JABWT implementation.	42
Figure 3.5	Run using the Impronto Simulator. (a) HelloServer. (b) HelloClient	49
Figure 4.1	Bluetooth profiles defined by the Bluetooth SIG [3]	52
Figure 4.2	Multiple Bluetooth connections can exist over a single Bluetooth link	53
Figure 4.3	For two devices to complete the pairing process, a common PIN must be entered	54
Figure 4.4	Device A attempts to authenticate device B.	54
Figure 4.5	Example of encryption	55
Figure 4.6	GCF defined by CLDC	57
Figure 4.7	Master connects to slave	58
Figure 4.8	EchoServer MIDlet when it starts	67
Figure 4.9	EchoClient (a) and EchoServer (b) communicating over RFCOMM.	75
Figure 5.1	OBEX CONNECT operation	81
Figure 5.2	OBEX SETPATH operation.	82

Figure 5.3	OBEX PUT operation	83
Figure 5.4	OBEX DISCONNECT operation	83
Figure 5.5	OBEX client API results from combination of two connection types	85
Figure 5.6	OBEX server API created by combining two well-known APIs	86
Figure 5.7	GCF with OBEX API	87
Figure 5.8	PUT request that combines <code>OutputStream</code> and <code>HeaderSet</code> into an OBEX packet	94
Figure 5.9	Phases of a PUT request	95
Figure 5.10	Phases of a GET operation	97
Figure 6.1	Cell phone performs a general inquiry.	110
Figure 6.2	All general and limited discoverable devices respond to a general inquiry	111
Figure 6.3	<code>BluetoothInfoMIDlet</code> running in J2ME Wireless Toolkit.	121
Figure 6.4	<code>DiscoveryMIDlet</code> retrieving list of preknown and cached devices. . .	123
Figure 6.5	<code>DiscoveryMIDlet</code> after discovering devices via inquiry	130
Figure 7.1	Unlike an inquiry, a service search is between only two devices	140
Figure 7.2	Life cycle of a service record for a run-before-connect service.	145
Figure 7.3	Example of server modifying its service record	159
Figure 7.4	Example code displays information about the default service record. . .	162
Figure 7.5	Lifecycle of a service record for a connect-anytime service.	203
Figure 8.1	Position of L2CAP in the Bluetooth stack	206
Figure 8.2	L2CAP channels transmit L2CAP packets to multiple destinations	207
Figure 8.3	JABWT defines two interfaces for L2CAP communications	213
Figure 8.4	User interface for MTU experiments. (a) A <code>receiveMTU</code> of 672 bytes is entered for the server. (b) A <code>transmitMTU</code> of 500 bytes is entered for the client.	231
Figure 9.1	First Screen of Tic-Tac-Toe MIDlet	248
Figure 9.2	Screenshot of <code>TTTCanvas</code>	271
Figure 10.1	Components of JABWT device.	321
Figure 10.2	Components of native Bluetooth device.	322
Figure 10.3	Components of MIDP device	323
Figure 10.4	Device with MIDP and a native Bluetooth stack.	323
Figure 10.5	Implementation components	326
Figure 10.6	Components of KVM extensions	327
Figure 10.7	Context-switching in J2ME.	329
Figure 10.8	Not blocking the KVM	330
Figure 10.9	OBEX implementations	332
Figure 10.10	JSR-82 TCK setup	334

Preface

Bluetooth® wireless technology is a short-range radio standard that provides new opportunities for wireless devices. This radio standard was designed originally as a way of eliminating the cables currently attached to nearly all consumer electronic devices. However, the goals for Bluetooth wireless technology grew as its designers recognized that it enables a new kind of wireless network between electronic devices.

Recent developments in the Java™ programming language make it possible for Java developers to write applications for wireless devices such as cell phones and personal digital assistants. In 2000, the Java programming community recognized the importance of creating a standard extension to the Java programming language for use with Bluetooth devices. A standard application programming interface (API) for Bluetooth was needed because each Bluetooth software protocol stack had its own API for application programmers. These proprietary APIs meant that a Bluetooth application had to be ported to different Bluetooth stacks to run on different devices. Apart from the work involved in writing the code, interoperability testing on the various devices cost time and money for the involved companies. A standard API would help alleviate all these problems.

A team of experts from across the industry was assembled for this effort under Java Specification Request 82 (JSR-82). The result was a specification for Java APIs for Bluetooth wireless technology (JABWT). The specification was the result of collaboration of many companies and individuals in the industry. Some helped define the specification by participating in the JSR-82 expert group, and many others contributed by providing valuable comments.

This book introduces and explains to the community of Java programmers the new standard extensions of the Java programming language for Bluetooth wireless technology.

Specification documents provide you with the API to which you are programming. But a book like this one explains why an API is needed and how to use it. In addition, because we were members of the Motorola team leading the JSR-82 standardization effort, we believe we can explain the rationale for various features of JABWT.

The objectives of this book are to

- Give an overview of Java 2 Platform, Micro Edition (J2ME™) and Bluetooth wireless technology
- Outline the JABWT architecture
- Explain the API in detail
- Provide example applications
- Present the issues related to implementing JABWT on a device

Intended Audience

The book is intended for software developers, academics, and other professionals who want to develop Java software for Bluetooth devices. The book also is aimed at device manufacturers who want to build devices with Java technology and Bluetooth wireless technology. To gain the most out of this book, you will find it helpful to have a working knowledge of J2ME and familiarity with Bluetooth wireless technology. The book cites several references that provide additional information on these subjects. We believe that a J2ME programmer will need no additional reading beyond this book to write JABWT applications.

How This Book Is Organized

Different readers of this book will be seeking different information. We have identified three sets of people:

1. Those looking for an overview to make decisions on projects
2. Those who will be leading projects or managing projects in this area
3. Programmers who need detailed information on how to program using JABWT

Apart from the introductory chapters, the chapters are organized into three main sections to accommodate the three sets of people identified above. The three divisions are

1. Overview: The executive introduction
2. API capabilities: The explanation for the project manager
3. Programming with the API: The programmer's guide

Readers can choose the sections that suit their needs in each chapter. Chapters 1 through 3 are overview chapters. Chapters 4 through 8 detail the various sections of the API. Chapter 9 provides examples of JABWT applications. Chapter 10 explains how device manufacturers can implement JABWT on a new device. Throughout the book many code examples are given to explain the API. The complete code for the examples can be found either in the chapters or in Appendix A. The complete JSR-82 API is in Appendixes B and C.

There is a website for this book where you can access the complete code examples found in the book. In addition, you can find the latest news about JABWT, book errata, and other useful links. To access the website, go to www.mkp.com and use the search option with the title of this book.

The topics in this book are organized as follows:

Chapter 1, Introduction, presents an overview of Bluetooth wireless technology and J2ME. It also provides a context for the JABWT specification.

Chapter 2, An Overview of JABWT, defines the goals, characteristics, and scope of JABWT.

Chapter 3, High-Level Architecture, presents the high-level architecture of JABWT.

Chapter 4, RFCOMM, discusses the APIs for Bluetooth serial port communications using RFCOMM.

Chapter 5, OBEX, introduces the architecture and the APIs for making OBEX connections.

Chapter 6, Device Discovery, discusses the APIs for Bluetooth device discovery.

Chapter 7, Service Discovery, describes the APIs for service discovery and service registration.

Chapter 8, L2CAP, presents the API for Bluetooth communications using the logical link control and adaptation protocol.

Chapter 9, Example Applications, illustrates the use of JABWT through two sample applications.

Chapter 10, Implementing JABWT on a Device, describes the basic porting process and highlights the major issues.

Chapter 11, Closing Remarks, provides a summary of the topics discussed in the book and discusses future trends.

Appendix A contains code for the examples from Chapters 7 and 8.

Appendix B contains the Bluetooth API from the package javax.bluetooth.

Appendix C contains the OBEX API from the package javax.obex.

Acknowledgments

A large number of people were involved with the original development of the Java APIs for Bluetooth wireless technology. As the three of us set out to write a book explaining those Bluetooth APIs, we were pleased to discover that we would again receive contributions and assistance from a large number of dedicated and talented individuals.

The authors thank Glade Diviney, Peter Kembro, and Ashwin Kamal Whitchurch for reviewing the entire book in draft form and making valuable comments and suggestions. Thanks also to R. Thiagarajan, N. Murugan, and Franck Thibaut, who commented on various chapters. Ramesh Errabolu, Ranjani Vaidyanathan, and Ravi Viswanathan from the Motorola JSR-82 team also reviewed and commented on several of the chapters. Of course, the authors are totally responsible for any errors that remain.

When this book was in the proposal stage, we received excellent advice and suggestions from Alok Goyal, Teck Yang Lee, Girish Managoli, Brent Miller, Venugopal Mruthyunjaya, N. Ramachandran, Rajeev Shorey, and Mark VandenBrink.

The Java APIs for Bluetooth wireless technology were developed by a team of industry experts, the JSR-82 expert group, and the team at Motorola that drafted the specification, wrote the reference implementation, and developed the conformance tests. The authors believe that the efforts and contributions of all these individuals produced an API that will have important benefits to the Java community. The authors would like to thank the members of the JSR-82 expert group for all their work on the API: Jouni Ahokas, Patrick Connolly, Glade Diviney, Masahiro Kuroda, Teck Yang Lee, Paul Mackay, Brent Miller, Jim Panian, Farooq Anjum, Charatpong Chotigavanich, Peter Dawson, Peter Duchemin, Jean-Philippe Galvan, Daryl Hlasny, Knud Steven Knudsen, Andrew Leszczynski, Martin Mellody, Anthony Scian, and Brad Threatt.

We greatly appreciate all of the contributions of the other members of the JSR-82 team at Motorola: Lawrence Chan, Judy Ho, Will Holcomb, Judy Lin, Mitra Mechanic, and Allen Peloquin. Jim Erwin, Jim Lynch, Aler Krishnan, Ed Wiencek, and Mark Patrick provided a great deal of assistance to the JSR-82 team. Special thanks go to Anne-Marie Larkin, vice president and director, Wireless Software, Applications and Services for her encouragement and her support of this project.

The authors would like to thank Rococo Software for providing us with their Impronto simulator for use as a second JABWT implementation to check the code examples.

The authors are very grateful to Rick Adams, Karyn Johnson, and Mamata Reddy of Morgan Kaufmann for giving us this opportunity and for all their hard work in producing the finished book.

Bala thanks Sundari, Sailesh, and Shiva for their understanding and support through long nights and weekends working on this project. Bala also thanks his mother, Suseela, and sister, Surya, for all their patient nurturing and Mr. B. Kanakasabai for being his lifelong friend and mentor.

Paul thanks his wife, Dianne, for her support and encouragement.

Tim thanks his wife, Karmen, for her encouragement, patience, and support.

C Bala Kumar

Paul Kline

Tim Thompson

About the Authors

C Bala Kumar is a Distinguished Member of the Technical Staff at Motorola. He chaired the industry expert group that defined the Java APIs for Bluetooth wireless technology. He currently leads the systems software team for wireless platforms in Motorola's Semiconductor Products Sector. He received his Master's degree in Electrical Engineering from the University of Texas at Austin.

Paul J. Kline is a Distinguished Member of the Technical Staff at Motorola and the maintenance lead for the JABWT specification. He currently works on the System Software Architecture team in Motorola's Semiconductor Products Sector. He received his Ph.D. in Mathematical Psychology from the University of Michigan.

Timothy J. Thompson is a Senior Software Engineer on the System Software Architecture team in Motorola's Semiconductor Products Sector. He was the OBEX architect on the JABWT specification team at Motorola. He received his Master's degree in Computer Science from Texas A&M University.

This Page Intentionally Left Blank

1

CHAPTER

Introduction

This chapter begins with an introduction to wireless connectivity and Bluetooth® wireless technology. It then gives:

- An overview of the Bluetooth protocol stack
- An overview of the Java™ 2 Platform, Micro Edition
- A description of the need for Java technology in Bluetooth devices

1.1 Wireless Connectivity

The rapid emergence of the Internet has changed the landscape of modern computing. We are in the information age. The term “information age” came about because of the exchange of massive amounts of data between computing devices using wired and wireless forms of communication. We are rapidly moving toward a world in which communications and computing are ubiquitous.

The convergence of computing and communications began in the early 1960s with the development of modems and the private branch exchange (PBX). The modem allowed computers to communicate with each other over telephone lines, and the PBX allowed computers to control the telephone exchange. The result was computer networks. Today, high-speed networks connect servers, personal computers, and other personal computing devices. High-end routers manage the networks. The distinction between voice and data networks has blurred, and the same network tends to carry both types of traffic. The desire and need to communicate with distant computers led to the creation of the Internet. The days of consumers’ buying a personal computer for stand-alone applications have disappeared. These days the primary motive for buying a personal computer is to use it as a communication tool, so that one can have Internet access to e-mail and the World Wide Web.

Increased dependence on the Internet and the need to stay connected from anywhere at all times have led to advances in mobile computing and communications. We have been communicating without wires for some time with satellites, cordless phones, cellular phones, and remote-control devices. However, in recent years the wireless communications industry has seen explosive growth. Long-range wireless communication invariably uses radio frequency (RF). Typically long-range communications use the licensed parts of the RF spectrum, and user fees apply. Short-range communications can use either RF or infrared and typically use unlicensed (i.e., free) parts of the frequency spectrum.

There are many short-range wireless standards, but the three main ones are Infrared from the Infrared Data Association® (IrDA®), Bluetooth wireless technology, and wireless local area network (WLAN). WLAN is also known as IEEE 802.11, and it comes in three main variants, 802.11b and 802.11g, which operate at 2.4 gigahertz (GHz), and 802.11a, which operates at 5 GHz. The IrDA created a wireless communications system that makes use of infrared light. Whereas RF communication can penetrate many objects, IrDA is limited to line of sight. Both 802.11b and Bluetooth wireless technologies communicate in the 2.4-GHz RF band but are aimed at different market segments. The 802.11b technology has a longer range but consumes substantially more power than Bluetooth wireless technology. The 802.11 variant is primarily for data. The only protocol for supporting voice is Voice over Internet Protocol (VoIP). Table 1.1 provides a comparison of these three technologies.

Wireless communications allow computing and communication devices to be used almost anywhere and to be used in new, progressive ways. The increase in wireless mobile Internet devices is proof that wireless connectivity is pervasive. Powerful software programming environments will help fuel this mobile computing explosion by enabling the development of compelling applications. The Java platform provides a powerful programming environment that has great promise for wireless devices. Many mobile devices now come with support for Java 2, Micro Edition (J2ME™) programs. This book explains how to program Bluetooth applications with the Java programming language.

Table 1.1 Comparison of Wireless Communication Technologies

Feature and Function	IrDA	Wireless LAN	Bluetooth Communication
Connection type	Infrared, narrow beam, line of sight	Spread spectrum, spherical	Spread spectrum, spherical
Spectrum	Optical 850–900 nm	RF 2.4 GHz (5 GHz for 802.11a)	RF 2.4 GHz
Transmission power	40-500 mW/Sr	100 mW	10–100 mW
Maximum data rate	9600 bps–16 Mbps (very rare)	11 Mbps (54 Mbps for 802.11a, 802.11g)	1 Mbps
Range	1 m	100 m	10–100 m
Supported devices	2	Connects through an access point	8 (active), 200 (passive)
Voice channels	No support	VoIP	3
Addressing	32-bit physical ID	48-bit MAC	48-bit MAC

1.2 What Is Bluetooth Wireless Technology?

Bluetooth wireless technology is an open specification for a low-cost, low-power, short-range radio technology for ad hoc wireless communication of voice and data anywhere in the world. Let’s examine each of these attributes:

- An open specification means that the specification is publicly available and royalty free.
- Short-range radio technology means devices can communicate over the air using radio waves at a distance of 10 meters (m). With higher transmission power the range increases to approximately 100 m.
- Because communication is within a short range, the radios are low power and are suited for portable, battery-operated devices.
- Bluetooth wireless technology supports both voice and data, allowing devices to communicate either type of content.

- Bluetooth wireless technology works anywhere in the world because it operates at 2.4 GHz in the globally available, license-free, industrial, scientific, and medical (ISM) band.

The ISM frequency band is available for general use by ISM applications, hence several other devices (e.g., WLAN, cordless phones, microwave ovens) operate in this band. Bluetooth wireless technology is designed to be very robust in the face of interference from other devices.

1.2.1 History of Bluetooth Wireless Technology

The origins of Bluetooth communications started in 1994, when Ericsson began a study to find alternatives to connecting mobile phones to its accessories. The engineers looked at a low-power and low-cost radio interface to eliminate cables between the devices. But the engineers also realized that for the technology to be successful it has to be an open standard and not a proprietary one. In early 1998, Ericsson joined Intel, International Business Machines (IBM), Nokia, and Toshiba and formed the Bluetooth Special Interest Group (SIG) to focus on developing an open specification for Bluetooth wireless technology. The original companies, known as promoter companies, publicly announced the global Bluetooth SIG in May 1998 and invited other companies to join the Bluetooth SIG as Bluetooth adopters in return for a commitment to support the Bluetooth specification. In July 1999, the Bluetooth SIG published version 1.0 of the Bluetooth specification. In December 1999, four new promoter companies—3Com, Agere, Microsoft, and Motorola—joined the Bluetooth SIG.

Since then, the awareness of Bluetooth wireless technology has increased, and many other companies have joined the Bluetooth SIG as adopters, which gives them a royalty-free license to produce Bluetooth-enabled products. Adopter companies also have early access to specifications and the ability to comment on them. Interest in the Bluetooth SIG has grown, and there are currently more than 2000 member companies. These companies represent academia and a variety of industries.

Why is this technology called Bluetooth wireless technology? It was named after a Danish Viking king, Harald Blåtand, who ruled circa A.D. 940–981. *Blåtand* loosely translates to “blue tooth.” During his

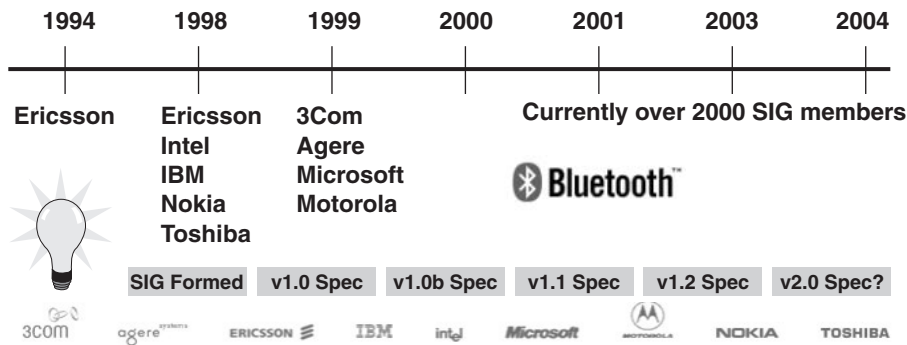


Figure 1.1 Bluetooth SIG timeline.

reign, King Harald Blåtand is supposed to have united and controlled Denmark and Norway. Because this new radio technology was expected to unify the telecommunication and computing industries, it seemed fitting to name it after King Harald. A part-time historian on the team proposed *Bluetooth* as the internal code name. Because the Bluetooth SIG marketing team could not come up with a better name that was not already trademarked, the name stuck.

1.2.2 Bluetooth Vision

Bluetooth wireless technology was originally developed as a cable replacement technology for connecting devices such as mobile phone handsets, headsets, and portable computers with each other (Figure 1.2). However, wireless connectivity between fixed and mobile devices enables many other usage scenarios other than cable replacement. By enabling wireless links and communication between devices, a short-range wireless network was created that gave rise to the notion of a personal area network (PAN). Designed as an inexpensive wireless networking system for all classes of portable devices, Bluetooth devices have the capability to form ad hoc networks. These networks should enable easy and convenient connections to printers, Internet access points, and personal devices at work and at home.

There are so many usage scenarios for Bluetooth wireless technology that the technology will likely be put to wide use. Let's look at a couple of the usage models.

The three-in-one phone usage model allows a mobile telephone to be used as a cellular phone in the normal manner, as a cordless phone that connects to a voice access point (e.g., cordless base station), and as an intercom or "walkie-talkie" for direct communication with another device. The cordless telephony and the intercom features use Bluetooth wireless technology.

The second use case is wireless telematics. Assume that a user who is talking on a cell phone approaches his or her automobile but wants to continue the phone conversation in the hands-free mode. Using Bluetooth communication the user can continue the phone conversa-

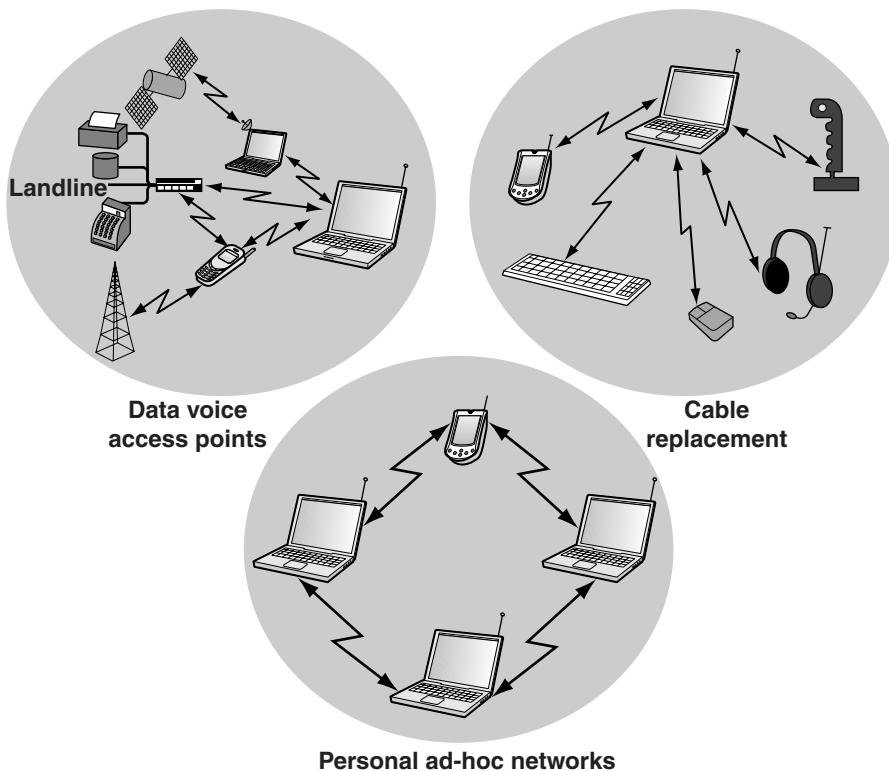


Figure 1.2 Bluetooth use cases.

tion using the microphone and speakers equipped in the dashboard of the automobile.

Another use case is the instant post card, whereby a user (on vacation, for example) with a digital camera transmits a photo via a data access point that could be a mobile phone or a local area network (LAN) access point. Similar use cases include automatic synchronization, business card exchange, hotel and airline check-in, electronic ticketing, and wireless games.

1.2.3 Bluetooth Specification

The Bluetooth specification is the result of cooperation by many companies under the Bluetooth SIG umbrella. The specification defines the over-the-air behavior to ensure compatibility of Bluetooth devices from different vendors. It defines the complete system from the radio up to the application level, including the software stack. The specification is very lengthy because of the breadth of topics it covers. At the highest level, the specification (version 1.1) is split into two volumes (in version 1.2 they are split into several subvolumes). Volume 1 [1] is the core specification and describes the protocol stack and related items such as testing and qualification. The Bluetooth protocol stack is defined as a series of layers somewhat analogous to the familiar Open Systems Interconnect (OSI) [2] standard reference for communication protocol stacks. Each layer of the protocol stack represents a different protocol and is separately described in the core specification.

The Bluetooth profiles are described in volume 2 of the Bluetooth version 1.1 specification [3]. Bluetooth profiles, essentially usage models, describe how applications are to use the Bluetooth stack. A Bluetooth profile is a set of capabilities of the protocol layers that represent a default solution for a usage model. Bluetooth profiles are the basis of Bluetooth protocol stack qualification, and any new implementations of a Bluetooth profile have to go through the qualification process described herein. The specification and profiles continue to evolve as new areas are identified in which Bluetooth wireless technology can be used. Bluetooth protocols and profiles are discussed in detail in the next section. For a detailed description of Bluetooth wireless technology, see books by Miller [4] and Bray [5] and their colleagues.

1.3 Overview of the Bluetooth Stack Architecture

This section provides a brief overview of the Bluetooth protocol stack. The Bluetooth protocol stack can be broadly divided into two components: the Bluetooth host and the Bluetooth controller (or Bluetooth radio module). The Host Controller Interface (HCI) provides a standardized interface between the Bluetooth host and the Bluetooth controller. Figure 1.3 illustrates the Bluetooth host and Bluetooth device classification.

The Bluetooth host is also known as the upper-layer stack and usually is implemented in software. It is generally integrated with the system software or host operating system. Bluetooth profiles are built on top of the protocols. They are generally in software and run on the host device hardware. For example, a laptop computer or a phone would be the host device. The Bluetooth host would be integrated with the operating system on the laptop or the phone.

The Bluetooth radio module or controller usually is a hardware module like a PC card (see Figure 1.3) that plugs into a target device. More and more devices have the Bluetooth controller built into the device. The upper stack interfaces to the Bluetooth radio module via the HCI. The Bluetooth radio module usually interfaces with the host

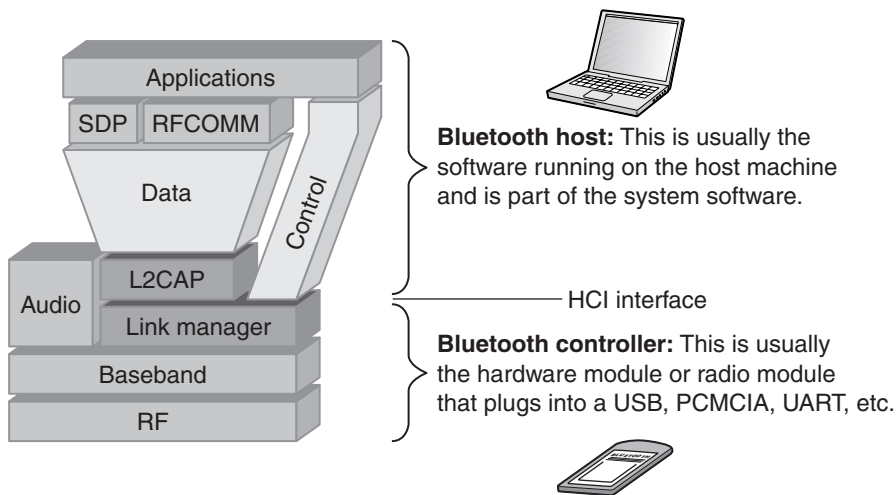


Figure 1.3 Bluetooth host and device classification.

system via one of the standard input/output (I/O) mechanisms, such as peripheral component microchannel interconnect architecture (PCMCIA), universal asynchronous receiver-transmitter (UART), and universal serial bus (USB). Although the Bluetooth host and the Bluetooth controller classifications apply to most devices, the two are integrated in some devices, headsets, for example, and HCI is not used. The various blocks in Figure 1.3 are part of the Bluetooth protocol stack, which is discussed next.

1.3.1 Bluetooth Protocols

Figure 1.4 shows a block diagram of the Bluetooth protocol stack. Several protocols are defined in the Bluetooth specification, but Figure 1.4 shows the common ones. The shaded boxes represent the protocols addressed by Java APIs for Bluetooth wireless technology (JABWT, where API stands for application programming interface). The protocol stack is composed of protocols specific to Bluetooth wireless technology, such as

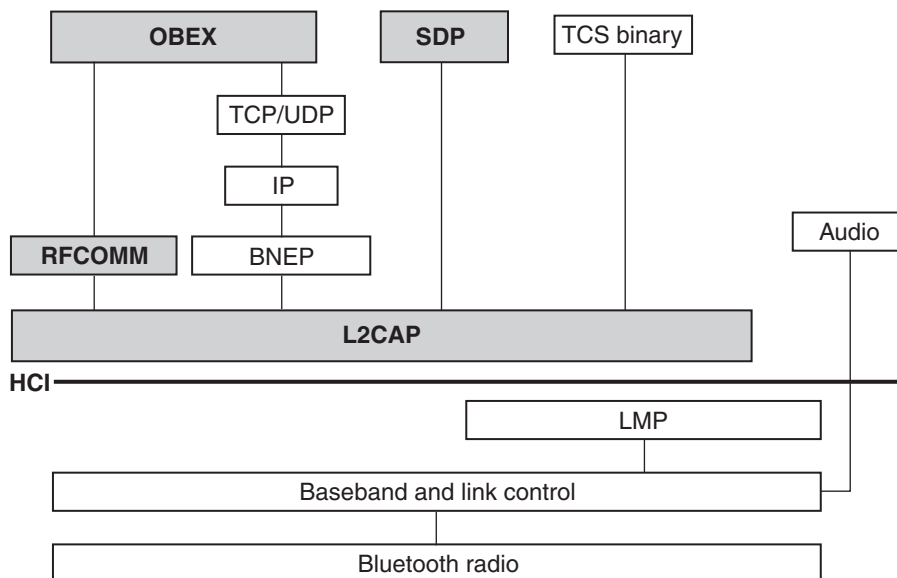


Figure 1.4 Bluetooth protocol stack.

the Service Discovery Protocol (SDP) and other adopted protocols, such as the Object Exchange protocol (OBEX™).

- The Bluetooth radio (layer) is the lowest defined layer of the Bluetooth specification. It defines the requirements of the Bluetooth transceiver device operating in the 2.4-GHz ISM band.
- The baseband and link control layer enables the physical RF link between Bluetooth units making a connection. The baseband handles channel processing and timing, and the link control handles the channel access control. There are two different kinds of physical links: synchronous connection oriented (SCO) and asynchronous connectionless (ACL). An ACL link carries data packets, whereas an SCO link supports real-time audio traffic.
- Audio is really not a layer of the protocol stack, but it is shown here because it is uniquely treated in Bluetooth communication. Audio data is typically routed directly to and from the baseband layer over an SCO link. Of course, if a data channel is used (e.g., in VoIP applications), audio data will be transmitted over an ACL link.
- The Link Manager Protocol (LMP) is responsible for link setup and link configuration between Bluetooth devices, managing and negotiating the baseband packet sizes. The LMP manages the security aspects, such as authentication and encryption, by generating, exchanging, and checking link and encryption keys.
- The HCI provides a command interface to the radio, baseband controller, and link manager. It is a single standard interface for accessing the Bluetooth baseband capabilities, the hardware status, and control registers.
- Logical Link Control and Adaptation Protocol (L2CAP) shields the upper-layer protocols from the details of the lower-layer protocols. It multiplexes between the various logical connections made by the upper layers.
- SDP provides a means for applications to query services and characteristics of services. Unlike in an LAN connection, in which one connects to a network and then finds devices, in a Bluetooth environment one finds the devices before one finds the service. In addition, the set of services available changes in an environment when devices are in motion. Hence SDP is quite different from service

discovery in traditional network-based environments. SDP is built on top of L2CAP.

- Serial ports are one of the most common communications interfaces used in computing and communication devices. The RFCOMM protocol provides emulation of serial ports over L2CAP. RFCOMM provides transport capabilities for upper-level services that use a serial interface as a transport mechanism. RFCOMM provides multiple concurrent connections to one device and provides connections to multiple devices.
- Bluetooth-enabled devices will have the ability to form networks and exchange information. For these devices to interoperate and exchange information, a common packet format must be defined to encapsulate layer 3 network protocols. The Bluetooth Network Encapsulation Protocol (BNEP) [6] encapsulates packets from various networking protocols, and the packets are transported directly over L2CAP. BNEP is an optional protocol developed after Bluetooth specification version 1.1 but based on the 1.1 version of the specification.
- Telephony Control Protocol Specification, Binary (TCS binary) defines the call control signaling for establishment of voice and data calls between Bluetooth devices. It is built on L2CAP.
- Adopted protocols such as OBEX and the Internet Protocol (IP), are built on one of the protocols discussed earlier (e.g., OBEX is built on RFCOMM, and IP is built on BNEP).
- The Bluetooth SIG also is defining newer protocols built on one of the protocols discussed earlier, but mainly they are built on top of L2CAP. Hardcopy Control Channel [7], Hardcopy Notification Channel [7], Audio/Video Control Transport Protocol [8], and Audio/Video Distribution Transport Protocol [9] are examples of some newer protocols.

1.3.2 Bluetooth Profiles

In addition to the protocols, Bluetooth profiles have been defined by the Bluetooth SIG [3]. A Bluetooth profile defines standard ways of using selected protocols and protocol features that enable a particular usage model. In other words, it defines how different parts of the Bluetooth specification can be used for a particular use case. A profile can be

described as a vertical slice through the protocol stack. It defines options in each protocol that are needed for the profile. The dependency of the profiles on protocol layers and features varies. Two profiles may use a different set of protocol layers and a different set of features within the same protocol layer.

A Bluetooth device can support one or more profiles. The four “basic” profiles are the Generic Access Profile (GAP), the Serial Port Profile (SPP), the Service Discovery Application Profile (SDAP), and the Generic Object Exchange Profile (GOEP).

- The GAP is the basis of all other profiles. Strictly speaking, all profiles are based on the GAP. GAP defines the generic procedures related to establishing connections between two devices, including the discovery of Bluetooth devices, link management and configuration, and procedures related to use of different security levels.
- The SDAP describes the fundamental operations necessary for service discovery. This profile defines the protocols and procedures to be used by applications to locate services in other Bluetooth-enabled devices.
- The SPP defines the requirements for Bluetooth devices necessary for setting up emulated serial cable connections using RFCOMM between two peer devices. SPP directly maps to the RFCOMM protocol and enables legacy applications using Bluetooth wireless technology as a cable replacement.
- The GOEP is an abstract profile on which concrete usage case profiles can be built. These are profiles using OBEX. The profile defines all elements necessary for support of the OBEX usage models (e.g., file transfer, synchronization, or object push).

Figure 1.5 shows the relationships among the various Bluetooth profiles. The Bluetooth profiles are hierarchical. For example, the File Transfer Profile is built on top of GOEP, which depends on SPP, which is built upon GAP. Bluetooth profiles also can be classified on the basis of the functional or services point of view. From a programming perspective, however, it is the profile hierarchy that is applicable. The basic profiles—GAP, SDAP, SPP, and GOEP—also are known as *transport profiles*, upon which other profiles, known as *application profiles*, can be built.

Many profiles are based on the basic profiles. More details on these profiles can be obtained from www.bluetooth.com. Figure 1.5 will

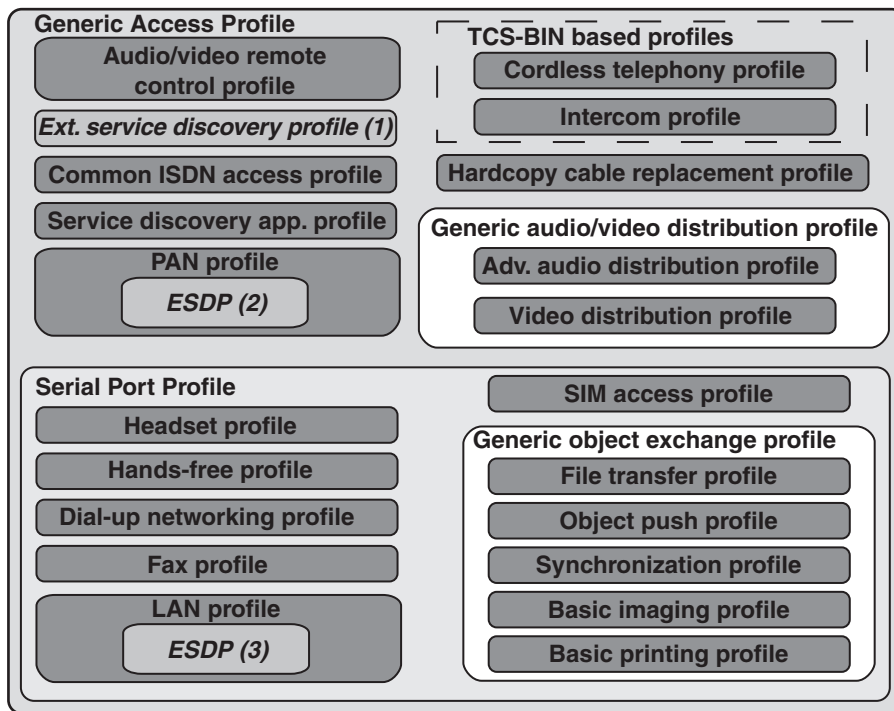


Figure 1.5 Bluetooth profile hierarchy.

probably be obsolete soon because more profiles are being continuously developed. Refer to the Bluetooth website for specifications on the latest available profiles.

1.3.3 Bluetooth Qualification

Bluetooth qualification is the certification process required for any product using Bluetooth wireless technology. The qualification process ensures that products comply with the Bluetooth specification. Only qualified products are entitled to use the free license to the patents required to implement Bluetooth wireless technology, the Bluetooth

brand, and the Bluetooth logo. Essentially, there are three levels of Bluetooth qualification testing:

- Core specification conformance
- Interoperability testing to ensure that devices work with one another at the profile level
- Bluetooth branding conformance

More details on the qualification process can be obtained from the Bluetooth Qualification Program Website [10].

1.4 What is J2ME?

This section gives a brief overview of J2ME. For details about J2ME, refer to books by Topley [11] and Riggs and colleagues [12].

J2ME is the Java platform for consumer and embedded devices such as mobile phones, pagers, personal organizers, television set-top boxes, automobile entertainment and navigation systems, Internet televisions, and Internet-enabled phones. J2ME is one of the three platform editions. The other two platform editions are Java 2 Platform, Enterprise Edition (J2EE™) for servers and enterprise computers and Java 2 Platform, Standard Edition (J2SE™) for desktop computers. A related technology is Java Card™ technology. The Java Card specifications enable Java technology to run on smart cards and other devices with more limited memory than a low-end mobile phone. These groupings are needed to tailor the Java technology to different areas of today's vast computing industry. Figure 1.6 illustrates the Java 2 platform editions and their target markets.

The J2ME platform brings the power and benefits of Java technology (code portability, object-oriented programming, and a rapid development cycle) to consumer and embedded devices. The main goal of J2ME is to enable devices to dynamically download applications that leverage the native capabilities of each device. Consumer and embedded space covers a range of devices from pagers to television set-top boxes that vary widely in memory, processing power, and I/O capabilities. To address this diversity, the J2ME architecture defines configurations, profiles, and optional packages to allow for modularity and customizability. Figure 1.7 shows the high-level relations between the layers of the J2ME architecture. The layers are explained further in the next section.

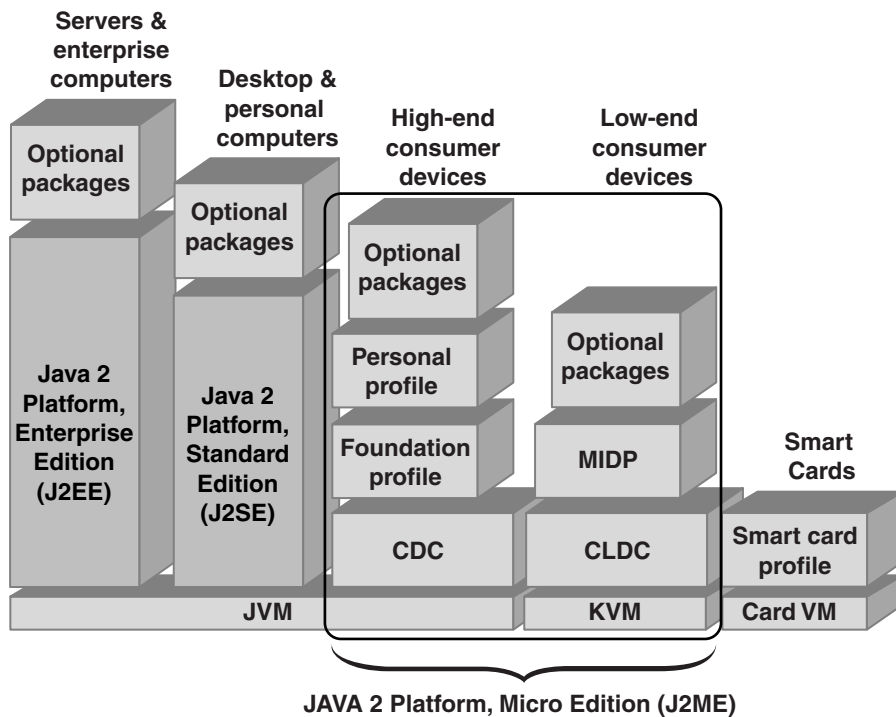


Figure 1.6 Java 2 Platforms.

1.4.1 Configurations

A Java virtual machine interprets the Java byte codes generated when Java programs are compiled. A Java program can be run on any device that has a suitable virtual machine and a suitable set of Java class libraries.

Configurations are composed of a Java virtual machine and a minimal set of class libraries. The Java virtual machine usually runs on top of a host operating system that is part of the target device's system software. The configuration defines the minimum functionality for a particular category or grouping of devices. It defines the minimum capabilities and requirements for a Java virtual machine and class libraries available on all devices of the same category or grouping. Currently, there are two J2ME configurations: the Connected, Limited Device

Configuration (CLDC) [13] and the Connected Device Configuration (CDC) [14].

Connected, Limited Device Configuration

The CLDC focuses on low-end consumer devices and is the smaller of the two configurations. Typical CLDC devices, such as personal organizers, mobile phones, and pagers, have slow processors and limited memory, operate on batteries, and have only intermittent network connections. A CLDC implementation generally includes a kilobyte virtual machine (KVM). It gets its name because of its small memory footprint (on the order of kilobytes). The KVM is specially designed for memory-constrained devices.

Connected Device Configuration

The CDC focuses on high-end consumer devices that have more memory, faster processors, and greater network bandwidth. Typical examples of CDC devices are television set-top boxes and high-end communicators. CDC includes a virtual machine that conforms fully to the Java Virtual Machine Specification [15]. CDC also includes a much larger subset of the J2SE platform than does CLDC.

1.4.2 Profiles

Configurations do not usually provide a complete solution. Profiles add the functionality and the APIs required to complete a fully functional

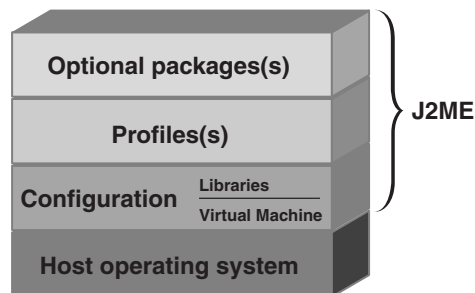


Figure 1.7 Components of J2ME architecture.

runtime environment for a class of devices. Configurations must be combined with profiles that define the higher-level APIs for providing the capabilities for a specific market or industry. It is possible for a single device to support several profiles. Examples of profiles are Mobile Information Device Profile (MIDP), Foundation Profile (FP), and Personal Profile (PP). A clarification: the Bluetooth profiles defined previously are not to be confused with the J2ME profiles discussed here. The two profiles are not related. *Bluetooth profile* refers to a set of functionality of the Bluetooth protocols for a particular usage case. J2ME profiles are a set of APIs that extend the functionality of a J2ME configuration.

Mobile Information Device Profile

The first profile that was created was MIDP [16]. This profile is designed for mobile phones, pagers, and entry-level personal organizers. MIDP combined with CLDC offers core application functionality, such as a user interface, network capability, and persistent storage. MIDP provides a complete Java runtime environment for mobile information devices. MIDP applications are called *MIDlets*. *MIDlet* is a class defined in MIDP and is the superclass for all MIDP applications.

Foundation Profile

The FP [17] is the lowest-level profile for CDC. Other profiles can be added on top as needed to provide application functionality. The FP is meant for embedded devices without a user interface but with network capability.

Personal Profile

The PP [18] is for devices such as high-end personal organizers, communicators, and game consoles that require a user interface and Internet applet support. PP replaces the PersonalJava™ technology and provides PersonalJava applications a clear migration path to the J2ME platform.

In addition there is a Personal Basis Profile (PBP) [19], which is a subset of PP aimed at devices that requires only a basic level of graphical presentation (for example, television set-top boxes).

1.4.3 Optional Packages

Many J2ME devices include additional technologies such as Bluetooth wireless technology, multimedia, wireless messaging, and database connectivity. Optional packages were created to fully leverage these technologies through standard Java APIs. Device manufacturers can include these optional packages as needed to fully utilize the features of each device.

In addition to the configurations, profiles, and optional packages, device manufacturers are able to define additional Java classes to take advantage of features specific to the device. These classes are called *licensee open classes* (LOCs). An LOC defines classes available to all developers. *Licensee closed classes* (LCCs) define classes available only to the device manufacturer. Programs using these classes may not be portable across devices having the same configuration and profiles.

1.5 Why Java Technology for Bluetooth Devices?

How an end user uses Bluetooth wireless technology varies from person to person. Two people with the same model of a Bluetooth-enabled phone might want to use it for different purposes. One person might want to be able to download video games to the phone and use the phone as a television remote control. The other person might want to use the same model phone to unlock car doors, operate kitchen appliances, and open and close garage doors. One way for both people to achieve their goals is to make it possible to download Bluetooth applications onto personal organizers and mobile phones to customize those handheld devices. To make downloading applications a reality, one needs a standard API that lets programmers write Bluetooth applications that work across many hardware platforms. To define this standard API, the Java language is the ideal choice. A Java API enables applications to run on different types of hardware, operating systems, and classes of device. In addition to portability, the Java language provides several other benefits:

- Rapid development of applications because of the better abstractions and high-level programming constructs provided by an object-oriented programming language.
- Ability to dynamically expand a program's functionality during execution by loading classes at runtime.

- Class file verification and security features that provide protection against malicious applications. These safeguards are required to customize devices by downloading applications.
- Standards with better user interfaces and that support sophisticated user interaction.
- Large developer community. The number of people who program in the Java language is continuously growing. The developer talent needed for programming in the Java language already exists, and there is no need to grow a developer community.

For these reasons, the decision was made to develop a standard API for Bluetooth wireless technology using the Java programming language. This standardization effort resulted in the Java APIs for Bluetooth Wireless Technology, or JABWT. As you will see later in this book, this standardization effort complements existing technologies rather than replacing them. JABWT is built on top of the already established and widely used Bluetooth protocol stack.

1.5.1 Java Community Process and JSR-82

Standard APIs in the Java programming language are defined through the Java Community ProcessSM (JCP). The JCP coordinates the evolution of the Java programming language. Each new API is developed as a Java Specification Request (JSR). All J2ME configurations, profiles, and optional packages are defined as JSRs. The process for defining a standard Java API is as follows:

1. The potential specification lead submits a new JSR.
2. The JCP executive committee reviews and votes on the JSR.
3. After JSR approval, the specification lead forms an expert group.
4. The expert group defines the specification.
5. JCP members review the specification during the community review period.
6. The specification is open for public review.
7. The specification lead submits the specification as the proposed final draft.
8. The executive committee votes on the specification to accept or deny the API.

9. If the vote passes, the final release of the specification is announced.

The above process was followed in standardizing the JABWT under JSR-82 [20]. The expert group that defined JABWT consisted of 18 companies and three individuals. The companies were Extended Systems, IBM, Mitsubishi Electric, Motorola, Newbury Networks, Nokia, Parthus Technologies, Research in Motion, Rococo Software, Sharp Laboratories of America, Sony Ericsson Mobile Communications, Smart Fusion, Smart Network Devices, Sun Microsystems, Symbian, Telecordia, Vaultus, and Zucotto. The API was defined as an optional package for J2ME devices based on CLDC.

1.5.2 What about J2SE and PersonalJava?

Because Bluetooth wireless technology can be found in J2SE and PersonalJava devices, you may ask why this standardization effort focused on J2ME devices. The expert group believed that the initial set of devices that would use Java language capabilities over the Bluetooth protocols would be in the J2ME device space. But, as the next chapters show, the API was defined in such a way as to rely heavily on one set of CLDC APIs known as the *Generic Connection Framework* (GCF).

That thinking paid off. An effort was undertaken to include the GCF in J2SE under JSR-197 (Generic Connection Framework Optional Package) [21]. The main goal of JSR-197 is to make the GCF into an optional package that allows applications that rely on the GCF in J2ME to migrate to J2SE. JSR-197 will bring the benefits of JABWT to J2SE.

1.5.3 Jini, JXTA, and Bluetooth Networks

As Bluetooth wireless technology becomes part of the Java programming language, many Java developers are confused about how JABWT will fit in with other Java technologies, such as the Jini architecture [22] and the JXTA (for Juxtapose) research project [23]. Jini is an architecture developed by Sun Microsystems to allow developers to create adaptive network services. Jini works by moving Java objects and data over a network to allow the network to change as the environment of the network changes. JXTA also was developed originally by Sun

Microsystems, but Sun turned it into an open source research project. The goal of the JXTA project is to develop a high-level virtual peer-to-peer network. The JXTA project has defined a communication protocol to enable peer-to-peer networking. The project is currently working on revising communication protocols and identifying applications and services that use the protocols.

Although these three technologies share many basic concepts, Bluetooth wireless technology is not a competitor of Jini and JXTA. Nor does using Bluetooth wireless technology eliminate the ability or need to use Jini and JXTA. These three technologies are similar because they all provide a way to dynamically identify peers and a peer's capabilities. Bluetooth wireless technology differs from Jini and JXTA because Bluetooth wireless technologies defines a set of communication protocols that allow two devices to communicate with each other, in particular, a wireless communication protocol. Whereas Jini and JXTA operate only at a higher level, the Bluetooth specification defines all the components from the application layer down to the physical communication layer. Jini and JXTA can seamlessly span different communications networks. For example, a Jini application could use a Bluetooth protocol to talk to an access point, which converts the requests to Telephony Control Protocol/Internet Protocol (TCP/IP) packets to talk to the Jini service. The Bluetooth specification addresses only how Bluetooth devices communicate, not how to transform requests over any communication medium.

It is likely that Jini and JXTA implementations in the future will use Bluetooth communications and be available over TCP/IP. The Jini and

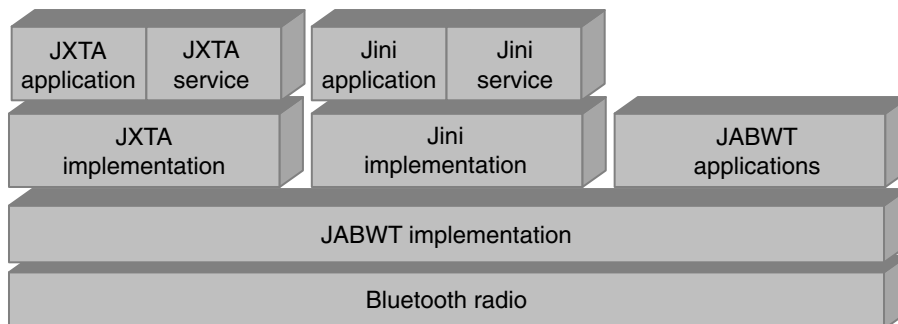


Figure 1.8 Jini and JXTA on a Bluetooth device with JABWT.

JXTA implementations themselves will be Bluetooth services. Figure 1.8 shows that from a JABWT implementation perspective, JXTA and Jini are simply other Bluetooth applications. In this way, Jini and JXTA provide added capabilities to a JABWT implementation.

1.6 Summary

The JABWT specification provides a standard set of APIs for developing Bluetooth applications. The Java APIs defined by JABWT are considered optional packages for J2ME. Applications written with JABWT are potentially portable to a wide range of devices with a wide range of Bluetooth radio modules and Bluetooth protocol stacks.

This chapter gives an overview of Bluetooth wireless technology and J2ME. These are two very large topics. To learn more about Bluetooth wireless technology, refer to the Bluetooth specifications [1, 3] or books on the subject [4, 5]. The following websites are a good place to start:

www.bluetooth.com

www.palowireless.com

To learn more about J2ME, see the books by Topley [11] and by Riggs and associates [12]. Helpful websites for J2ME and JABWT are

java.sun.com

www.jcp.org

www.jcp.org/jsr/detail/82.jsp

There are several articles [24], white papers, and tutorials on these subjects on the Web. There are several newsgroups on Bluetooth wireless technology, but the following two are devoted to JABWT:

groups.yahoo.com/group/jabwt

groups.yahoo.com/group/jsr82

This chapter notes the need for Java technology on Bluetooth devices and explains the process of defining JABWT.

6 CHAPTER

Device Discovery

This chapter covers the following topics:

- Retrieving information on the local device
- Why is device discovery needed?
- Making a device discoverable
- Retrieving devices without an inquiry
- How to start an inquiry
- Changing security on a link
- Working with remote devices

6.1 Overview

Because the typical Bluetooth radio is part of a mobile device, a Bluetooth device must be able to dynamically locate devices within the area. A Bluetooth device must also be able to determine what services are on the devices found. The Bluetooth specification separates discovery of devices and discovery of services into separate processes. In the device discovery process, the local Bluetooth device finds the other Bluetooth devices in the area. In the service discovery process, the Bluetooth device determines which services the other devices have running on them.

In Bluetooth terms, device discovery is known as an *inquiry*. When a Bluetooth device issues an inquiry, the other devices in the area respond to the inquiry requests depending on their discoverable mode. These devices respond with their Bluetooth address and class of device record. The Bluetooth address is a 6-byte unique identifier assigned to every Bluetooth device by the manufacturer. The class of device record describes the type of Bluetooth device and provides a general indication of the types of services available on the device.

At the time of publication, the Bluetooth SIG had defined two types of inquiries: general and limited. A general inquiry is used to find all the Bluetooth devices in an area. A limited inquiry is used to find all devices in an area that are discoverable for only a limited length of time. A general inquiry is similar to asking all people in a room to say their names. A limited inquiry is similar to asking all people in a room to say their names only if they are accountants. Which devices respond to an inquiry request depends on the discoverable mode of the device. A Bluetooth device can be general, limited, or not discoverable. A general discoverable device responds only to general inquiries. Limited discoverable devices respond to general and limited inquiries. A device cannot respond to any inquiries if it is not discoverable.

During a general inquiry, the device performing the inquiry asks all the devices in the area to respond to a general inquiry request. Only

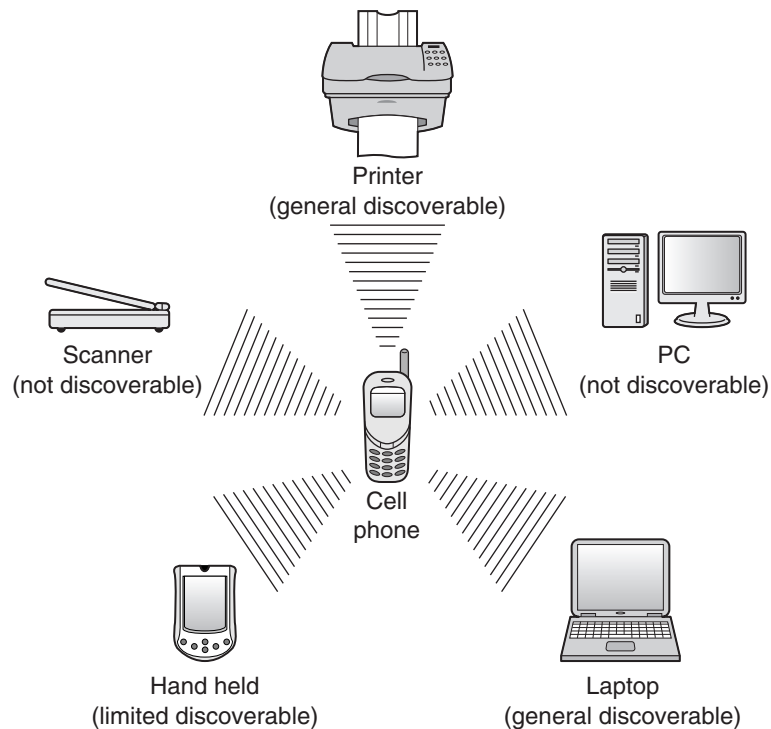


Figure 6.1 Cell phone performs a general inquiry.

devices that are limited or general discoverable respond to the request. For example, in Figure 6.1, the cell phone issues the general inquiry request. Although they may receive the request, the PC and scanner do not respond to the request because they are in the not discoverable mode (Figure 6.2.). The remaining devices in the area do respond to the cell phone with their Bluetooth addresses and class of device record.

6.2 API capabilities

JABWT provides two approaches to device and service discovery. First, JABWT provides methods that allow full control over device and service discovery. The second approach leaves the device and service discovery up to the JABWT implementation. This separation was created to allow the developers to develop Bluetooth profiles with the API and to

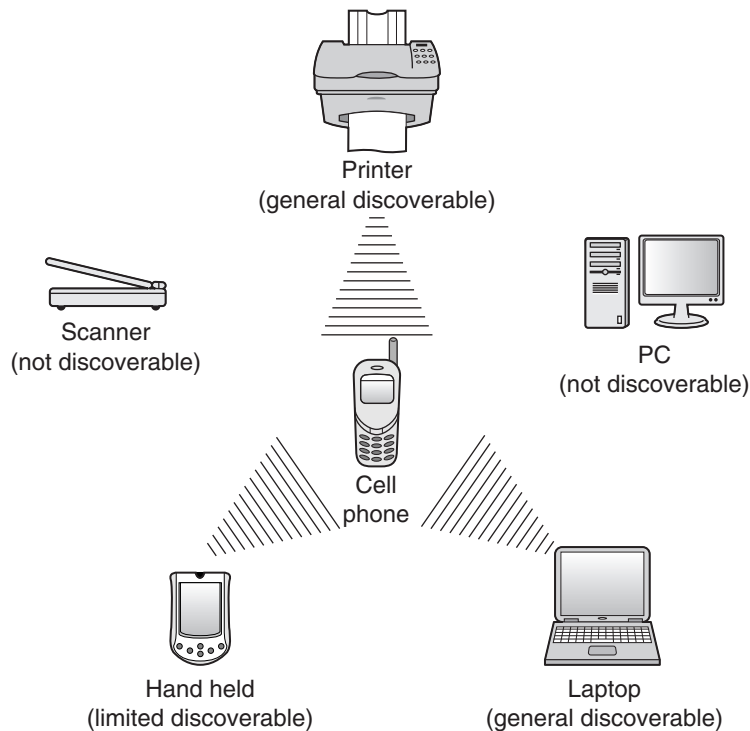


Figure 6.2 All general and limited discoverable devices respond to a general inquiry.

optimize their applications while also allowing developers to quickly get a Bluetooth application up and running. (The second approach is presented in Section 7.3.8.)

Device discovery starts with the JABWT application specifying the type of inquiry to perform, either general or limited. (Additional inquiry types are possible but have yet to be defined by the Bluetooth SIG.) The API returns each device found during an inquiry back to the application as the device is found via a `deviceDiscovered()` event. The device found and the class of device record of the remote device are returned.

The class of device record specifies the type of device responding and the services available on the device. The class of device record is made of the major service class, major device class, and minor device class. The major service class defines the services available on a device. The following list defines all the major service classes defined by the Bluetooth SIG when this book was published.

- Positioning
- Networking
- Capturing
- Object Transfer
- Audio
- Information
- Limited Discoverable
- Rendering
- Telephony

A device can have multiple major *service* classes. In other words, a device can have an audio and a telephony service running on it at the same time, which makes the major service class audio and telephony. The major device class and minor device class describe the physical type of device of which the Bluetooth radio is a part. A device can have only a single major *device* class. The Bluetooth SIG has defined the following major device classes:

- Computer (e.g., desktop, notebook, PDA, organizers)
- Phone (e.g., cellular, cordless, pay phone, modem)
- LAN/network access point
- Audio/video (e.g., headset, speaker, stereo, video display)

- Peripheral (e.g., mouse, joystick, keyboard)
- Imaging (e.g., printing, scanner, camera, display)
- Miscellaneous

The minor device class is defined on the basis of the major device class. The list of all the minor device classes is available from the Bluetooth SIG [30]. The minor device class is simply a more specific description of the device. For example, a device can be classified as an imaging device. The minor device class specifies whether the device is a camera, scanner, or printer. This system allows developers to perform service searches on only the devices that fit their needs. In the example, the developer is able to eliminate the camera and scanner devices from the service search if the developer is looking for a printing service.

JABWT provides a simpler approach to retrieving remote devices that eliminates the need for performing a complete inquiry. When it performs an inquiry, an application must wait eight to ten seconds for a 95% chance of finding all the devices in the area. In addition to taking time, an inquiry is power consuming, which is a major concern for embedded, battery-powered devices. To minimize the need for an inquiry, JABWT allows an application to retrieve without performing an inquiry a list of devices that would “likely” be in an area. These devices are called *predefined*. There are two types of predefined devices: pre-known and cached. Pre-known devices are devices with which the local device frequently interacts. Pre-known devices are set in the BCC. (Section 3.2 provides a full description of the BCC.) For example, a PDA frequently synchronizes with a desktop computer. The user of the PDA would specify the desktop computer as a pre-known device. A pre-known device does not guarantee that the device is currently reachable or available, but it does give an educated guess. The second type of device that can be retrieved without an inquiry is a cached device. Cached devices are Bluetooth devices found with a previous inquiry. The inquiry does not have to be performed by this application. The cached devices might have been found by a previous inquiry performed by another application on this same Bluetooth device.

6.3 Programming with the API

The `LocalDevice` class provides access to the local Bluetooth device. There is only a single `LocalDevice` object for the entire JABWT

implementation. This `LocalDevice` object provides methods of retrieving information on the local device and a gateway for starting the different discovery processes.

6.3.1 Accessing the Local Device

Retrieving information and manipulating a local Bluetooth device begin with retrieving the `LocalDevice` object. The `LocalDevice` class has a private constructor, which prevents an application from creating a new `LocalDevice` object on its own. For an application to retrieve the `LocalDevice` object for the JABWT implementation, the application calls the `LocalDevice.getLocalDevice()` method. The `getLocalDevice()` method may throw a `BluetoothStateException` if the Bluetooth stack or radio is not working properly.

Once the `LocalDevice` has been retrieved, more information can be gathered on the local device, such as the device's Bluetooth address, friendly name, current discoverable mode, and class of device record. The `getBluetoothAddress()` method returns the device's address. The `getFriendlyName()` method returns the device's user-friendly name or returns null if the name could not be retrieved. The `getDiscoverable()` method returns the local device's current discoverable mode. The `DiscoveryAgent` class contains constants for general discoverable (`GIAC`, which stands for General Inquiry Access Code), limited discoverable (`LIAC`, which stands for Limited Inquiry Access Code), and not discoverable (`NOT_DISCOVERABLE`). Other discoverable modes are possible but have not been defined by the Bluetooth SIG. The `getDiscoverable()` method returns the actual value of the other discoverable modes if the local device is in the mode. Finally, the class of device record can be retrieved by means of the `getDeviceClass()` method. The `getDeviceClass()` method returns null if the class of device record could not be retrieved. (Section 6.3.4 describes manipulating the class of device record through a `DeviceClass` object.)

The `LocalDevice` class has a method for requesting a different discoverable mode. The `setDiscoverable()` method takes the requested discoverable mode as an argument. The discoverable mode can be `DiscoveryAgent.GIAC`, `DiscoveryAgent.LIAC`, `DiscoveryAgent.NOT_DISCOVERABLE`, or any discoverable mode in the range 0x9E8B00 to 0x9E8B3F. The `setDiscoverable()` method returns true if the

discoverable mode changes to the requested mode. This method returns `false` if the BCC denies the request or the local device does not support the requested discoverable mode. The `setDiscoverable()` method throws a `BluetoothStateException` if the requested change cannot occur at this time because the device is in a state that does not allow the change.

What happens when one device asks for limited discoverable while another asks for general discoverable? This is left up to the BCC. The JSR-82 specification does not place requirements on how the BCC resolves conflicting discoverable requests. The possible approaches include but are not limited to the following:

1. The first requested discoverable mode is honored until the application ends.
2. The last request is always honored.
3. One discoverable mode has a higher priority than another. For example, general discoverable has a higher priority than limited discoverable. Not discoverable is the lowest priority. A request to a higher priority is honored, whereas a request for a lower priority is denied.

The `LocalDevice` class also has the `getProperty()` method for retrieving additional information on the capabilities of the Bluetooth radio and stack. The `LocalDevice.getProperty()` method works similarly to the `System.getProperty()` method. The `LocalDevice.getProperty()` method takes, as a `String`, the specific property whose value is to be retrieved. The `getProperty()` method returns the value of the property as a `String`. Table 6.1 lists all the properties available through the `getProperty()` method. A JABWT implementation can add parameters to the `getProperty()` method but must support at a minimum all the properties in Table 6.1. The argument is case sensitive.

To show how to use these methods within an application, the `BluetoothInfoMIDlet` follows. The `BluetoothInfoMIDlet` displays a `Form` with the Bluetooth address, friendly name, current discoverable mode, and all the properties of the local Bluetooth device. The first step to creating the MIDlet is creating the `BluetoothInfoMIDlet` class and adding code to its `startApp()` method to display a `Form`. The next step is to add an exit `Command` to the `Form` to destroy the MIDlet.

Table 6.1 Properties Available through `LocalDevice.getProperty`

Property	Description	Valid Value
<code>bluetooth.api.version</code>	The version of the JABWT that is supported. This property does not relate to the Bluetooth specification number.	"1.0" for the current version of the JABWT
<code>bluetooth.master.switch</code>	Is master/slave switch allowed?	"true" or "false"
<code>bluetooth.sd.attr.retrievable.max</code>	Maximum number of service attributes to be retrieved per service record.	A base 10 integer (e.g., "1", "2")
<code>bluetooth.connected.devices.max</code>	Maximum number of connected devices supported.	A base 10 integer (e.g., "1", "2")
<code>bluetooth.l2cap.receiveMTU.max</code>	Maximum ReceiveMTU size in bytes supported in L2CAP.	A base 10 integer (e.g., "1", "2")
<code>bluetooth.sd.trans.max</code>	Maximum number of concurrent service discovery transactions.	A base 10 integer (e.g., "1", "2")
<code>bluetooth.connected.inquiry.scan</code>	Can the local device respond to an inquiry request while the device has established a link to another device?	"true" or "false"
<code>bluetooth.connected.page.scan</code>	Can the local device accept a connection from a new remote device if it is already connected to another remote device?	"true" or "false"
<code>bluetooth.connected.inquiry</code>	Can the local device start an inquiry while it is connected to another device?	"true" or "false"
<code>bluetooth.connected.page</code>	Can the local device establish a connection to a remote device if the local device is already connected to another device?	"true" or "false"

```

package com.jabwt.book;

import java.lang.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.bluetooth.*;

public class BluetoothInfoMIDlet extends BluetoothMIDlet {
    /**
     * Called when the MIDlet is started to display the
     * properties of the
     * MIDlet on a Form.
     */
    public void startApp() throws
        MIDletStateChangeException {
        Display currentDisplay = Display.getDisplay(this);

        Form infoForm = new Form("Device Info");
        currentDisplay.setCurrent(infoForm);

        // Add the exit command and set the listener
        infoForm.addCommand(new Command("Exit",
            Command.EXIT, 1));
        infoForm.setCommandListener(this);
    }
}

```

Now that the `BluetoothInfoMIDlet` has been created, the `getBluetoothInfo()` method is added. The `getBluetoothInfo()` method retrieves the Bluetooth device information and displays it on the screen. This method first retrieves the `LocalDevice` object. Next, the Bluetooth address is retrieved and displayed on the screen. Before the user-friendly name returned from `getFriendlyName()` is displayed, the `getBluetoothInfo()` method must verify that null was not returned. If null was returned, the appropriate message is added to the Form; otherwise, the user-friendly name is displayed. A switch statement is used to display the discoverable mode to resolve the value of the discoverable mode with its associated name. Finally, each of the properties available via the `LocalDevice.getProperty()` method is retrieved and appended to the Form. The values returned from

getProperty() do not need to be checked for null because these values are required to be part of the JABWT implementation.

```
public class BluetoothInfoMIDlet
    extends BluetoothMIDlet {

    public void startApp() throws
        MIDletStateChangeException {
        Display currentDisplay = Display.getDisplay(this);

        Form infoForm = new Form("Device Info");
        currentDisplay.setCurrent(infoForm);

        getBluetoothInfo(infoForm);

        // Add the exit command and set the listener
        infoForm.addCommand(new Command("Exit",
            Command.EXIT, 1));
        infoForm.setCommandListener(this);
    }

    /**
     * Displays the Bluetooth device information on the
     * screen.
     * @param f the Form to display the information on
     */
    private void getBluetoothInfo(Form f) {
        LocalDevice local = null;

        // Retrieve the local Bluetooth device object
        try {
            local = LocalDevice.getLocalDevice();
        } catch (BluetoothStateException e) {
            f.append("Failed to retrieve the local device (" +
                e.getMessage() + ")");
            return;
        }

        // Retrieve the Bluetooth address
        f.append("Address: " + local.getBluetoothAddress());
    }
}
```

```
f.append("\n");
// Retrieve the Bluetooth friendly name
String name = local.getFriendlyName();
if (name == null) {
    f.append("Name: Failed to Retrieve");
} else {
    f.append("Name: " + name);
}
f.append("\n");

// Retrieve the current discoverable mode
int mode = local.getDiscoverable();
StringBuffer text = new StringBuffer(
    "Discoverable Mode: ");
switch (mode) {
    case DiscoveryAgent.NOT_DISCOVERABLE:
        text.append("Not Discoverable");
        break;
    case DiscoveryAgent.GIAC:
        text.append("General");
        break;
    case DiscoveryAgent.LIAC:
        text.append("Limited");
        break;
    default:
        text.append("0x");
        text.append(Integer.toString(mode, 16));
        break;
}
f.append(text.toString());
f.append("\n");

// Retrieve all the default properties
// and display them on the screen.
f.append("API Version: " +
    local.getProperty("bluetooth.api.version"));
f.append("\n");
f.append("Master Switch Supported: " +
    local.getProperty("bluetooth.master.switch"));
f.append("\n");
```

```

        f.append("Max Attributes: " +
            local.getProperty(
                "bluetooth.sd.attr.retrievable.max"));
        f.append("\n");
        f.append("Max Connected Devices: " +
            local.getProperty(
                "bluetooth.connected.devices.max"));
        f.append("\n");
        f.append("Max Receive MTU: " +
            local.getProperty(
                "bluetooth.l2cap.receiveMTU.max"));
        f.append("\n");
        f.append("Max Service Discovery Transactions: " +
            local.getProperty("bluetooth.sd.trans.max"));
        f.append("\n");
        f.append("Connection Options\n");
        f.append(" Inquiry Scan Supported: " +
            local.getProperty(
                "bluetooth.connected.inquiry.scan"));
        f.append("\n");
        f.append(" Page Scan Supported: " +
            local.getProperty(
                "bluetooth.connected.page.scan"));
        f.append("\n");
        f.append(" Inquiry Supported: " +
            local.getProperty("bluetooth.connected.inquiry"));
        f.append("\n");
        f.append(" Page Supported: " +
            local.getProperty("bluetooth.connected.page"));
        f.append("\n");
    }
}

```

The BluetoothInfoMIDlet displays a great deal of information about the local Bluetooth device and its JABWT implementation. Figure 6.3 shows a screen shot of the BluetoothInfoMIDlet running.

Simple Device Discovery

JABWT provides device and service discovery capabilities via the DiscoveryAgent class and the DiscoveryListener interface. Each



Figure 6.3 BluetoothInfoMIDlet running in J2ME Wireless Toolkit (emulation only).

device has a single `DiscoveryAgent` object. The `DiscoveryAgent` provides methods to start device and service searches. The `DiscoveryListener` interface is used by the `DiscoveryAgent` to pass devices and services back to the application as they are found. This `DiscoveryAgent` object is retrieved from the local device via the `LocalDevice.getDiscoveryAgent()` method. The following method shows how to retrieve the `DiscoveryAgent` object for a Bluetooth device.

```
public static DiscoveryAgent getLocalDiscoveryAgent() {
    try {
        /*
         * Retrieve the local Bluetooth device object for
         * this device.
         */
    }
}
```

```

        * This method may throw a BluetoothStateException
        * if the local
        * device could not be initialized.
        */
        LocalDevice local = LocalDevice.getLocalDevice();
        DiscoveryAgent agent = local.getDiscoveryAgent();
        return agent;
    } catch (BluetoothStateException e) {
        return null;
    }
}

```

The `getLocalDiscoveryAgent()` method must catch a `BluetoothStateException` because `LocalDevice.getLocalDevice()` may throw it. Once the `LocalDevice` is retrieved, the call to `getDiscoveryAgent()` returns the `DiscoveryAgent` associated with the `LocalDevice`. Multiple calls to `getDiscoveryAgent()` return the same object.

The `DiscoveryListener` interface is the other intricate part of device and service discovery in the JABWT. The `DiscoveryListener` interface is implemented by an application to receive devices and service records as they are discovered. The interface also provides an application with a notification that the inquiry or service search has completed.

The easiest way to retrieve a list of `RemoteDevices` is using the `retrieveDevices()` method in the `DiscoveryAgent` class. The `retrieveDevices()` method returns the list of pre-known devices if the `DiscoveryAgent.PREKNOWN` argument is used and the list of cached devices if the `DiscoveryAgent.CACHED` option is set. Pre-known devices are devices with which the local device commonly interacts. Devices can be registered as pre-known in the BCC. There is no guarantee that a pre-known device is in the area or can be connected to. Cached devices are devices that have been found via a previous inquiry. How many devices are cached and for how long is implementation dependent. Because of these facts, there is no guarantee that a cached or pre-known device is currently available, but the `retrieveDevices()` method is a quick way to get to the service search phase.

To show how to use the `retrieveDevices()` method, the `DiscoveryMIDlet` is created to display all the pre-known and cached devices. To add this capability, the `startApp()` method creates a `List` and makes it the current displayable. The `startApp()` method calls the



Figure 6.4 DiscoveryMIDlet retrieving list of pre-known and cached devices (emulation only).

`addDevices()` method to retrieve the pre-known and cached devices to print out the Bluetooth addresses of each of these devices. To identify which devices are pre-known and which are cached, “-P” or “-C” is appended to each device’s Bluetooth address to represent pre-known and cached, respectively. Figure 6.4 shows the DiscoveryMIDlet running within the J2ME Wireless Toolkit.

```
package com.jabwt.book;

import java.lang.*;
import java.util.*;
import java.io.*;
import javax.microedition.midlet.*;
```

```
import javax.microedition.lcdui.*;
import javax.bluetooth.*;

public class DiscoveryMIDlet extends BluetoothMIDlet {
    /**
     * The List of remote devices
     */
    private List deviceList;

    /**
     * The DiscoveryAgent for the local device.
     */
    private DiscoveryAgent agent;

    /**
     * Retrieves the list of pre-known and cached devices.
     * Updates the
     * display to show the list of devices.
     */
    public void startApp() throws MIDletStateChangeException {
        // Create a new List and set it to the current
        // displayable
        deviceList = new List("List of Devices", List.IMPLICIT);
        deviceList.addCommand(new Command("Exit",
            Command.EXIT, 1));
        deviceList.setCommandListener(this);
        Display.getDisplay(this).setCurrent(deviceList);

        // Retrieve the DiscoveryAgent object. If
        // retrieving the local device causes a
        // BluetoothStateException, something is wrong so
        // stop the app from running.
        try {
            LocalDevice local = LocalDevice.getLocalDevice();
            agent = local.getDiscoveryAgent();
        } catch (BluetoothStateException e) {
            // Prevent the application from starting if the
            // Bluetooth device
            // could not be retrieved.
            throw new MIDletStateChangeException(
                "Unable to retrieve local Bluetooth device.");
        }
    }
}
```

```

        addDevices();
    }

    /**
     * Updates the List of devices with the cached and
     * pre-known devices.
     */
    private void addDevices() {
        // Retrieve the pre-known device array and append the
        // addresses
        // of the Bluetooth device. If there are no pre-known
        // devices, move
        // on to cached devices.
        RemoteDevice[] list = agent.retrieveDevices(
            DiscoveryAgent.PREKNOWN);
        if (list != null) {
            for (int i = 0; i < list.length; i++) {
                String address = list[i].getBluetoothAddress();
                deviceList.insert(0, address + "-P", null);
            }
        }

        // Retrieve the cached device array and add the
        // addresses to the
        // list.
        list = agent.retrieveDevices(DiscoveryAgent.CACHED);
        if (list != null) {
            for (int i = 0; i < list.length; i++) {
                String address = list[i].getBluetoothAddress();
                deviceList.insert(0, address + "-C", null);
            }
        }
    }
}

```

6.3.2 Device Discovery via Inquiry

Starting an inquiry is more complicated than simply retrieving a list of devices. An inquiry requires the Bluetooth radio to issue requests for all devices in the area to respond according to their discoverable mode.

This process uses power and can prevent the radio from being used for other purposes. Before an application can request an inquiry, the application must implement the `DiscoveryListener` interface. This interface requires that the `deviceDiscovered()` and the `inquiryCompleted()` methods be implemented for device discovery. To actually start an inquiry, the `startInquiry()` method is used. The `startInquiry()` method takes, as arguments, the type of inquiry and an implementation of the `DiscoveryListener` interface.

JABWT provides constants for the two types of inquiries defined by the Bluetooth SIG. The `DiscoveryAgent.GIAC` inquiry type should be passed to `startInquiry()` for a general inquiry. A general inquiry locates all devices in the general or limited discoverable mode. For a limited inquiry, the `DiscoveryAgent.LIAC` inquiry type should be used. Again, a limited inquiry finds all devices in the area in the limited discoverable mode. A device is placed in the limited discoverable mode when it is discoverable for only a short time. In addition to general and limited inquiry, the Bluetooth SIG specified that other inquiry access codes may be defined in the range `0x9E8B00` to `0x9E8B3F` in the future. Therefore, `startInquiry()` accepts any value in this range.

The `startInquiry()` method returns `true` if the inquiry is started. Because not all devices support all the inquiry access codes, the `startInquiry()` method returns `false` if a valid inquiry access code is provided but the code is not supported. Valid inquiry access codes are in the range `0x9E8B00` to `0x9E8B3F` along with `GIAC` and `LIAC`. The `startInquiry()` method throws an `IllegalArgumentException` if an invalid inquiry access code is passed as an argument. The `startInquiry()` method throws a `BluetoothStateException` if the inquiry could not be started because the device is in a state that does not allow an inquiry to be completed. A device may be in such a state when the device is already performing an inquiry or when the local device is already connected to another device.

After starting the inquiry using `startInquiry()`, JABWT returns devices to the application via `deviceDiscovered()` events. A `deviceDiscovered()` event occurs every time a remote Bluetooth device is found by the Bluetooth radio. The `deviceDiscovered()` event provides the `RemoteDevice` object and associated `DeviceClass` object each time the event occurs. The `RemoteDevice` object provides the Bluetooth address of a remote Bluetooth device along with methods to retrieve the friendly name and security controls for the remote device. The

DeviceClass object contains the class of device of the RemoteDevice. (The DeviceClass object is explained further in Section 6.3.4).

JABWT provides a way to cancel an inquiry. An application may want to cancel an inquiry once it finds a specific Bluetooth device or if the application is paused or destroyed. The `cancelInquiry()` method cancels an inquiry. To prevent one application from canceling the inquiry of another Bluetooth application, the `cancelInquiry()` method takes one parameter, the `DiscoveryListener` object used when the inquiry was started. The `cancelInquiry()` method returns `true` if the inquiry was canceled. If `cancelInquiry()` returns `false`, an inquiry could not be found associated with the `DiscoveryListener` provided, so no inquiry is canceled.

To notify the application that the inquiry has been completed, the `inquiryCompleted()` event was added to JABWT. The `inquiryCompleted()` event provides the reason the inquiry ended as an argument to the method. The `DiscoveryListener.INQUIRY_COMPLETED` reason is specified if the inquiry completes normally. The `DiscoveryListener.INQUIRY_TERMINATED` reason is passed as part of the `inquiryCompleted()` event if the inquiry was canceled by the application using the `cancelInquiry()` method. The call to `cancelInquiry()` is a non-blocking call. The `inquiryCompleted()` event occurs independently of the `cancelInquiry()` method ending. Finally, the `inquiryCompleted()` event receives a `DiscoveryListener.INQUIRY_ERROR` reason if an error occurs during processing of the inquiry.

The following code shows a simple MIDlet that starts an inquiry and displays all the devices that respond to the inquiry request. The Bluetooth address of each device is displayed in a `List`. When the inquiry ends and the `inquiryCompleted()` method is called, an `Alert` appears to notify the user that the inquiry has ended. If an error occurs during processing of the MIDlet, an `Alert` is displayed to the user to notify the user of the error.

```
public class DiscoveryMIDlet extends BluetoothMIDlet
    implements DiscoveryListener {
```

```
    /**
     * Retrieves the list of pre-known and cached
     * devices. Updates the
```

```

    * display to show the list of devices.
    */
    public void startApp() throws
        MIDletStateChangeException {
        // Create a new List and set it to the current
        // displayable
        deviceList = new List("List of Devices",
            List.IMPLICIT);
        deviceList.addCommand(new Command("Exit",
            Command.EXIT, 1));
        deviceList.setCommandListener(this);
        Display.getDisplay(this).setCurrent(deviceList);

        // Retrieve the DiscoveryAgent object. If
        // retrieving the local device causes a
        // BluetoothStateException, something is wrong
        // so stop the app from running.
        try {
            LocalDevice local = LocalDevice.getLocalDevice();
            agent = local.getDiscoveryAgent();
        } catch (BluetoothStateException e) {
            // Prevent the application from starting if
            // the Bluetooth device
            // could not be retrieved.
            throw new MIDletStateChangeException(
                "Unable to retrieve local Bluetooth device.");
        }

        addDevices();

        try {
            agent.startInquiry(DiscoveryAgent.GIAC, this);
        } catch (BluetoothStateException e) {
            throw new MIDletStateChangeException(
                "Unable to start the inquiry");
        }

    }

    ...

    /**

```

```
* Called each time a new device is discovered.
* This method prints the device's Bluetooth
* address to the screen.
*
* @param device the device that was found
* @param cod the class of device record
*/
public void deviceDiscovered(RemoteDevice device,
    DeviceClass cod) {
    String address = device.getBluetoothAddress();
    deviceList.insert(0, address + "-I", null);
}

/**
* Called when an inquiry ends. This method
* displays an Alert to notify the user the inquiry
* ended. The reason the inquiry ended is displayed
* in the Alert.
*
* @param type the reason the inquiry completed
*/
public void inquiryCompleted(int type) {
    Alert dialog = null;

    // Determine if an error occurred. If one did
    // occur display an Alert
    // before allowing the application to exit.
    if (type != DiscoveryListener.INQUIRY_COMPLETED) {
        dialog = new Alert("Bluetooth Error",
            "The inquiry failed to complete normally",
            null, AlertType.ERROR);
    } else {
        dialog = new Alert("Inquiry Completed",
            "The inquiry completed normally", null,
            AlertType.INFO);
    }

    dialog.setTimeout(Alert.FOREVER);
    Display.getDisplay(this).setCurrent(dialog);
}
```



Figure 6.5 DiscoveryMIDlet after discovering devices via an inquiry (emulation only).

```

    public void servicesDiscovered(int transID,
        ServiceRecord[] record) {
    }
    public void serviceSearchCompleted(int transID, int
        type) {
    }
}

```

Most of the DiscoveryMIDlet code is required by the MIDP specification. The important parts of the code are found in the `startApp()`, `deviceDiscovered()`, and `inquiryCompleted()` methods. The inquiry is started with `startInquiry()` in the `startApp()` method so that it occurs each time the MIDlet is made active. If started in the constructor, the inquiry occurs only when the MIDlet is created. If a Bluetooth-

`StateException` occurs during retrieval of the `LocalDevice` object or the start of the inquiry, the `startApp()` method throws a `MIDlet-StateException` to notify the KVM that the MIDlet is not able to run correctly. This procedure simplifies the user experience by allowing the KVM to handle the user interaction in the case of this type of error.

The `deviceDiscovered()` and `inquiryCompleted()` methods must be implemented because the `DiscoveryMIDlet` implements the `DiscoveryListener` interface. The `deviceDiscovered()` method is important because this is the method used to pass the remote devices found in the inquiry back to the MIDlet. For the purpose of this MIDlet, all this method does is get the remote device's Bluetooth address and add it to the `List`. (Figure 6.5 shows the `DiscoveryMIDlet` running.) The `inquiryCompleted()` method verifies that the inquiry completed successfully. If the inquiry did not complete properly, an `Alert` is displayed to the user to notify the user of the error. If the inquiry did complete properly, an `Alert` saying so is displayed to the user.

Because the user may exit from the MIDlet before the inquiry ends, code must be added to cancel the inquiry. Therefore the `commandAction()` method is modified to call `cancelInquiry()`. Because calling `cancelInquiry()` when no inquiry is occurring does nothing, the `commandAction()` method calls `cancelInquiry()` every time the user exits from the MIDlet.

```
public class DiscoveryMIDlet extends BluetoothMIDlet
    implements DiscoveryListener {

    ...

    /**
     * Called when a Command is selected. If it is an
     * Exit Command, then the
     * MIDlet will be destroyed.
     *
     * @param c the Command that was selected
     * @param d the Displayable that was active when
     * the Command was selected
     */
    public void commandAction(Command c, Displayable d) {
        if (c.getCommandType() == Command.EXIT) {
            // Try to cancel the inquiry.
        }
    }
}
```

```

        agent.cancelInquiry(this);
        notifyDestroyed();
    }
}

```

6.3.3 Retrieving Information from a Remote Device

A number of methods provide additional information about a remote device. Before any of these methods can be called, a `RemoteDevice` object must be created. There is no public constructor for the `RemoteDevice` class, so an application cannot directly instantiate a new `RemoteDevice` object. The application must use one of the three ways to get a `RemoteDevice` object. First, `RemoteDevice` objects are created in the device discovery process. `RemoteDevice` objects are passed to the application as arguments via `deviceDiscovered()` events.

Second, a class that extends the `RemoteDevice` class can be written and instantiated by an application. The following code does just this.

```

package com.jabwt.book;

import javax.bluetooth.*;

public class MyRemoteDevice extends RemoteDevice {
    /**
     * Creates a new RemoteDevice object based upon the
     * address provided.
     *
     * @param address the Bluetooth address
     */
    public MyRemoteDevice(String address) {
        super(address);
    }
}

```

The address provided in the constructor must be 12 hex characters with no preceding "0x." If the address is the same as that of the local device or if it contains non-hex characters, the constructor throws an `IllegalArgumentException`. If the address string is null, the con-

structor throws a `NullPointerException`. After instantiating a new `MyRemoteDevice` object, an application can call any of the `RemoteDevice` methods.

The third and final way to get a `RemoteDevice` object is using the `RemoteDevice.getRemoteDevice()` static method. The `getRemoteDevice()` method takes, as an argument, a Bluetooth connection to a remote device. The `getRemoteDevice()` method returns a `RemoteDevice` object representing the device to which the Bluetooth connection is connected. The `getRemoteDevice()` method throws an `IOException` if the connection is closed. The method throws an `IllegalArgumentException` if the connection is not a Bluetooth connection object or if the connection object is a notifier object.

After the `RemoteDevice` object is retrieved, the `getBluetoothAddress()`, `isTrustedDevice()`, and `getFriendlyName()` methods can be invoked. The `getBluetoothAddress()` method returns the Bluetooth address of the remote device. The `getFriendlyName()` method in the `RemoteDevice` class is different from the `getFriendlyName()` of the `LocalDevice` method. The `getFriendlyName()` method of the `RemoteDevice` class takes a boolean argument that specifies whether the JABWT implementation should always retrieve the friendly name from the remote device or if it should retrieve the name only if the friendly name for the remote device is not known. Retrieving the friendly name requires the local device to establish a link to the remote device to retrieve the name. Because the friendly name on a device rarely changes, using a cached value if one exists eliminates the need to establish the link to the remote device. The `getFriendlyName()` method throws an `IOException` if the remote device could not be contacted to retrieve the friendly name.

Bluetooth security can be specified after a connection is established by means of the `RemoteDevice` class. The `RemoteDevice` class provides methods for authenticating, encrypting, and authorizing a connection after a connection has been established to a remote device by the `authenticate()`, `encrypt()`, and `authorize()` methods, respectively.

The `authenticate()` method authenticates the remote device represented by the `RemoteDevice` object. The `authenticate()` method requires an existing connection to the `RemoteDevice`. If no connection exists, the `authenticate()` method throws an `IOException`. Calling `authenticate()` can cause a pairing to occur if the remote and local devices have not paired previously. The `authenticate()` method returns `true` if the remote device is authenticated;

otherwise, it returns `false`. If the remote device has already been authenticated, the `authenticate()` method returns immediately with the value `true`. In other words, once a link has been authenticated, JABWT does not try to authenticate the remote device again until the link is destroyed and a new one is created.

The `encrypt()` method works slightly differently. It allows encryption on a connection to be turned on and off. This method takes two parameters: the connection to change the encryption on and a boolean specifying whether encryption should be turned on or off. The `Connection` object passed to `encrypt` must be to the same remote Bluetooth device the `RemoteDevice` object is representing. The `Connection` object must also be a `RFCOMM`, `L2CAP`, or `OBEX` over `RFCOMM` connection. Like `authenticate()`, the `encrypt()` method returns `true` if the change succeeds and `false` if it fails.

Changing the encryption on a link is more complicated than simply authenticating a link. The request to turn on encryption can fail for a variety of reasons. First, encryption requires the link to be authenticated. If the authentication or pairing fails, then the request to turn on encryption also fails. Second, the remote device may not support or may not want encryption enabled on the link. Third, the BCC may not allow encryption on the link.

Turning off encryption is even more complicated because turning off encryption actually makes the link less secure. A request to turn off encryption may fail for two reasons. First, the remote device may require that encryption be enabled on the link. Second, the BCC may not allow encryption to be turned off. This may be a system-wide policy, or another application may be running on the device that requires the link to be encrypted. For all of these reasons, a call to `encrypt()` should be considered a request and its return value checked.

The final method in the `RemoteDevice` class that allows a change in the security of a connection is the `authorize()` method. Recall that authorization is done on a connection basis as opposed to a link basis. The `authorize()` method takes the `Connection` object to authorize. Because authorization requires a link to be authenticated, a call to `authorize()` can cause authentication and pairing if these events have not occurred on the link. After it has been verified that the link has been authenticated, the `authorize()` method requests the BCC to authorize the connection. The `authorize()` method returns `true` if the connection is authorized and `false` if the connection is not authorized.

The `RemoteDevice` class has three methods that allow an application to determine the security level on a connection. The `isAuthenticated()` and `isEncrypted()` methods return true if the link to the remote device has been authenticated or encrypted, respectively. Both methods return false if the requested security is not enabled or if there is no link between the two devices. The `isAuthorized()` method works slightly differently. This method takes the `Connection` object to check for authorization. The `isAuthorized()` method returns true if the connection has been authorized and false if it has not been authorized.

Finally, the `RemoteDevice` class contains a method that allows an application to determine whether a device is a trusted device. The `isTrustedDevice()` method returns true if the `RemoteDevice` object represents a trusted device. A trusted device is a device that always passes authorization. This condition is set and maintained by the BCC.

6.3.4 Using the DeviceClass Class

The `DeviceClass` is a unique object. It provides three methods to access the class of device record. The class of device record specifies the physical type of the device and the general services it provides. The `getServiceClasses()` method retrieves the list of all service classes on the device. Each time a service registers itself with a device, the type of service is specified in the class of device record. This procedure allows another device to identify whether a remote device may have a service it is looking for. For example, if an application is looking for a printing service, the application should look for a device that has a rendering service class. This eliminates the overhead of performing a service search on a device that does not have the requested service.

The `getServiceClasses()` method returns an integer representing all the major service classes available on a device. (The Bluetooth SIG in the Bluetooth Assigned Numbers document [30] defines the major service classes.) Because a device can have multiple service classes, the Bluetooth SIG defines a service class by setting a bit in the class of device record. For example, bit 18 is set for the rendering service class. If a device has a rendering and an audio service, bits 18 and 21 are set. In this situation, `getServiceClasses()` returns an integer with bits 18 and 21 set or the value 18874368 (0x1200000). To determine whether a device has a rendering service, bit 18 must be isolated. The following code provides a way of doing this.

```

boolean checkForRenderingService(DeviceClass d) {
    // The Rendering service bit is bit 18. Setting bit
    // 18 produces the
    // number 0x40000.
    if ((d.getServiceClasses() & 0x40000) != 0) {
        return true;
    } else {
        return false;
    }
}

```

The `checkForRenderingService()` method isolates the rendering service bit by performing an AND on the service class of the device and a number with only bit 18 set. If the result of this AND is zero, then bit 18 is not set, and a rendering service is not available on the device. Table 6.2 lists the major service classes currently defined by the Bluetooth SIG, the bit number of the service class, and the integer value of setting only that bit. The Bluetooth SIG may add service classes in the future. Application developers should use the major service class cautiously because it gives only an indication of the types of services available on a device.

Table 6.2 Major Service Classes Defined by the Bluetooth SIG

Service Class	Type of Service	Bit Number	Hex Value
Limited Discoverable Mode	Device is in the limited discoverable mode	13	0x2000
Positioning	Location identification	16	0x10000
Networking	LAN, ad hoc, etc.	17	0x20000
Rendering	Printing, speaker, etc.	18	0x40000
Capturing	Scanner, microphone, etc.	19	0x80000
Object Transfer	V-inbox, v-folder, etc.	20	0x100000
Audio	Speaker, microphone, headset service, etc.	21	0x200000
Telephony	Cordless telephony, modem, headset service, etc.	22	0x400000
Information	Web server, Wireless Application Protocol (WAP) server, etc.	23	0x800000

Table 6.3 Major Device Classes Defined by the Bluetooth SIG

Major Device Class	Example	Hex Value
Computer	Desktop, notebook, PDA, organizer	0x100
Phone	Cellular, cordless, pay phone, modem	0x200
LAN/network access point		0x300
Audio/video	Headset, speaker, stereo, video display, VCR	0x400
Peripheral	Mouse, joystick, keyboard	0x500
Imaging	Printer, scanner, camera, display	0x600
Miscellaneous	All other devices	0x000

The major device class is different from the service classes. The major device class reports the physical type of device to which the Bluetooth radio is connected. Because a device cannot have more than one major device class, there is no need to check a bit. The `getMajorDeviceClass()` returns the major device class value. At present, the Bluetooth SIG has defined seven major device classes. Table 6.3 lists all major device classes defined by the Bluetooth SIG at the time of publication of this book. The following code shows how simple it is to check for the major device class.

```
boolean checkForImaging(DeviceClass d) {
    if (d.getMajorDeviceClass() == 0x600) {
        return true;
    } else {
        return false;
    }
}
```

The minor device class returned via the `getMinorDeviceClass()` method must be interpreted on the basis of the major device class. Each major device class has a list of minor device classes that specify more information about the specific device. For example, the minor device class specifies whether a device with the computer major device class is a desktop, notebook, or PDA. The full listing of minor device classes for each major device class is available in the Bluetooth Assigned Numbers document [30].

6.4 Summary

Device discovery is a key part of any JABWT application. In Bluetooth terms, device discovery is known as *inquiry*. There are two types of inquiry: general and limited. Devices respond to inquiry requests according to their discoverable mode. There are three types of discoverable mode: not discoverable, limited discoverable, and general discoverable. When a device issues a general inquiry, all devices that are limited and general discoverable respond. When a device performs a limited inquiry, only devices that are limited discoverable respond.

All device and service discovery is started with the `DiscoveryAgent` class. The `DiscoveryAgent` class provides two different methods for discovering devices. The `retrieveDevices()` method allows applications to retrieve a list of devices found via a previous inquiry or a list of devices with which the local device frequently communicates. The `startInquiry()` method actually performs an inquiry. As devices are found, they are passed back to the application via `deviceDiscovered()` events. In addition to the devices, `deviceDiscovered()` events also pass back the class of device record. The class of device record contains information on the type of device and the services available on the device.

This chapter shows how to retrieve additional information on local and remote devices. The `LocalDevice` class provides methods that allow the applications to request the current discoverable mode, retrieve the friendly name of the local device, and retrieve information on the JABWT implementation. The `RemoteDevice` class provides similar methods for retrieving additional information on the remote device. The `RemoteDevice` class also provides methods for setting and retrieving different security settings on the link to the remote device.

7

CHAPTER

Service Discovery

This chapter covers the following topics:

- What is a Bluetooth service?
- What is a service record?
- How to perform a service search
- Retrieving additional service record attributes
- Using the simple device and service discovery API
- What is service registration?
- How are service records created and added to the SDDB?
- How are service records modified by server applications?
- What is a connect-anytime service and how does it differ from a run-before-connect service?

7.1 Overview

After the devices in an area are discovered, the next step before connecting to a device is finding the services a device has running on it. Unlike device discovery, the service discovery process involves only a single pair of devices (Figure 7.1). The service discovery process requires the device searching for services to ask a device with services whether it has a service defined by a service record that has a specific set of attributes. If a remote device has a service with the attributes specified, the remote device returns the service record describing the service. The service record has multiple attributes. These attributes provide additional information on a specific service. These attributes may contain anything, including information on how to connect to the service.

Service discovery follows the client-server model. A service discovery client issues a service search request to a service discovery server. The

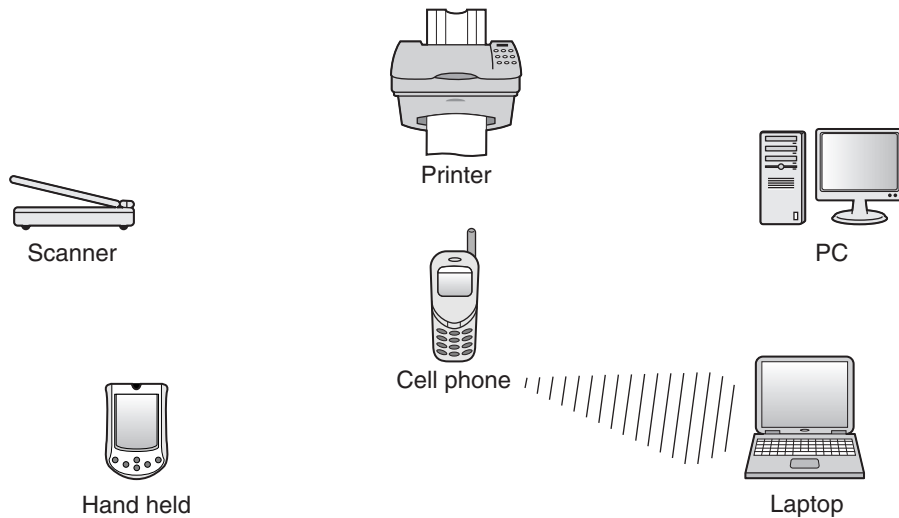


Figure 7.1 Unlike an inquiry, a service search is between only two devices.

service discovery server determines whether the server has any services that meet the search criteria. For a server to know what services are available, each service registers with a service discovery database (SDDB) kept by the Bluetooth stack. When it receives a service search request, the server searches the SDDB for the specified service.

Chapter 3 lists the typical responsibilities of a Bluetooth server application:

1. Creating a service record that describes the service offered by the application
2. Adding the service record to the server's SDDB to make potential clients aware of this service
3. Registering the Bluetooth security measures associated with a service
4. Accepting connections from clients that request the service offered
5. Updating the service record in the SDDB if characteristics of the service change
6. Removing or disabling the service record in the SDDB when the service is no longer available

Responsibilities 1, 2, 5, and 6 compose a subset of the server responsibilities having to do with advertising a service to client devices. We call this subset *service registration*. This chapter describes the process of service registration with JABWT and the process of service discovery with JABWT. Server responsibility 3, which involves security, is discussed in Chapter 4. Server responsibility 4, which involves accepting client connections, is discussed in Chapter 4 for serial port servers and Chapter 5 for OBEX servers. Chapter 8 discusses the process of accepting client connections for L2CAP servers.

7.1.1 Key Concepts

A service record answers the following questions:

- What kind of service does this server application offer?
- How does a client connect to this service?

Figure 3.3 is an overview of the components involved in service registration and service discovery. Service records and the Service Discovery Protocol (SDP) are described in great detail in Part E of the Bluetooth specification v1.1 [1]. However, the JABWT specification is the first standard API for Bluetooth service registration. The following questions about service registration were left unanswered by the Bluetooth specification:

- How are service records created?
- How are service records added to the SDDB so clients can discover them?
- How are service records modified?
- How are service records removed from the SDDB (or otherwise disabled) so clients can no longer discover them?

The Bluetooth specification did not define requirements in these areas because a standardized approach to service registration was not required for ensuring interoperability of Bluetooth devices from different manufacturers. Consequently, the mechanics of service registration were left for Bluetooth stack implementations to define. The result was a variety of different APIs for accomplishing service registration. The standard API defined by JABWT service registration makes it possible to

write Bluetooth server applications that are portable across all JABWT implementations. JABWT service registration also potentially serves as a model for Bluetooth APIs in other programming languages.

7.1.2 Services Defined by Bluetooth Profiles

The Bluetooth SIG has provided profile specifications [3] that describe standardized services. Examples of some of these services are LAN access services, file transfer services, business-card exchange services, and synchronization services. Additional profiles describing standardized services were developed after the Bluetooth profiles specification v1.1 [3]. For example, three printing profiles were defined that offer various forms of printing services [7, 27, 28].

If a service is defined by a Bluetooth profile, then the profile specification describes the requirements for the service record, device security, device discoverable modes, and so on. If you want to claim that your service implements a Bluetooth profile, you have to qualify your application through the Bluetooth qualification process [10].

7.1.3 Custom Services

Developers can define their own Bluetooth server applications beyond and independently of those specified in the Bluetooth profiles and make these services available to remote clients. Applications that do not claim to provide a service described in a Bluetooth profile do not need to undergo the Bluetooth qualification process. Custom services have a great deal more latitude about how they are implemented than do Bluetooth profile implementations. The developers of custom services provide the software for both communicating Bluetooth devices. The server application can be tailored to particular characteristics of the client implementation. This process is different from that for servers for Bluetooth profiles, which must be written to work with many different implementations of the client application.

7.2 API capabilities

Once a list of devices has been retrieved via device discovery, the next step for a Bluetooth application is determining which applications or

services are available on the remote Bluetooth device. JABWT provides a nonblocking way to retrieve all service records that meet a specific set of requirements on a remote Bluetooth device. A service record describes a service and is made of a set of attributes. Attributes specify how to connect to a service, the name of the service, a description of the service, and other useful information. When an application searches for a service, the application provides a set of UUIDs to search for. (A UUID is a bit sequence that uniquely identifies a characteristic of a service.) UUIDs are used to describe attributes of a service. Some UUIDs are specified by the Bluetooth specification. Other UUIDs are defined on a service-by-service basis. JABWT provides a way to specify a set of attributes to retrieve once a service is found. When a service is found that contains all the UUIDs specified, the service's service record is returned via a `servicesDiscovered()` event.

To provide a simple way to get a JABWT application up and running, JABWT defines a method that performs both device and service discovery while hiding the details of both capabilities. The `selectService()` method allows an application to specify a single UUID, which is used to locate the requested service on a remote device. The `selectService()` method returns a connection string that can be used by `Connector.open()` to connect to the service found. This is a blocking method and can take longer than 10 seconds in some situations. Therefore an application should invoke this method in a separate thread to prevent the application from appearing frozen to the user.

7.2.1 Run-before-Connect Services

Ordinarily, a server application must be running and ready to accept connections before a client attempts to make a connection to the server. Server applications that have this requirement are called *run-before-connect services* in the JABWT specification. Figure 7.2 is a Unified Modeling Language (UML) sequence diagram that illustrates the messages involved in service registration for a run-before-connect service. Each arrow in the sequence diagram is a message. The top-to-bottom ordering of the arrows indicates the time sequence of the messages. The boxes at the top of the vertical *lifelines* indicate the objects that send or receive the messages. (If these diagramming conventions are unfamiliar, a description of UML sequence diagrams can be found in Fowler and

Scott [31].) In Figure 7.2 the boxes are all Java objects created by a JABWT program with the exception of the `SDDBRecord`. The `SDDBRecord` is a service record in the SDP server's database. The `SDDBRecord` is not directly accessible by a JABWT application.

Figure 7.2 illustrates the answers that JABWT offers to the following questions about Bluetooth service records for run-before-connect services:

- How are service records created?
- How are service records added to the SDDB so clients can discover them?
- How are service records removed from the SDDB (or otherwise disabled) so clients can no longer discover them?

The methods `open()`, `acceptAndOpen()`, and `close()` shown in Figure 7.2 are part of the GCF defined by CLDC [13]. These methods are used by a variety of J2ME applications for I/O operations. The basic approach taken by JABWT is to add additional behavior to these GCF methods so that service records are automatically created and then automatically added and removed from the SDDB, as follows:

- `Connector.open(String url)` creates a Bluetooth service record if the parameter `url` starts with `btspp://localhost:`, `btgoep://localhost:`, or `btl2cap://localhost:`.
- The first time an `acceptAndOpen()` message is sent to the notifier, a copy of the service record is added to the SDDB.
- When a `close()` message is sent to the notifier, the service record is removed from the SDDB or disabled. (The "X" at the bottom of the `SDDBRecord` timeline in Figure 7.2 is sequence diagram notation for deleting an object.)

One consequence of the JABWT approach is that in many cases the server application can rely only on the automatic behavior of the JABWT implementation and does not need to contain any code to explicitly manipulate service records.

7.2.2 Servers Are Usually Discoverable

The GAP specification [3] describes several modes of operation that characterize Bluetooth devices. Chapter 6 describes the discoverable

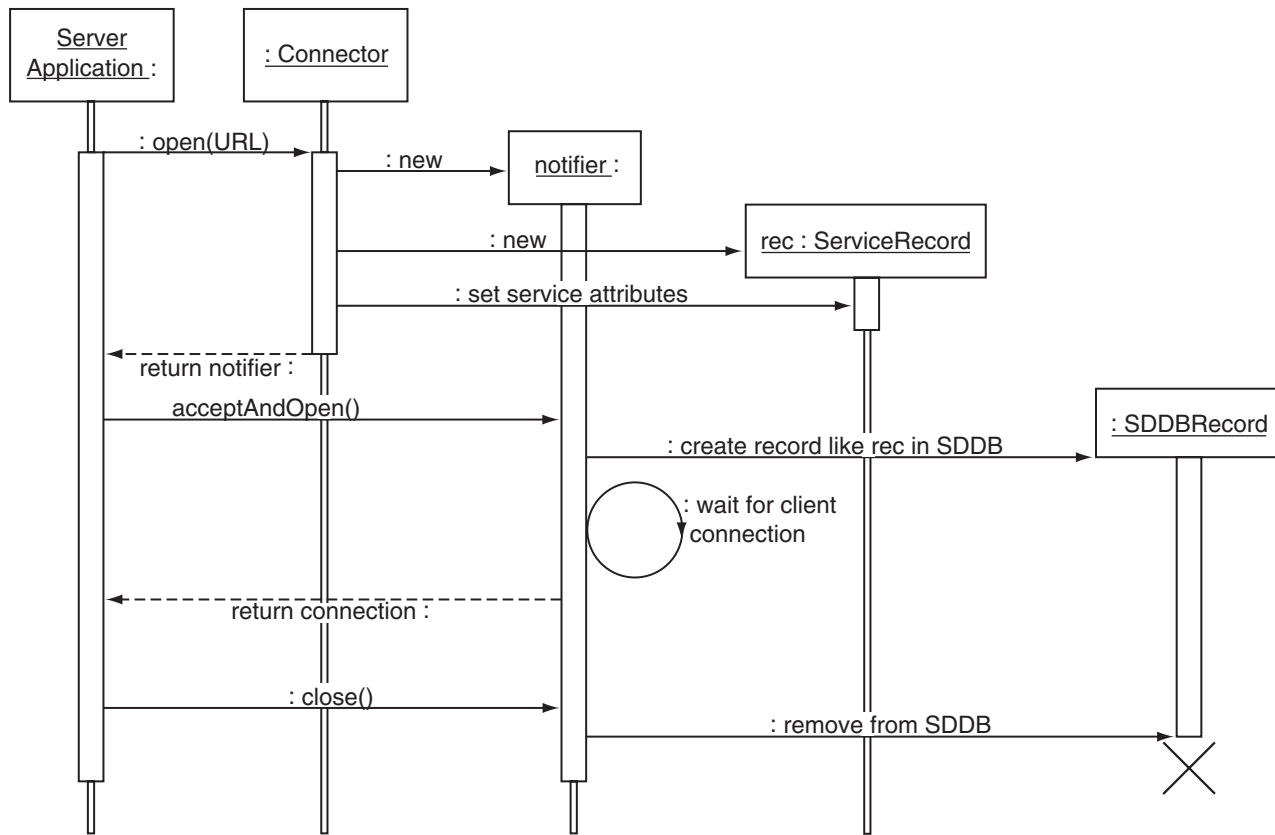


Figure 7.2 Life cycle of a service record for a run-before-connect service.

mode that determines whether the local device will respond to an inquiry conducted by a remote device. Most server applications want the local device to be in either general discoverable or limited discoverable mode so clients can find them and access their service. JABWT server applications can request that the device be made discoverable by means of the instance method `LocalDevice.setDiscoverable()`, with an argument of either `DiscoveryAgent.GIAC` or `DiscoveryAgent.LIAC`.

There are some scenarios in which it makes sense for a server to be nondiscoverable. These scenarios require that the clients have some advance knowledge about the existence of the server device and its Bluetooth address. For example, users who own a PC and a PDA that both have Bluetooth wireless technology might want their server applications to be accessible only to their own devices. If server applications are run on both the PC and the PDA, users can make those services available only to their own devices by

- Making the PC nondiscoverable using the BCC on the PC
- Making the PDA nondiscoverable using the BCC on the PDA
- Making the PDA a pre-known device for the PC using the BCC on the PC
- Making the PC a pre-known device for the PDA using the BCC on the PDA

Because device users may have their own reasons for making the local device nondiscoverable, the JABWT implementation is not the final authority on whether the device will enter a discoverable mode. The implementation makes a request to the BCC to make the local device discoverable, but this request might not be satisfied if the device user has chosen to make the local device nondiscoverable (see Chapter 3).

7.2.3 Register a Service

This section describes the service records automatically created by the JABWT implementation for server applications. These service records allow clients to find the service and make a connection to the server. In many cases, these automatically generated service records are sufficient, and the server application does not need to take any other action.

These descriptions of the default service records are provided so that developers

- Can decide whether these service records are sufficient to advertise their services, and
- Can determine how to modify the service records when modifications are necessary

Service Records for Serial Port

Table 7.1 illustrates the service record automatically created by the JABWT implementation when a server application executes the following statement:

```
Connector.open("btspp://localhost:68EE141812D211D78EED00B0D
03D76EC;name=SPPEX");
```

The Bluetooth Profiles specification [3] contains a template for the service record used by the SPP. The JABWT implementation uses this template to create a service record and inserts the appropriate value for the RFCOMM server channel identifier into the service record. The result is a minimal but sufficient service record.

The service record in Table 7.1 shows four (attribute ID, attribute value) pairs. Each pair describes one attribute of the service. The shaded rows in Table 7.1 are the attribute IDs and the unshaded rows are the attribute values. The Bluetooth SDP uses a value between 0 and $2^{16}-1$ (65535) to represent each attribute ID in a service record, and these are shown in Table 7.1 as hexadecimal numbers. For example, the attribute ID 0x0001 indicates the ServiceClassIDList attribute ID, one of the attribute IDs defined by the Bluetooth SDP specification [1]. Table 7.2 provides a list of the most common attribute IDs defined in the Bluetooth Assigned Numbers [30].

Each attribute value is a DataElement. A DataElement is a self-describing data structure that contains a type and a value. For example, the third attributeValue in the service record in Table 7.1 is a simple DataElement with type String and value "SPPEX." The value "SPPEX" is extracted by the JABWT implementation from the parameter "name=SPPEX" in the connection string. The JABWT implementation uses this string "SPPEX" to construct a DataElement of type String. This

Table 7.1 Service Record Created by Connector.open("btspp://localhost:...")

ServiceClassIDList<0x0001>
DataElement(type = DATSEQ, DataElement(type = UUID, UUID(68EE141812D211D78EED00B0D03D76EC) —from the connection string) DataElement(type = UUID, UUID(SerialPort<0x1101>)))
ProtocolDescriptorList<0x0004>
DataElement(type = DATSEQ, DataElement(type = DATSEQ, DataElement(type = UUID, UUID(L2CAP<0x0100>))) DataElement(type = DATSEQ, DataElement(type = UUID, UUID(RFCOMM<0x0003>)) DataElement(type = U_INT_1, 1 —server channel identifier. —Assigned by the stack; —added to the service record —by JABWT)))
ServiceName<0x0100>
—Name of the service in the primary language of the service record
DataElement(type = STRING, "SPPEX" —from "name=SPPEX" in the connection string)

ServiceRecordHandle<0x0000>
DataElement(type = U_INT_4, 12345 —value assigned by the SDP Server)

DataElement is then used as the ServiceName attribute of the service record.

A DataElement may be one of the following types.

- Null
- Integer (1, 2, 4, 8, and 16 byte)
- Unsigned integer (1, 2, 4, 8, and 16 byte)
- URL
- UUID
- String
- Boolean
- DATALT—Data element alternative
- DATSEQ—Data element sequence

Three data element types require further explanation: UUID, data element sequence, and data element alternative.

A data element sequence is a list of data elements in which all elements are part of the definition. In other words, a data element sequence is an all-inclusive set of data elements. A data element alternative, on the other hand, is a data element whose value is a list of data elements of which any one may be selected. Put slightly differently, a data element alternative is a set of data elements of which any one of the values may be used.

The first attribute value in Table 7.1 contains the thirty-two-digit hexadecimal number 68EE141812D211D78EED00B0D03D76EC. This number is also extracted from the connection string. The number is used to create a 128-bit UUID and then is wrapped in a DataElement of type UUID. UUIDs are used extensively in creating service records, and their meaning varies depending on where they are used in the service record. In this case, the UUID represents one of the ServiceClasses in the ServiceClassIDList. ServiceClasses are very important in identifying

Table 7.2 Some Service Record Attribute IDs Defined by Bluetooth Assigned Numbers

Name	ID	Type	Description
ServiceRecordHandle	0x0000	32-bit unsigned integer	Uniquely identifies each service on a device.
ServiceClassIDList	0x0001	A data element sequence of UUIDs	Defines the service classes that describes the service. The service classes are defined by the Bluetooth SIG.
ServiceID	0x0003	UUID	Uniquely identifies the service instance associated with this service record.
ProtocolDescriptorList	0x0004	A data element sequence of data element sequences of UUID and optional parameters	Describes the protocols to use to connect to the service.
ServiceInfoTimeToLive	0x0007	32-bit unsigned integer	Defines the length of time this service record is valid and will remains unchanged.
ServiceAvailability	0x0008	8-bit unsigned integer	Describes the relative availability of the service to accept additional connections.
BluetoothProfileDescriptorList	0x0009	A data element sequence of data element sequence pairs	Specifies all the profiles this service implements.
DocumentationURL	0x000A	URL	A URL that points to the documentation for the service.
IconURL	0x000C	URL	A URL that points to an icon that may be used to represent the service.
ServiceName	0x0100	String	The name of the service in the primary language of this service record.
ServiceDescription	0x0101	String	A description of the service in the primary language of this service record.

services. For example, each Bluetooth profile is associated with a particular ServiceClass UUID. If a client wants to find the service record for a particular JABWT server application, it can search for the ServiceClass UUID used by that server application.

The other service class in the ServiceClassIDList is the 16-bit UUID, 0x1101, which identifies this as a serial port service record. This list of two service classes summarizes the type of service being offered. Because this attribute value represents a list of service classes, the two DataElements that represent individual service classes are wrapped in another DataElement that represents the entire list (or sequence) of service classes. This wrapper DataElement for the list has type DATSEQ, which is an abbreviation for DataElement Sequence.

In the ServiceClassIDList attributeValue in Table 7.1 we see both 16-bit and 128-bit UUIDs. Although a Bluetooth UUID always represents a 128-bit value, the Bluetooth specification defines both 16-bit and 32-bit “short forms” or aliases for some common 128-bit Bluetooth UUIDs. For example, the Serial Port Service Class ID, 0000110100001000800000805F9B34FB, is a 128-bit UUID that has a 16-bit short form 0x1101. It takes fewer bits to store and transmit a service record when these short form UUIDs are used, so the short forms are generally used when they are available. The short-form UUIDs are defined by the Bluetooth Assigned Numbers [30].

A 16-bit UUID can be converted to a 128-bit UUID by means of the following formula:

$$\text{UUID}_{128} = (\text{UUID}_{16} * 2^{96}) + 0x0000000000001000800000805F9B34FB$$

Table 7.3 shows examples of the use of this formula to convert from a 16-bit UUID to a 128-bit UUID.

Table 7.3 Examples of Conversion from a 16-Bit UUID to a 128-Bit UUID

Mnemonic	16-Bit UUID	128-Bit UUID
RFCOMM	0x0003	0x0000000300001000800000805F9B34FB
BNEP	0x000F	0x0000000F00001000800000805F9B34FB
L2CAP	0x0100	0x0000010000001000800000805F9B34FB
OBEXObjectPush	0x1105	0x0000110500001000800000805F9B34FB

The ProtocolDescriptorList attribute value has the most complicated structure of the four attributes shown in Table 7.1. The ProtocolDescriptorList describes how clients can connect to the service described by the service record. It lists the protocol stack needed to communicate with the service and any protocol-specific parameters needed to uniquely address the service. In the example shown in Table 7.1, a connection to this serial port service can be made by means of a stack of protocols that consists of the L2CAP layer and the RFCOMM layer. The implication is that the server application communicates directly with RFCOMM. Server channel 1 has been assigned to the server application by the Bluetooth stack, and this channel identifier is included in the service record so clients know the proper channel identifier to use to make a connection to the service.

The structure of the ProtocolDescriptorList is a list of lists with one sublist for every stack layer involved in the communications. So conceptually this looks like ((L2CAP), (RFCOMM, 1)), where parentheses are used as shorthand for a DataElement of type DATSEQ. The first element (L2CAP) indicates that L2CAP is the lowest protocol layer used to access this service. Strictly speaking, other Bluetooth stack protocols below L2CAP are involved, but stack layers below L2CAP are not included in SDP service records. The second element, (RFCOMM, 1), consists of two elements. The first element is the name of the next higher layer protocol, RFCOMM. The second element is a protocol-specific parameter, 1, which is the RFCOMM server channel identifier.

The list-of-lists structure is represented in the service record as an attribute value with structure

```
DataElement(type = DATSEQ,
            DataElement(type = DATSEQ, ...)
            DataElement(type = DATSEQ, ...))
```

Short-form UUIDs are used to represent the protocols L2CAP and RFCOMM. A DataElement of type U_INT_1 represents the server channel identifier 1 used by RFCOMM. This type of DataElement describes an unsigned integer of size 1 byte. In addition to the U_INT_1 type, there are DataElements types for signed and unsigned integers that are 1, 2, 4, 8, or 16 bytes long.

The third attribute shown in Table 7.1, ServiceName, has already been discussed. Its value is a DataElement of type String with value

“SPPEX.” The Bluetooth SDP specification defines the `ServiceName` attribute as a brief string representing the service suitable for display to the device user.

The fourth service attribute shown in Table 7.1 is the `ServiceRecordHandle`. This is a required attribute for every service record and plays an important role in the implementation of Bluetooth service discovery. However, the `ServiceRecordHandle` should be considered internal bookkeeping irrelevant to JABWT applications. JABWT applications may not modify this attribute. For this reason, the `ServiceRecordHandle` attribute is omitted from the rest of the tables describing service records in this book.

The SDP specification [1] describes the binary format used to transmit (attribute ID, attribute value) pairs from a service record over the air to another Bluetooth device. However, JABWT applications do not see that binary representation. Table 7.1 provides a representation of a service record that maps more directly onto the Java objects visible to JABWT applications. Table 7.1 refers to the JABWT classes `DataElement` and `UUID`.

Service Records for L2CAP

As an alternative to serial port communications using `bt spp`, JABWT server applications can communicate using L2CAP or OBEX. Servers that use these protocols need service records different from the one shown for `bt spp`. Again, the JABWT implementation is responsible for automatically creating the service records. L2CAP is described in Chapter 8. A server’s call to `Connector.open()` using the following `bt l2cap` connection string creates a service record like the one shown in Table 7.4:

```
"bt l2cap://localhost:BA661F1C148911D783C300B0D03D76EC;
name=An L2CAP Server"
```

There are several differences between Table 7.1 and Table 7.4:

- `SerialPort` has been removed from the `ServiceClassIDList`.
- `RFCOMM` has been removed from the `ProtocolDescriptorList`.
- The value of the Protocol/Service Multiplexer (PSM), `0x1001`, is included as an L2CAP parameter in the `ProtocolDescriptorList`.

Table 7.4 A Service Record Created by Connector.open("bt12cap://localhost:...")

ServiceClassIDList<0x0001>
DataElement(type = DATSEQ, DataElement(type = UUID, UUID(BA661F1C148911D783C300B0D03D76EC) —from the connection string))
ProtocolDescriptorList<0x0004>
DataElement(type = DATSEQ, DataElement(type = DATSEQ, DataElement(type = UUID, UUID(L2CAP<0x0100>)) DataElement(type = U_INT_2, 0x1001 —Protocol/Service Multiplexer. —Assigned by the stack; —filled in by JABWT)))
ServiceName<0x0100>
DataElement(type = STRING, "An L2CAP Server" —from "name=An L2CAP Server" —in the connection string)

Because L2CAP servers talk directly to the L2CAP layer in the Bluetooth stack, L2CAP is the last element in the ProtocolDescriptorList that describes the sequence of protocol layers that must be traversed to reach this server application. The protocol and service multiplexer parameter is required because L2CAP is a multiplexing layer, so multiple applications may be interacting with the L2CAP layer on the server device. The PSM value in the service record enables L2CAP to identify the particular application or protocol above L2CAP that will provide the service described by the service record. L2CAP uses the PSM value to set up an L2CAP channel to the correct application when a client connects to this service (see Chapter 8).

As was the case for the SPP, the JABWT implementation adds all the mandatory service attributes of the service record for L2CAP. The result is a minimal, but sufficient, service record.

Service Records for OBEX over RFCOMM

A third option for a Bluetooth server application is to use OBEX for communications. The following statement is an example of a connection string used for OBEX over RFCOMM:

```
"btgoep://localhost:0E18AE04148A11D7929B00B0D03D76EC;
    name=An OBEX Server"
```

The abbreviation goep in the protocol btgoep refers to the Generic Object Exchange Profile [3], which is the base Bluetooth profile shared by all of the Bluetooth OBEX profiles. The relation between OBEX and the GOEP profile is similar to the relation between the RFCOMM protocol and the SPP.

This call to `Connector.open("btgoep://localhost: ...")` creates a service record like the one shown in Table 7.5. There are only a few differences between this service record and the one shown in Table 7.1 for btsp. In Table 7.5 the service record contains OBEX as the last item in its `ProtocolDescriptorList`. This indicates that the server application talks directly to the OBEX layer of the Bluetooth stack. Another difference between Table 7.5 and Table 7.1 is that `ServiceClassIDList` in Table 7.5 does not include the `SerialPort` service class ID included for btsp.

Table 7.6 summarizes the three different protocols used by Bluetooth servers and the `Connector.open()` methods used to create service records for all three protocols. As shown in Table 7.6, the `Connector.open()` method is primarily defined in the CLDC specification [13]. Whereas CLDC provides the primary specification for the behavior of the `Connector.open()` method, the JABWT specification describes the valid url arguments for Bluetooth servers and the behavior of `Connector.open()` in creating Bluetooth service records.

Add the Service Record to the SDDb

Although the `Connector.open()` methods in Table 7.6 create a minimal service record for run-before-connect services, that service

Table 7.5 A Service Record Created by Connector.open("btgoep://localhost:...")

ServiceClassIDList<0x0001>
DataElement(type = DATSEQ, DataElement(type = UUID, UUID(0E18AE04148A11D7929B00B0D03D76EC) —from the connection string))
ProtocolDescriptorList<0x0004>
DataElement(type = DATSEQ, DataElement(type = DATSEQ, DataElement(type = UUID, UUID(L2CAP<0x0100>))) DataElement(type = DATSEQ, DataElement(type = UUID, UUID(RFCOMM<0x0003>)) DataElement(type = U_INT_1, 20 —server channel identifier)) DataElement(type = DATSEQ, DataElement(type = UUID, UUID(OBEX<0x0008>))))
ServiceName<0x0100>
DataElement(type = STRING, "An OBEX Server" —from "name=" in —the connection string)

record is not yet visible to client devices. The server that created the service record can access it and make modifications to it if desired. However, it is possible only for clients of run-before-connect services to connect after the server calls acceptAndOpen(). For this reason, the JABWT implementation adds a service record to the SDDb only the first time the server calls one of the acceptAndOpen() methods in Table 7.7.

Table 7.6 Methods That Create a Service Record

Protocol	Interface	Methods	Specification
btspp	Connector	open(url)	CLDC
		open(url, mode)	
		open(url, mode, timeout)	
btl2cap	Connector	open(url)	CLDC
		open(url, mode)	
		open(url, mode, timeout)	
btgoep	Connector	open(url)	CLDC
		open(url, mode)	
		open(url, mode, timeout)	

Once the service record is in the SDDB, client applications can discover that service record and attempt to connect to the server application.

A RFCOMM service can accept multiple connections from different clients by calling `acceptAndOpen()` repeatedly for the same notifier. Each client accesses the same service record and connects to the service using the same RFCOMM server channel. If the underlying Bluetooth system does not support multiple connections, then the implementation of `acceptAndOpen()` throws a `BluetoothStateException`. L2CAP and OBEX over RFCOMM services also can accept multiple clients.

A `ServiceRegistrationException` is thrown by all of the `acceptAndOpen()` methods in Table 7.7 if they fail to add a service record to the SDDB.

Remove the Service Record from the SDDB

Once the notifier associated with a run-before-connect service is closed, it is no longer possible to call `acceptAndOpen()` to accept another client connection. For this reason, the JABWT implementation removes the service record from the SDDB or disables the service record. Table 7.8 shows the different types of notifiers that add this behavior to the `close()` method inherited from the GCF interface `javax.microedition.io.Connection`.

Table 7.7 Methods That Add Service Records to the SDDB

Protocol	Interface	Methods	Specification
btsp	StreamConnectionNotifier	acceptAndOpen()	CLDC
btl2cap	L2CAPConnectionNotifier	acceptAndOpen()	JABWT
btgoep	SessionNotifier	acceptAndOpen(handler) acceptAndOpen(handler, authenticator)	JABWT

Table 7.8 Methods That Remove or Disable Service Records

Protocol	Interface	Methods	Specification
btsp	StreamConnectionNotifier	close()	CLDC
btl2cap	L2CAPConnectionNotifier	close()	CLDC
btgoep	SessionNotifier	close()	CLDC

7.2.4 Modifications to Service Records

In many cases, it is desirable to modify the service record created by the JABWT implementation. For example, if your service corresponds to a Bluetooth profile, you will have to modify the service record so that the record conforms to the requirements of the profile. Even if you are writing a custom application and are not required to have a standardized service record, you may want to modify your service record to provide various kinds of useful information to potential clients. Many optional attributes are defined in Part E of the Bluetooth SDP specification [1] that server applications can use to describe the properties of their service. It is also possible to add application-specific, user-defined attributes to the service record that are not defined by the Bluetooth specification.

Figure 7.3 adds JABWT methods for modifying service records to the sequence diagram shown in Figure 7.2. The LocalDevice class provides a getRecord() method that a server application can use to obtain

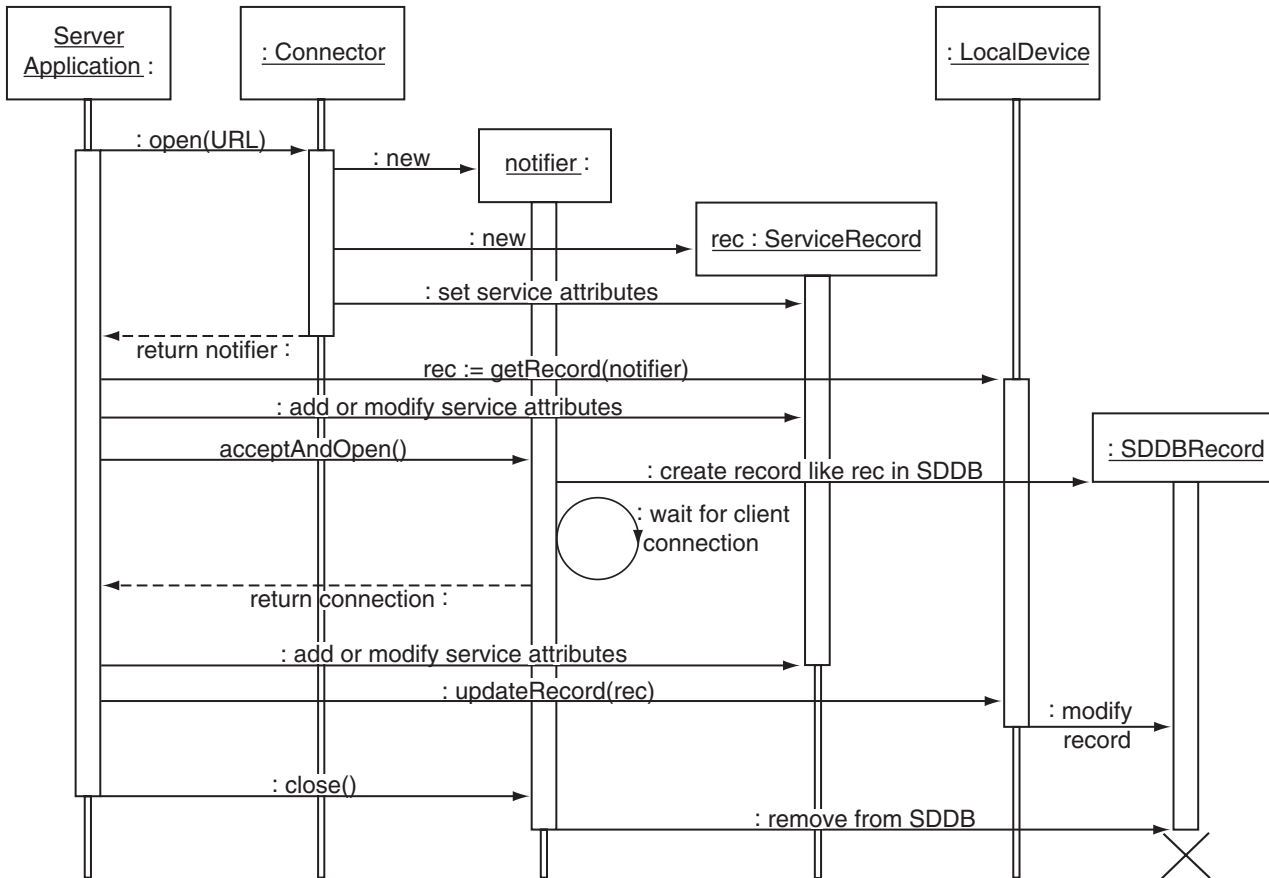


Figure 7.3 Example of server modifying its service record.

its `ServiceRecord`. The server can modify the `ServiceRecord` object by adding or modifying attributes using `ServiceRecord.setAttributeValue()`. As shown in Figure 7.3, any modifications the server application makes to its `ServiceRecord` before calling `acceptAndOpen()` for the first time will be reflected in the service record added to the SDDb by `acceptAndOpen()`.

Any changes made to the service record object by a JABWT application after the first call to `acceptAndOpen()` are not reflected in the service record in the SDDb seen by clients. This is because the service record in the SDDb is essentially a copy of the service record Java object at the time of the first call to `acceptAndOpen()`. To modify service records already in the SDDb, JABWT provides the instance method `LocalDevice.updateRecord(serviceRecord)`.

7.2.5 Device Service Classes

As described in Chapter 6, clients can consult the `DeviceClass` of any device they discover to determine what kind of device has been found (e.g., phone, PDA, or PC). The `DeviceClass` also indicates the major service classes offered by the discovered device (e.g., rendering, telephony, or information). This means there are two different ways in which a server application describes the service it offers:

- By adding a service record to the SDDb
- By activating major service class bits in the `DeviceClass`

The server application can use the `setDeviceServiceClasses()` method of `ServiceRecord` to turn on some of the service class bits of the device to reflect the new service being offered. A server application is not required to use the `setDeviceServiceClasses()` method. However, it is recommended that a server use the method to describe its service in terms of the major service classes. Keeping the major service classes up to date reduces the likelihood that clients will erroneously skip over this device when looking for a service.

The `close()` message also causes the JABWT implementation to deactivate any service class bits that were activated by `setDeviceServiceClasses()`, unless another service whose notifier is not yet closed also activated some of the same bits.

7.3 Programming with the API

The programming examples in this chapter are divided into examples of service registration and examples of service discovery. Sections 7.3.1 through 7.3.4 provide examples of service registration. These sections show examples of the use of methods for creating and modifying service records. The examples in these sections are all server applications. These servers simply create a service record and add it to the SDDb. No client applications are needed to illustrate this behavior. Not all of the code needed to produce a running application is presented in the text. The complete code is available in Appendix A. Sections 7.3.5 through 7.3.8 provide examples of service discovery. These sections extend the `DiscoveryMIDlet` that was introduced in Chapter 6 to discover various aspects of the service defined in Section 7.3.2.

7.3.1 Automatic Generation of Service Records

In this first example, the server application makes no modifications to the service record. This is the simplest case. Figure 7.4 shows the output produced by the `DefaultBtsppRecordMIDlet`. The display shows the connection string that clients can use to connect to this server:

```
btspp://0080375a0000:1
```

The display also shows that the service record for the server has five service attributes and lists their attribute IDs as hex numbers.

The `DefaultBtsppRecordMIDlet` implements the `Runnable` interface. The `run()` method first calls the method `askToBeGeneralDiscoverable()` defined in the `DefaultBtsppRecordServer` class to attempt to make the server device general discoverable. The `run()` method calls the method `defineDefaultBtsppService()` to create the service record and define the `StreamConnectionNotifier`. The new service record is obtained from the `LocalDevice`, and a brief description of the service record is appended to a display Form, output. Finally, the `run()` method calls the `acceptClientConnections()` method defined in the `DefaultBtsppRecordServer` class. This method adds the service record to the SDDb and waits for client connections.



Figure 7.4 Example Code Displays Information about the Default Service Record (emulation only).

```
public class DefaultBtsppRecordMIDlet extends MIDlet
    implements Runnable, CommandListener {
    StreamConnectionNotifier notifier;
    /* The form displayed to the user. */
    private Form output;
    ...
    public void run() {
        LocalDevice theRadio;
        // Define the serial port service and create the notifier
        try {
            theRadio = LocalDevice.getLocalDevice();
```

```

        server = new DefaultBtsppRecordServer();
        server.askToBeGeneralDiscoverable(theRadio);
        notifier = server.defineDefaultBtsppService();
    } catch (IOException e) {
        output.append("Unable to start server (IOException: " +
            e.getMessage() + ")");
        return;
    }
    if (notifier != null) {
        ServiceRecord record = theRadio.getRecord(notifier);
        output.append("URL=" + server.getURL(record));
        output.append(server.describeAttributes(record));
    } else {
        output.append("Unable to start server");
        return;
    }
    // Use the notifier to establish serial port connections
    server.acceptClientConnections(notifier);
}
}

```

Now that we have seen the overall flow of execution defined by the `DefaultBtsppRecordMIDlet`, we will examine the `DefaultBtsppRecordServer` class. The `askToBeGeneralDiscoverable()` method uses the `setDiscoverable()` method to request that the device be made general discoverable. This enables client devices that do device discovery with the GIAC mode to find the server device. If `setDiscoverable()` returns false, indicating that the request was not granted, or if it throws an exception, the server just proceeds. Any clients that know the Bluetooth address for this server can access this service even if the device is not discoverable. For example, clients that include the server device among their pre-known devices can access the server (see Chapter 6).

The `defineDefaultBtsppService()` method calls `Connector.open(connString)` to create a `StreamConnectionNotifier`. That same call also creates a default btspp service record such as the one shown in Table 7.1 and associates it with the notifier.

```

public class DefaultBtsppRecordServer {
    boolean stop = false;

```

```

void askToBeGeneralDiscoverable(LocalDevice dev) {
    try {
        /* Request that the device be made discoverable */
        dev.setDiscoverable(DiscoveryAgent.GIAC);
    } catch (BluetoothStateException ignore) {
        /* discoverable is not an absolute requirement */
    }
}

public StreamConnectionNotifier
defineDefaultBtsppService() {
    StreamConnectionNotifier notifier;

    String connString =
        "btspp://localhost:" +
        "68EE141812D211D78EED00B0D03D76EC;" +
        "name=SPPEX";
    try {
        notifier =
            (StreamConnectionNotifier)
            Connector.open(connString);
    } catch (IOException e){
        return null;
    }
    return notifier;
}

public String getURL(ServiceRecord record) {
    String url =
        record.getConnectionURL(
            ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
            false);
    if (url != null) {
        return url.substring(0, url.indexOf(";"));
    } else {
        return "getConnectionURL()==null";
    }
}

public String describeAttributes(ServiceRecord
record) {
    int[] attributeIDs = record.getAttributeIDs();
    StringBuffer strBuf = new StringBuffer(100);
    strBuf.append("\n").append(Integer.toString(
        attributeIDs.length));
}

```

```
        strBuf.append(" Attributes: ");
        for (int i = 0; i < attributeIDs.length; i++) {
            strBuf.append("<0x");
            strBuf.append(Integer.toHexString(attributeIDs[i]));
            strBuf.append(">\n");
        }
        return strBuf.toString();
    }

    public void acceptClientConnections(
        StreamConnectionNotifier notifier) {
        if (notifier == null) {
            return;
        }
        try {
            while (!stop) {
                StreamConnection clientConn = null;
                /*
                 * acceptAndOpen() waits for the next client to
                 * connect to this service. The first time through
                 * the
                 * loop, acceptAndOpen() adds the service record to
                 * the SDDb and updates the service class bits
                 * of the device.
                 */
                try {
                    clientConn =
                        (StreamConnection)notifier.acceptAndOpen();
                } catch (ServiceRegistrationException e1) {
                } catch (IOException e) {
                    continue;
                }
                /*
                 * Code to communicate to a client over clientConn
                 * would go here.
                 */
            }
        } finally {
            try {
                shutdown(notifier);
            } catch (IOException ignore) {
            }
        }
    }
}
```

```

    }

    public void shutdown(StreamConnectionNotifier notifier)
        throws IOException {
        stop = true;
        notifier.close();
    }
}

```

The `getURL()` method returns a connection string that clients can use to connect to the `DefaultBtsppRecordServer`. The `getURL()` method calls the JABWT `getConnectionURL()` method to get the connection string, and then the string is shortened for display by removing the parameter list. As shown in Figure 7.4, the result is `btsp://0080375a0000:1`, where `0080375a0000` is the Bluetooth address of the local device, and `1` is the server channel identifier. Typically, clients send the `getConnectionURL()` message to a service record obtained during service discovery to obtain a connection string to connect to that service. Here we send the same message to the server's own service record to obtain the connection string for display by the `DefaultBtsppRecordMIDlet`.

The `describeAttributes()` method uses the JABWT method `getAttributesIDs()` to obtain an array of the attribute IDs that are part of the new service record. The `describeAttributes()` method returns a string that includes the number of attributes in this array and the hexadecimal values of the attribute IDs. The `DefaultBtsppRecordMIDlet` displays this string on the user interface. These attribute IDs can be compared with the ones shown for the default `btsp` service record in Table 7.1. The `DefaultBtsppRecordMIDlet` displays the attribute IDs in Table 7.1. (Some JSR-82 implementations do not return a `ServiceRecordHandle`, `0x0000`.) One additional attribute, `ServiceRecordState 0x002`, might also be displayed. The Bluetooth stack may add the `ServiceRecordState` attribute to a service record to make it easier for clients to determine whether that service record has changed. If the value of the `ServiceRecordState` attribute has not changed since the last time it was checked, the client knows that none of the attributes in the service record have changed.

The last method defined in `DefaultBtsppRecordServer` is `acceptClientConnections()`. This method calls `acceptAndOpen()`, which adds the service record to the SDDb, where it will be visible to

clients. The `acceptAndOpen()` method then blocks and waits for a client to connect. Once a client makes a connection, the `acceptAndOpen()` method returns a `StreamConnection` that the server can use to communicate with that client using RFCOMM (see Chapter 4).

7.3.2 Modifying a Service Record

This section illustrates how a server can modify its service record by adding additional service attributes. Suppose we want to create the service record shown in Table 7.9 for a two-person Bluetooth game. The JABWT implementation automatically adds the first three attributes shown in Table 7.9 when it creates the service record. The last three attributes must be added by the server application. Two of the service attributes added, `ServiceDescription` and `DocumentationURL`, are standard attributes defined in the SDP specification [1, Part E]. A `ServiceDescription` is a brief description of the service (fewer than 200 characters). The `DocumentationURL` provides a pointer to a Web page for detailed documentation of the service. The third attribute added, `0x2222`, is a nonstandard, application-specific service attribute. This attribute shows the highest score achieved to date by the user of this device. Clients might use this attribute to select a suitable opponent for the game or to assign handicaps.

The `defineGameService()` method shown below illustrates how the service record shown in Table 7.9 can be created by a server application. The statement

```
notifier = (StreamConnectionNotifier)
Connector.open(connString)
```

creates the service record and returns a notifier cast to a `StreamConnectionNotifier`. The notifier is used to access the new service record by the statement

```
ServiceRecord record = localDev.getRecord(notifier);
```

The `defineGameService()` method then adds three additional service attributes to the service record before that record is made visible to clients. The method `setAttributeValue()` is used to add each attribute to the service record.

Table 7.9 The Service Record for a Bluetooth Game

ServiceClassIDList<0x0001>
DataElement(type = DATSEQ, DataElement(type = UUID, UUID(0FA1A7AC16A211D7854400B0D03D76EC)) DataElement(type = UUID, UUID(SerialPort<0x1101>)))
ProtocolDescriptorList<0x0004>
DataElement(type = DATSEQ, DataElement(type = DATSEQ, DataElement(type = UUID, UUID(L2CAP<0x0100>))) DataElement(type = DATSEQ, DataElement(type = UUID, UUID(RFCOMM<0x0003>)) DataElement(type = U_INT_1, 3 –server channel identifier)))
ServiceName<0x0100>
DataElement(type = STRING, "A Bluetooth Game")
ServiceDescription<0x0101>
DataElement(type = STRING, "This game is fun! It is for two people. You can play it on your cell phones.")
DocumentationURL<0x000A>
–Where to find documentation

<pre> DataElement(type = URL, "http://www.gameDocsOnSomeWebPage.com") </pre>
<pre> <0x2222> -An application-specific attribute for the highest score in the game </pre>
<pre> DataElement(type = U_INT_4, 10000) </pre>

```

public StreamConnectionNotifier
defineGameService(LocalDevice localDev, long highScore) {
    StreamConnectionNotifier notifier;
    String connString =
        "btspp://localhost:0FA1A7AC16A211D7854400B0D03D76EC;" +
        "name=A Bluetooth Game";
    try {
        notifier =
            (StreamConnectionNotifier)Connector.open(connString);
    } catch (IOException e2){
        return null;
    }
    ServiceRecord record = localDev.getRecord(notifier);
    // Add optional ServiceDescription attribute; attribute
    // ID 0x0101.
    record.setAttributeValue(0x0101,
        new DataElement(DataElement.STRING,
            "This game is fun! It " +
            "is for two people. " +
            "You can play it on " +
            "your cell phones."));
    // Add optional DocumentationURL attribute; attribute ID
    // 0x000A.
    record.setAttributeValue(0x000A,
        new DataElement(DataElement.URL,
            "http://" +
            "www.gameDocsOnSomeWebpage.com"));
    /*

```

```

    * Add an application-specific attribute for the highest
    * score achieved by this player to date.
    */
    record.setAttributeValue(0x2222,
        new DataElement(DataElement.U_INT_4,
            highScore));
    return notifier;
}

```

When the server does `acceptAndOpen()`, the service record added to the SDDb has the additional service attributes added by the `defineGameService()` method. When using application-specific service attributes, keep in mind that the Bluetooth specification reserves certain attribute ID values. Attribute IDs in the range 0x000D to 0x01FF are reserved and should not be used.

7.3.3 Support for String Attributes in Several Languages

Table 7.10 shows how Bluetooth service records can include strings in more than one language. In addition to the `ServiceName` and `ServiceDescription` attributes shown in Table 7.9, three service attributes have been included in the service record. Two of these attributes provide the `ServiceName` and `ServiceDescription` in French. The third attribute, the `LanguageBaseAttributeIDList`, describes the two languages used in this service record and provides the information needed to differentiate the English strings from the French strings.

To support use of multiple languages in service records, the Bluetooth SDP uses a base-plus-offset scheme for all service attributes of type string. In the service record shown in Table 7.10, the base for English service attributes is 0x0100. The base for French service attributes is 0x0120. The SDP specification [1] defines the `ServiceDescription` as having an offset of 0x0001. This means the attribute ID of the `ServiceDescription` in English in this service record is given by the English base plus offset; or

$$0x0100 + 0x0001 = 0x0101$$

The attribute ID of the `ServiceDescription` in French is given by the French base plus offset, or

$$0x0120 + 0x0001 = 0x0121$$

ServiceName is defined as having an offset of 0x0000, so the ServiceName in English has attribute ID

$$0x0100 + 0x0000 = 0x0100$$

The ServiceName in French has attribute ID

$$0x0120 + 0x0000 = 0x0120$$

The LanguageBaseAttributeIDList contains the attribute ID base for each language. LanguageBaseAttributeIDList is an optional service attribute. However, if a service record were to use more than one language, it would be very difficult for applications to use the other languages without knowing the attribute ID base value for the other languages. The attributeValue for the LanguageBaseAttributeIDList is a list, or DATSEQ, of DataElements of type U_INT_2, that is, of type unsigned 2-byte integer.

The elements of the LanguageBaseAttributeIDList are implicitly grouped into triplets, where each triplet is for a particular language. The first element of the triplet is the language code as standardized by ISO 639 [32]. For English, this code is “en”; for French, it is “fr.” The second element of the triplet is the character encoding used for the language. Unicode Transformation Format 8 (UTF-8) is an example of character encoding. The Internet Assigned Numbers Authority maintains a standard list of character encodings [33]. Each encoding in the standard has a MIBenum value; for example, UTF-8 is 0x006A (decimal 106). The MIBenum value of the character encoding is the second element in each triplet contained in the LanguageBaseAttributeIDList. The third element of each triplet is the attribute ID base value for the triplet’s language.

The base values used in the third element of each triplet are not standardized. The service records in an SDDb may use different base values for the same language. The only rules are the following:

- The attribute ID base for the primary language used in a particular service record must be 0x0100.
- If there is a LanguageBaseAttributeIDList in the service record, the first language in this list must use the attribute ID base 0x0100; that is, the first language must be the primary language.

The attribute ID base used for a second language is not standardized; that is, we chose the value of 0x0120 for the base value for French in Table 7.10. In selection of base values, care must be taken to avoid

Table 7.10 Selected Attributes from a Service Record with English and French Strings.

...
ServiceName<0x0100> –Name of the service in the primary language of the service record
DataElement(type = STRING, "A Bluetooth Game" –from "name=" in the connection string)
ServiceDescription<0x0101> –Description of the service in the primary language
DataElement(type = STRING, "This game is fun! It is for two people. You can play it on your cell phones.")
ServiceName<0x0120> –Name of the service in French
DataElement(type = STRING, "Jeu de Bluetooth")
ServiceDescription<0x0121> –Description of the service in French
DataElement(type = STRING, "Ce jeu est amusant ! Il se joue à deux. Vous pouvez y jouer sur vos téléphones mobiles.")
LanguageBaseAttributeIDList<0x0006> –Describe the languages used in the service record
DataElement(type = DATSEQ, DataElement(type = U_INT_2, 0x656E –ASCII for "en", English) DataElement(type = U_INT_2, 0x006A –the MIBenum for UTF-8)

```

DataElement(type = U_INT_2,
            0x0100 –attribute ID base for English)
DataElement(type = U_INT_2,
            0x6672 –ASCII for "fr", French)
DataElement(type = U_INT_2,
            0x006A –the MIBenum for UTF-8)
DataElement(type = U_INT_2,
            0x0120 –attribute ID base for French))

```

conflicts with the attribute IDs defined by the Bluetooth Assigned Numbers. One recommendation is to choose attribute ID base values so that the sum of base value plus offset falls either in the range 0x0100 to 0x01FF or in the range 0x0900 to 0xFFFF. The same recommendation applies to user-defined, service-specific attributes with string values.

7.3.4 Service Records for Bluetooth Profiles

The Bluetooth Profiles specification [3] describes a number of common tasks that will be accomplished with Bluetooth wireless technology. The profiles document lists requirements that help in achieving interoperability between devices with independent implementations of these standardized tasks. Developers who intend to implement one of these Bluetooth profiles with their JABWT application need to study the specification [3] closely so that their applications can pass any Bluetooth qualification tests for the profile and can successfully interoperate with other devices that also support the profile.

The profile specifications place requirements on both client and server roles for each profile. Part of the requirements for servers is a specification of what the service record will look like for the profile service. As an example, Table 7.11 shows the service record for the Object Push Profile from the Bluetooth profiles specification [3]. This format is followed by all of the Bluetooth profiles.

The Object Push Profile describes how electronic business cards and other similar objects can be transmitted between Bluetooth devices by means of OBEX. The server in the Object Push Profile is called a Push Server, and the client is called a Push Client. As Table 7.11 shows, the

Table 7.11 Service Record Defined by the Bluetooth Object Push Profile

Item	Definition	Type	Value	AttrID	Status	Default Value
ServiceClassIDList				0x0001	Mand.	
ServiceClass #0	UUID for OBEXObjectPush"	UUID	0x1105		Mand.	0x1105
ProtocolDescriptorList				0x0004	Mand.	
Protocol ID #0	UUID for L2CAP protocol	UUID	0x0100		Mand.	0x0100
Protocol ID #1	UUID for RFCOMM protocol	UUID	0x0003		Mand.	0x0003
Parameter #0	Server channel	UINT8	varies		Mand.	varies
Protocol ID #2	UUID for OBEX	UUID	0x0008		Mand.	0x0008
ServiceName	Displayable Text name	String	varies	0x0000 + base	Opt.	"OBEX Object Push"
BluetoothProfile				0x0009	Opt.	
DescriptorList						
Profile ID #0	Supported profile	UUID	0x1105			0x1105
Version #0	Profile version	UINT16	0x0100			0x0100
Supported Formats List	Supported Formats List	Data Element Sequence of UINT8	Formats: 0x01 = vCard 2.1 0x02 = vCard 3.0 0x03 = vCal 1.0 0x04 = iCal 2.0 0x05 = vNote 0x06 = vMessage 0xFF = any type of object.	0x0303	Mand.	

UUID with short form 0x1105 is defined as the Service Class ID for Push Servers that conform to the Object Push Profile. Push Clients can look for service records that contain this UUID to identify Push Servers that have demonstrated they meet the requirements of the Object Push Profile.

The Status column in Table 7.11 indicates service record entries that are mandatory (Mand.) and entries that are optional (Opt.) according to the Bluetooth specification.

The UUID values in the Value and Default Value columns shown in Table 7.11 are obtained from the Bluetooth Assigned Numbers [30]. The service attribute ID values in the AttrID column also are obtained from the Bluetooth Assigned Numbers. The version of Table 7.11 in the Object Push Profile specification does not contain these values explicitly but instead refers to the assigned numbers.

The service record in Table 7.11 can be translated into the same notation used earlier in this chapter to describe the `bt spp`, `bt l2cap`, and `bt goep` service records. That translation is shown in Table 7.12. The translation is straightforward for the most part. However, the two notations use different units (bits versus bytes) when describing types for numbers. Table 7.11 from the Bluetooth profiles specification uses `Uint8` and `Uint16`, which refer to unsigned 8-bit and 16-bit integers. Table 7.12, on the other hand, uses `U_INT_1` and `U_INT_2` for these same two quantities. Table 7.12 uses the names of JABWT constants for all the “type =” entries. The `javax.bluetooth.DataElement` constants `U_INT_1` and `U_INT_2` refer to 1-byte and 2-byte integers. The representations in Table 7.11 and Table 7.12 are equivalent but use different units (bits versus bytes) when describing the type of a number.

There are several points to observe about Table 7.12. In addition to the familiar attributes `ServiceClassIDList`, `ProtocolDescriptorList`, and `ServiceName` are several attributes that we have not seen before. Table 7.12 includes a `BluetoothProfileDescriptorList`, which is an attribute used to declare that this service conforms to version number 1.00 of the Object Push Profile. The service records for Bluetooth profiles commonly include a `BluetoothProfileDescriptorList` attribute, although that attribute usually is optional. The short-form UUID for OBEXObjectPush, 0x1105, is used in the `BluetoothProfileDescriptorList` to designate the Object Push Profile. Because this UUID also is used in the

Table 7.12 Service Record for an OBEX Object Push Server

ServiceClassIDList<0x0001>
DataElement(type = DATSEQ, DataElement(type = UUID, UUID(OBEXObjectPush<0x1105>) —Object Push UUID))
ProtocolDescriptorList<0x0004>
DataElement(type = DATSEQ, DataElement(type = DATSEQ, DataElement(type = UUID, UUID(L2CAP<0x0100>))) DataElement(type = DATSEQ, DataElement(type = UUID, UUID(RFCOMM<0x0003>)) DataElement(type = U_INT_1, 4 –server channel identifier)) DataElement(type = DATSEQ, DataElement(type = UUID, UUID(OBEX<0x0008>)))
ServiceName<0x0100>
DataElement(type = STRING, "OBEX Object Push Server")
BluetoothProfileDescriptorList<0x0009>
DataElement(type = DATSEQ, DataElement(type = DATSEQ, DataElement(type = UUID, UUID(OBEXObjectPush<0x1105>)) DataElement(type = U_INT_2, 0x0100 – version 1.00)))

Supported Formats List<0x0303>

```
DataElement(type = DATSEQ,
    DataElement(type = U_INT_1,
        0x01 – vCard 2.1)
    DataElement(type = U_INT_1,
        0x02 – vCard 3.0))
```

ServiceClassIDList, this marks the second appearance of this same UUID in the service record.

Table 7.12 also includes one attribute, Supported Formats List, that is unique to the Object Push Profile. The Supported Formats List attribute describes the different object formats supported by this Push Server. The Supported Formats List entry in Table 7.11 describes the different object formats recognized by the Object Push Profile. In Table 7.12, the two formats for electronic business cards have been specified. The Object Push Profile requires that Phone Book applications support data exchange using the vCard 2.1 content format [34].

The following example code shows how a server application can create an Object Push service record. After `Connector.open()` is used to create the service record, the `defineObjectPushService()` method adds the `BluetoothProfileDescriptorList` attribute to the service record by calling the application method `setBluetoothProfileList()`. The code for this method is presented later. It is followed by code that adds the Supported Formats List attribute to the service record to declare that this server understands the vCard 2.1 and 3.0 formats.

```
/**
 * Create the service record for an OBEX Object Push
 * server as defined
 * by the Bluetooth Object Push profile.
 */
public SessionNotifier defineObjectPushService() {
    SessionNotifier notifier;
    // The UUID 000011050000... is the long-form UUID for the
    // short form 0x1105 defined for the Object Push
    // service ID by
    // assigned numbers.
```

```
String connString =
    "btgoep://localhost:" +
        "0000110500001000800000805F9B34FB;" +
        "name=OBEX Object Push Server";
// Connector.open() assigns a RFCOMM server channel
// and creates a service record using this channel.
try {
    notifier =
        (SessionNotifier)Connector.open(connString);
} catch (ServiceRegistrationException e1) {
    // The open method failed because unable to obtain
    // an RFCOMM
    // server channel.
    return null;
} catch (IOException e2){
    // The open method failed due to another IOException
    return null;
}
try {
    localDev = LocalDevice.getLocalDevice();
} catch (BluetoothStateException e) {
    return null;
}
ServiceRecord record = localDev.getRecord(notifier);
// Add the optional service attribute
// BluetoothProfileDescriptorList
ServiceRecordUtilities.setBluetoothProfileList(record,
    0x1105, 0x0100);
DataElement objFormatsDE = new
    DataElement(DataElement.DATSEQ);
// supported format 0x01 = vCard 2.1
objFormatsDE.addElement(new
    DataElement(DataElement.U_INT_1, 0x01));
// supported format 0x02 = vCard 3.0
objFormatsDE.addElement(new
    DataElement(DataElement.U_INT_1, 0x02));
// Add mandatory Supported Formats List, attribute ID
// 0x0303
record.setAttributeValue(0x0303, objFormatsDE);
```

```

// An Object Push Server provides an Object Transfer
// service.
// Bit 20 of the Class of Device is for Object Transfer.
record.setDeviceServiceClasses(0x100000);
return notifier;
}

```

Table 7.12 shows that the Object Push service record has the following ServiceClassIDList:

```

DataElement(type = DATSEQ,
             DataElement(type = UUID,
                         UUID(OBEXObjectPush<0x1105>)
                         —Object Push UUID))

```

This step is accomplished by the method `defineObjectPushService()` by using the connection string:

```

String connString =
    "btgoep://localhost:" +
    "0000110500001000800000805F9B34FB;" +
    "name=OBEX Object Push Server";

```

The JABWT implementation inserts the UUID from the connection string into the ServiceClassIDList of the service record. Although there is a short form UUID of 0x1105 for the OBEXObjectPush service class, we have used the long form in the connection string. The reason is that a UUID in a connection string is always interpreted as a 128-bit UUID. This means that 1105 would be interpreted as the 128-bit UUID 00000000000000000000000000001105 not as the 128-bit UUID 0000110500001000800000805F9B34FB.

In the example code, the `defineObjectPushService()` method uses the `setDeviceServiceClasses()` method of the `ServiceRecord` interface to describe the object transfer major service class provided by the server application

```

record.setDeviceServiceClasses(0x100000);

```

The specification of the Object Push Profile requires that the Push Server indicate that it offers this object transfer service in its device class. A server uses the `setDeviceServiceClasses()` method to associate the `ServiceRecord` with all of the major service classes that describe that service. Later, when a server first calls `acceptAndOpen()`, both its

service record and its major service class bits are made visible to client devices. In the case of the major service classes, `acceptAndOpen()` performs a logical OR of the current settings of the service class bits of the device with the major service classes declared by the `setDeviceServiceClasses()` method. This OR operation might activate additional service class bits that indicate new capabilities for the device.

The `defineObjectPushService()` method makes a static method call to create the `BluetoothProfileDescriptorList`. The code for that static method is shown below. It creates the `DataElement` structure required for the `BluetoothProfileDescriptorList` attribute by the Object Push Profile. This method uses the `DataElement.addElement()` method to assemble `DataElements` of type `DATSEQ`. One `DataElement` of type `DATSEQ` is used for the pair of profile and version number. Another `DataElement` of type `DATSEQ` is used for the list of pairs. In this case, there is only one pair in the list of pairs (Table 7.12).

```
public class ServiceRecordUtilities {
    public static final int ATT_ID_BLUETOOTH_PROFILE_LIST =
        0x0009;
    /**
     * Sets the value of the BluetoothProfileDescriptorList
     * attribute to
     * be the profile represented by a short-form UUID value
     * and version number.
     * @param record The service record to be modified
     * @param profileUuidValue The short-form UUID for the
     * profile from the Bluetooth Assigned Numbers
     * @param version The version of the profile this
     * service conforms to.
     * The format is 0xMMmm where MM is the
     * major version
     * number and mm is the minor version number.
     */
    public static void setBluetoothProfileList(
        ServiceRecord record,
        long profileUuidValue,
        int version) {
        UUID profileUuid = new UUID(profileUuidValue);
```

```

        DataElement profileUuidDE = new
            DataElement(DataElement.UUID,
                profileUuid);
        DataElement versionDE = new
            DataElement(DataElement.U_INT_2, version);
        DataElement profileVersionPairDE
            = new DataElement(DataElement.DATSEQ);
        DataElement profileDescriptorDE = new
            DataElement(DataElement.DATSEQ);
        // Create a pair with profile UUID and profile version
        profileVersionPairDE.addElement(profileUuidDE);
        profileVersionPairDE.addElement(versionDE);
        // Add the pair to the list of profiles
        profileDescriptorDE.addElement(profileVersionPairDE);
        // Set the BluetoothProfileDescriptorList to a DATSEQ data
        // element containing the UUID-version pair for this
        // profile.
        record.setAttributeValue(ATT_ID_BLUETOOTH_PROFILE_LIST,
            profileDescriptorDE);
    }
}

```

7.3.5 Service Discovery

Service discovery within Bluetooth wireless technology can be as complicated or as simple as desired. Like an inquiry, service discovery is a non-blocking request. As service records are discovered, they are passed to the application as events. Also like an inquiry, an event occurs at the end of the service search to notify the application that the service search has been completed. Unlike an inquiry, many devices support multiple service searches at any one time. The number of service searches that the local device supports can be retrieved via a Bluetooth device property. The “bluetooth.sd.trans.max” property can be retrieved via the `LocalDevice.getProperty()` method to determine the maximum number of concurrent service searches.

For the local device to search for a service on a remote device, the local device sends a list of UUIDs to search for to the remote device. The remote device checks all the service records on it for all of the UUIDs sent to it. For every service record that has all the UUIDs, the remote

device sends back the `ServiceRecordHandle` and the requested attributes for that service record.

Starting a service search on a remote device begins with the `DiscoveryAgent.searchServices()` method. The `searchServices()` method takes four arguments. The first argument is the list of attributes to retrieve in any service record that meets the other search criteria. By default, the `searchServices()` method retrieves the `ServiceRecordHandle`, `ServiceClassIDList`, `ServiceRecordState`, `ServiceID`, and `ProtocolDescriptorList` attributes. These attributes are known as the *default attributes*. If the attributes list argument is null, only the default attributes are retrieved. If a list of attributes is provided, the default attributes are retrieved in addition to the list provided. With the default attributes, the application has enough information to establish a connection to the service. Additional attributes may be retrieved if additional information about the service is needed. The second argument, the list of UUIDs to search for, specifies all the UUIDs that must exist in a service to be retrieved. The more complete this list of UUIDs, the less likely it is that a service record will contain all of these UUIDs. The third argument, the remote device to search, is a `RemoteDevice` object received via an inquiry or a call to `retrieveDevices()`. The final argument is the `DiscoveryListener` object that will be notified when the services are discovered.

The `searchServices()` method returns the transaction ID for the service search if the device is able to start the service search. The transaction ID allows an application to cancel the search, identify which search located a service, and determine when a specific search is completed. The `searchServices()` method may throw a `BluetoothStateException` if the local device has reached the maximum number of service searches or if the current service search could not be started.

As services are discovered, they are sent to the `DiscoveryListener` via the `servicesDiscovered()` method. The transaction ID of the service search along with all the service records found during the search also is provided. The `servicesDiscovered()` method can be called multiple times for a single service search request. The service records are returned as an array of `ServiceRecord` objects. Each of these `ServiceRecord` objects contains all the attributes requested in the call to `searchServices()` along with the default attributes.

Table 7.13 Status Codes for Service Searches

Completion Status	Reason
SERVICE_SEARCH_COMPLETED	At least one service record was found and the search completed normally.
SERVICE_SEARCH_TERMINATED	The service search was canceled by a call to <code>cancelServiceSearch()</code> .
SERVICE_SEARCH_ERROR	An error occurred during the service search.
SERVICE_SEARCH_NO_RECORDS	No records were found during the service search.
SERVICE_SEARCH_DEVICE_NOT_REACHABLE	The <code>RemoteDevice</code> specified to <code>searchServices()</code> could not be reached (i.e., a connection could not be established to the remote device).

When the service search is completed, the `serviceSearchCompleted()` method is called. The `serviceSearchCompleted()` method provides the transaction ID of the search that ended and a completion status code. Table 7.13 lists all the completion status codes and what those codes mean.

Like an inquiry, a service search can be canceled with the `cancelServiceSearch()` method. The `cancelServiceSearch()` method takes as an argument the transaction ID of the service search to cancel. The method returns true if the search was canceled. Canceling the search also causes a `serviceSearchCompleted()` event to occur with the `SERVICE_SEARCH_TERMINATED` status code. If the method returns false, either the service search has already ended or the transaction ID is not valid.

Returning to the `DiscoveryMIDlet` introduced in Chapter 6, the next step is to search for services. The `DiscoveryMIDlet` will be modified to search for the Bluetooth game service defined earlier in this chapter. To determine which device to search, the `DiscoveryMIDlet` waits until the user selects a Bluetooth device from a `List`. The `DiscoveryMIDlet` then searches the device specified for the UUID defined by the Bluetooth game service. After retrieving all the services that use this UUID, the `DiscoveryMIDlet` displays the name of each service. Before starting the service search, the `DiscoveryMIDlet` must

be modified to maintain a list of the `RemoteDevice` objects found via device discovery and to keep track of when the device is in an inquiry.

```
public class DiscoveryMIDlet extends BluetoothMIDlet
    implements DiscoveryListener {
    ...
    /**
     * Keeps track of the RemoteDevice objects.
     */
    private Vector deviceVector;
    /**
     * Specifies if an inquiry is currently occurring.
     */
    private boolean isInInquiry;
    ...
    public void startApp() throws MIDletStateChangeException {
        isInInquiry = false;
        // Create a new List and set it to the current
        // displayable
        deviceList = new List("List of Devices", List.IMPLICIT);
        deviceList.addCommand(new Command("Exit",
            Command.EXIT, 1));
        deviceList.setCommandListener(this);
        Display.getDisplay(this).setCurrent(deviceList);
        // Retrieve the DiscoveryAgent object. If the
        // retrieving the
        // local device causes a BluetoothStateException,
        // something is wrong
        // so stop the app from running.
        try {
            LocalDevice local = LocalDevice.getLocalDevice();
            agent = local.getDiscoveryAgent();
        } catch (BluetoothStateException e) {
            throw new MIDletStateChangeException(
                "Unable to retrieve local Bluetooth device.");
        }
        deviceVector = new Vector();
        addDevices();
    }
}
```

```

        try {
            agent.startInquiry(DiscoveryAgent.GIAC, this);
        } catch (BluetoothStateException e) {
            throw new MIDletStateChangeException(
                "Unable to start the inquiry");
        }

        isInInquiry = true;
    }

    public void deviceDiscovered(RemoteDevice device,
        DeviceClass cod) {
        String address = device.getBluetoothAddress();
        deviceList.insert(0, address + "-I", null);
        deviceVector.insertElementAt(device, 0);
    }

    public void inquiryCompleted(int type) {
        isInInquiry = false;

        Alert dialog = null;
        // Determine if an error occurred. If one did occur
        // display an Alert
        // before allowing the application to exit.
        if (type != DiscoveryListener.INQUIRY_COMPLETED) {
            dialog = new Alert("Bluetooth Error",
                "The inquiry failed to complete normally", null,
                AlertType.ERROR);
        } else {
            dialog = new Alert("Inquiry Completed",
                "The inquiry completed normally", null,
                AlertType.INFO);
        }

        dialog.setTimeout(Alert.FOREVER);
        Display.getDisplay(this).setCurrent(dialog);
    }

    private void addDevices() {
        // Retrieve the pre-known device array and append the
        // addresses
        // of the Bluetooth device. If there are no pre-know
        // devices, move on to cached devices.
        RemoteDevice[] list =
            agent.retrieveDevices(DiscoveryAgent.PREKNOWN);
    }

```

```

    if (list != null) {
        for (int i = 0; i < list.length; i++) {
            String address = list[i].getBluetoothAddress();
            deviceList.insert(0, address + "-P", null);
            deviceVector.insertElementAt(list[i], 0);
        }
    }

    // Retrieve the cached device array and append the
    // addresses to the
    // list.
    list = agent.retrieveDevices(DiscoveryAgent.CACHED);
    if (list != null) {
        for (int i = 0; i < list.length; i++) {
            String address = list[i].getBluetoothAddress();
            deviceList.insert(0, address + "-C", null);
            deviceVector.insertElementAt(list[i], 0);
        }
    }
}

```

Now that the `DiscoveryMIDlet` keeps track of each `RemoteDevice` object found and when the `MIDlet` is performing an inquiry, the `DiscoveryMIDlet` can be modified to perform a service search. The service search is started when a user selects one of devices displayed on the screen. Because many Bluetooth devices cannot start service searches while the device is performing an inquiry, the inquiry is canceled if one is occurring before the search is started.

```

public class DiscoveryMIDlet extends BluetoothMIDlet
    implements DiscoveryListener {
    ...
    /**
     * The List of service records that were found.
     */
    private List serviceRecordList;
    ...

    public void commandAction(Command c, Displayable d) {

```

```
        if (c.getCommandType() == Command.EXIT) {
            if (isInInquiry) {
                // Try to cancel the inquiry.
                agent.cancelInquiry(this);
            }
            notifyDestroyed();
        } else if (c == List.SELECT_COMMAND) {
            // Since the deviceList is currently visible, the user
            // must
            // have selected a device to search so display the
            // serviceRecordList screen.
            serviceRecordList = new List("Services Found",
                List.IMPLICIT);
            serviceRecordList.addCommand(new Command("Exit",
                Command.EXIT, 1));
            serviceRecordList.setCommandListener(this);
            Alert splash = null;
            // If an inquiry is presently occurring, cancel the
            // inquiry
            // before starting the service search. Otherwise,
            // start the
            // service search
            if (isInInquiry) {
                agent.cancelInquiry(this);
                splash = new Alert("Cancel Inquiry",
                    "Ending the inquiry and starting the service search",
                    null, AlertType.INFO);
            } else {
                splash = new Alert("Starting Search",
                    "Starting the service search",
                    null, AlertType.INFO);
                startServiceSearch();
            }
            splash.setTimeout(2000);
            Display.getDisplay(this).setCurrent(splash,
                serviceRecordList);
        }
    }
}
/**
```

```

    * Starts the service search.
    */
private void startServiceSearch() {
    try {
        // Search for the Bluetooth Game service record and
        // retrieve
        // the name attribute in addition to the default
        // attributes.
        UUID[] uuidList = new UUID[1];
        uuidList[0] = new
            UUID("0FA1A7AC16A211D7854400B0D03D76EC", false);
        int[] attrList = new int[1];
        attrList[0] = 0x100;
        // The RemoteDevices are in the deviceVector in the
        // same order as
        // they are on the screen so getting the index
        // allows us to
        // retrieve the correct RemoteDevice object.
        int index = deviceList.getSelectedIndex();
        RemoteDevice d =
            (RemoteDevice)deviceVector.elementAt(index);
        int id = agent.searchServices(attrList, uuidList,
            d, this);
    } catch (BluetoothStateException e) {
        Alert error = new Alert("Error",
            "Unable to start the service search (" +
            e.getMessage() +
            ")", null, AlertType.ERROR);
        error.setTimeout(Alert.FOREVER);
        Display.getDisplay(this).setCurrent(error, deviceList);
    }
}

```

```

public void inquiryCompleted(int type) {
    isInInquiry = false;
    Alert dialog = null;
    // Determine if an error occurred. If one did occur
    // display an Alert
    // before allowing the application to exit.
    if (type != DiscoveryListener.INQUIRY_COMPLETED) {

```

```

        // If the device inquiry was terminated, then the
        // user must have
        // selected a Remote Device to perform a service
        // search on so
        // start the service search.
        if (type == DiscoveryListener.INQUIRY_TERMINATED) {
            startServiceSearch();

            return;
        } else {
            dialog = new Alert("Bluetooth Error",
                              "The inquiry failed to complete normally", null,
                              AlertType.ERROR);
        }
    } else {
        dialog = new Alert("Inquiry Completed",
                          "The inquiry completed normally", null,
                          AlertType.INFO);
    }
    dialog.setTimeout(Alert.FOREVER);
    Display.getDisplay(this).setCurrent(dialog);
}
}

```

The `startServiceSearch()` method is called from two different parts of the previous code. The `startServiceSearch()` method is called from the `commandAction()` method if an inquiry is not in progress. If an inquiry is in progress, the `commandAction()` method cancels the inquiry and the `startServiceSearch()` method is called from the `inquiryCompleted()` method when the cancel is processed.

The `startServiceSearch()` method starts the service search. This method performs a service search for the Bluetooth game service described earlier in this chapter. When a service is found with the Bluetooth game service's UUID, the 0x100 attribute also is retrieved. This is the `ServiceName` attribute ID as defined by the Bluetooth SIG. (This is used later in this chapter.)

Even though the `startServiceSearch()` method starts the search, the `DiscoveryMIDlet` does not do anything with the services that it finds at present. Therefore the `servicesDiscovered()` method is modified to display the number of service records returned to the `DiscoveryMIDlet`. The `serviceSearchCompleted()` method is also modified to display a message to the user when the service search ends.

```

public class DiscoveryMIDlet extends BluetoothMIDlet
    implements DiscoveryListener {

    ...

    public void servicesDiscovered(int transID,
        ServiceRecord[] record) {
        serviceRecordList.insert(0,
            Integer.toString(record.length), null);
    }

    /**
     * Called when the service search has ended. Displays a
     * message to the
     * user that the service search completed and specifies
     * if the search
     * completed normally.
     *
     * @param transID the transaction ID
     * @param type specifies how the service search completed
     */
    public void serviceSearchCompleted(int transID, int type) {
        Alert dialog = null;

        // Determine if an error occurred. If one did occur
        // display an Alert
        // before allowing the application to exit.
        if (type !=
            DiscoveryListener.SERVICE_SEARCH_COMPLETED) {
            dialog = new Alert("Bluetooth Error",
                "The service search failed to complete normally",
                null,
                AlertType.ERROR);
        } else {
            dialog = new Alert("Service Search Completed",
                "The service search completed normally", null,
                AlertType.INFO);
        }
        dialog.setTimeout(Alert.FOREVER);
        Display.getDisplay(this).setCurrent(dialog);
    }
}

```

7.3.6 Working with Service Records

After a `ServiceRecord` is retrieved from a service search, the next step is to determine whether the service described by the `ServiceRecord` is the desired service. Once the service is determined to be the desired service, the `getConnectionURL()` method can be called to retrieve the connection string that establishes a connection to the service. This connection string may then be passed to `Connector.open()` to establish the connection. The `getConnectionURL()` method also allows the application to specify the security requirements of the connection and whether the local device is the master or the slave (see section 4.3 for more information on Bluetooth security).

Before calling `getConnectionURL()`, the application must determine whether the `ServiceRecord` describes the service desired. This step highlights the need to be as specific as possible when determining the list of UUIDs to search for. Being as specific as possible minimizes the need to do additional work to determine whether the `ServiceRecord` returned is for the desired service. In most situations, a complete list of UUIDs used in the service search eliminates the need for additional verification. When the service record is discovered in this situation, all that is required is calling the `getConnectionURL()` method to begin using the service.

There are situations that require additional verification or determination. For example, if the local device is able to locate two instances of the same service, the local device could connect to the service that is currently less busy. The application may also want to request additional information that allows the user of the application to determine which service to use.

To actually access the values of each of the attributes, the `getAttributeValue()` method should be used. The `getAttributeValue()` method returns the attribute value of the attribute ID specified or null if the attribute ID is not in this service record. The value of an attribute is encapsulated in the `DataElement` class. The `DataElement` class provides accessor methods to determine the type of the data element and its value. Table 7.14 lists the different types of data elements and how these types relate to the `DataElement` class.

Before the value of a data element is retrieved, the `getDataType()` method should be called to verify the data type of the value. This step should always be done before retrieving the value of a

Table 7.14 Bluetooth Data Element Types and Their Associated Java Types

Bluetooth Type	DataElement	Java Type	Method for Retrieving Value from DataElement
	Data Type		
Null	NULL	Represents a null value	None
Unsigned integer (1 byte)	U_INT_1	long value in the range of 0 to 255	getLong()
Unsigned integer (2 bytes)	U_INT_2	long value in the range of 0 to $2^{16}-1$	getLong()
Unsigned integer (4 bytes)	U_INT_4	long value in the range of 0 to $2^{32}-1$	getLong()
Unsigned integer (8 bytes)	U_INT_8	byte[] value in the range of 0 to $2^{64}-1$	getValue()
Unsigned integer (16 bytes)	U_INT_16	byte[] value in the range of 0 to $2^{128}-1$	getValue()
Integer (1 byte)	INT_1	long value in the range -128 to 127	getLong()
Integer (2 bytes)	INT_2	long value in the range -2^{15} to $2^{15}-1$	getLong()
Integer (4 bytes)	INT_4	long value in the range -2^{31} to $2^{31}-1$	getLong()
Integer (8 bytes)	INT_8	byte[] value in the range -2^{63} to $2^{63}-1$	getValue()
Integer (16 bytes)	INT_16	byte[] value in the range -2^{127} to $2^{127}-1$	getValue()
URL	URL	java.lang.String	getValue()
UUID	UUID	javax.bluetooth.UUID	getValue()
Boolean	BOOL	boolean	getBoolean()
String	STRING	java.lang.String	getValue()
Data element sequence	DATSEQ	java.util.Enumeration	getValue()
Data element alternative	DATALT	java.util.Enumeration	getValue()

data element because calling the wrong method on a DataElement object causes a ClassCastException to be thrown.

To show how to use DataElements and ServiceRecords, we are modifying the DiscoveryMIDlet to display the ServiceName attribute value for each service record found. This is done by modifying the servicesDiscovered() method. First, the servicesDiscovered() method retrieves the DataElement for the ServiceName attribute. Before the string contained in the DataElement is extracted, the

getDataType() method must be called to determine the type of attribute value stored in the DataElement object. After it is verified that the data element is a string, the value of the data element is displayed.

```
public class DiscoveryMIDlet extends BluetoothMIDlet
    implements DiscoveryListener {
```

```
...
```

```
/**
 * The service records that were found in the last
 * service search.
 */
private Vector serviceRecordVector;
```

```
...
```

```
/**
 * Called each time a service is discovered. Retrieve
 * the name attribute
 * from the service record and display it on the
 * screen. Add the service
 * record to the service record Vector for later.
 *
 * @param transID the transaction ID
 * @param record the service records that were found
 */
```

```
public void servicesDiscovered(int transID,
    ServiceRecord[] record) {
```

```
// Process each service record individually
for (int i = 0; i < record.length; i++) {
    //Retrieve the name attribute from the service record
    DataElement nameElement =
        (DataElement)record[i].getAttributeValue(0x100);
    // The name attribute is only valid if it exists and
    // is a string
    // If either of these conditions fail, move on to the
    // next service record.
    if ((nameElement != null) &&
        (nameElement.getDataType() == DataElement.STRING)) {
        //Retrieve the name and display it on the screen.
```

```

        String name = (String)nameElement.getValue();
        serviceRecordList.insert(0, name, null);
        serviceRecordVector.insertElementAt(record[i], 0);
    }
}

...

private void startServiceSearch() {
    serviceRecordVector = new Vector();
    try {
        // Search for the Bluetooth Game service record and
        // retrieve
        // the name attribute in addition to the default
        // attributes.
        UUID[] uuidList = new UUID[1];
        uuidList[0] = new UUID(
            "0FA1A7AC16A211D7854400B0D03D76EC", false);
        int[] attrList = new int[1];
        attrList[0] = 0x100;

        // The RemoteDevices are in the deviceVector in the
        // same order as
        // they are on the screen so getting the index
        // allows us to
        // retrieve the correct RemoteDevice object.
        int index = deviceList.getSelectedIndex();
        RemoteDevice d =
            (RemoteDevice)deviceVector.elementAt(index);

        int id = agent.searchServices(attrList, uuidList, d,
            this);
    } catch (BluetoothStateException e) {
        Alert error = new Alert("Error",
            "Unable to start the service search (" +
            e.getMessage() +
            ")", null, AlertType.ERROR);
        error.setTimeout(Alert.FOREVER);
        Display.getDisplay(this).setCurrent(error, deviceList);
    }
}
}

```

In addition to displaying the service name of each service discovered, every `ServiceRecord` object returned to the application is stored in a `Vector` so that additional information can be gathered from the service record later.

7.3.7 Retrieving Additional Attributes after Service Discovery

The `getAttributeIDs()` method returns the IDs of all the attributes that have been retrieved from the remote device. This method does not return all the attributes defined in the service record on the remote device. Why would a `ServiceRecord` that has been discovered not have all the attributes of the service record on the remote device? The answer is simple. To reduce the amount of data actually sent over the air. If there is no intention to actually use an attribute, there is no reason to retrieve the attribute.

Sometimes the local device needs an attribute only in specific instances. JABWT provides a way to retrieve these attributes after the service search has been completed. For retrieving additional attributes, the `populateRecord()` method is called with the list of additional attributes to retrieve. The `populateRecord()` method returns `true` if some or all of the attributes specified are retrieved. The method may also throw an `IOException` if the remote device that has the service described by the `ServiceRecord` cannot be reached or the service is no longer available.

The `populateRecord()` method goes over the air to retrieve these additional attributes. Unlike `searchServices()`, which issues a request and then returns, a call to `populateRecord()` does not return until it fails or the information is retrieved. Because the `populateRecord()` method blocks, be aware of where this method is called. Calling the method within an event handler can affect the user's experience.

To show how to use the `populateRecord()` method, we modify the `DiscoveryMIDlet` to retrieve the `ServiceDescription` attribute. This procedure requires a few modifications to the `DiscoveryMIDlet`. Because it implements `Runnable` through the `BluetoothMIDlet` class, the `DiscoveryMIDlet` must have a `run()` method. When a user selects a service name from the List displayed on the screen, a new thread is created for the `DiscoveryMIDlet` that retrieves the `ServiceDescription` attribute by means of the `populateRecord()` method.

```

public class DiscoveryMIDlet extends BluetoothMIDlet
    implements DiscoveryListener {
    ...
    public void commandAction(Command c, Displayable d) {
        if (c.getCommandType() == Command.EXIT) {
            if (isInInquiry) {
                // Try to cancel the inquiry.
                agent.cancelInquiry(this);
            }
            notifyDestroyed();
        } else if (c == List.SELECT_COMMAND) {
            // Determine if the deviceList is the one that was
            // selected.
            if (d == deviceList) {
                // Since the deviceList is currently visible, the
                // user must
                // have selected a device to search so display the
                // serviceRecordList screen.
                serviceRecordList = new List("Services Found",
                    List.IMPLICIT);
                serviceRecordList.addCommand(new Command("Exit",
                    Command.EXIT, 1));
                serviceRecordList.setCommandListener(this);
                Alert splash = null;
                // If an inquiry is presently occurring, cancel
                // the inquiry
                // before starting the service search. Otherwise,
                // start the
                // service search
                if (isInInquiry) {
                    agent.cancelInquiry(this);
                    splash = new Alert("Cancel Inquiry",
                        "Ending the inquiry and starting the service"
                            + "search",
                        null, AlertType.INFO);
                } else {
                    splash = new Alert("Starting Search",
                        "Starting the service search",

```

```
        null, AlertType.INFO);
        startServiceSearch();
    }
    splash.setTimeout(2000);
    Display.getDisplay(this).setCurrent(splash,
        serviceRecordList);
} else {
    // Since the serviceRecordList is presently being
    // displayed,
    // get the description attribute for the service
    // that was
    // selected.
    new Thread(this).start();
}
} else {
    // The user must have selected the Back command. So
    // display the
    // names of all the services that were found.
    serviceRecordList = new List("Services Found",
        List.IMPLICIT);
    serviceRecordList.addCommand(new Command("Exit",
        Command.EXIT, 1));
    serviceRecordList.setCommandListener(this);
    for (int i = 0; i < serviceRecordVector.size(); i++) {
        // Services were only added to the
        // serviceRecordVector if
        // they had a valid name attribute. Therefore,
        // there is no need
        // to test the nameElement.
        ServiceRecord record =
            (ServiceRecord)serviceRecordVector.elementAt(i);
        DataElement nameElement =
            (DataElement)record.getAttributeValue(0x100);
        String name = (String)nameElement.getValue();
        serviceRecordList.insert(0, name, null);
    }
    Display.getDisplay(this).setCurrent(
        serviceRecordList);
}
```

```

    }
}
/**
 * This thread is started when the user selects a
 * service name. This
 * thread retrieves the description of the service and
 * displays the
 * description on the screen.
 */
public void run() {
    Alert error = null;
    try {
        // Identify the service record selected by the user
        int index = serviceRecordList.getSelectedIndex();
        ServiceRecord record =
            (ServiceRecord)serviceRecordVector.elementAt(index);
        // Retrieve the description attribute from the
        // remote device
        int[] attrList = new int[1];
        attrList[0] = 0x101;
        if (record.populateRecord(attrList)) {
            // Retrieve the description data element and
            // verify that it
            // exists and is a String
            DataElement descriptionDataElement =
                record.getAttributeValue(0x101);
            if ((descriptionDataElement != null) &&
                (descriptionDataElement.getDataType() ==
                 DataElement.STRING)) {
                // Display the description on the screen
                String description =
                    (String)descriptionDataElement.getValue();
                Form descriptionForm = new Form(
                    "Service Description");
                descriptionForm.append(description);
                descriptionForm.addCommand(new Command("Exit",
                    Command.EXIT, 1));
                descriptionForm.addCommand(new Command("Back",
                    Command.OK, 1));
            }
        }
    } catch (Exception e) {
        error = new Alert("Error", e.getMessage(), Alert.ERROR);
        error.show();
    }
}
}

```

```

        descriptionForm.setCommandListener(this);
        Display.getDisplay(this).setCurrent(
            descriptionForm);
        return;
    }
}
error = new Alert("Error",
    "Failed to retrieve the description of this service",
    null, AlertType.ERROR);
} catch (IOException e) {
    error = new Alert("Error",
        "Failed to retrieve the description (IOException: " +
        e.getMessage() + ")", null, AlertType.ERROR);
}
// Display the error message on the screen
error.setTimeout(2000);
Display.getDisplay(this).setCurrent(error);
}
}

```

Note that in the `run()` method, the `ServiceDescription` attribute is not simply retrieved and displayed. Like the `ServiceName` attribute displayed on the screen previously, the `DataElement` returned for the `ServiceDescription` attribute is inspected to determine whether the `DataElement` represents a `String`. Also, the return value of `populateRecord()` is checked to verify that the `ServiceDescription` attribute was retrieved.

In addition to starting the thread that retrieves the `ServiceDescription` attribute, the `commandAction()` method is modified to allow the user to return to the list of service names after retrieving the service description. This procedure allows the user to retrieve the `ServiceDescription` attribute of other services that were found during the service search.

7.3.8 Simple Device and Service Discovery

To make things easier for developers, the `selectService()` method combines the process of device and service discovery. The `selectService()` method returns a connection string that can be used by `Connector.open()` to connect to the service. If a service cannot be found

that meets the requirements of the search, `selectService()` returns null. The `selectService()` method has three arguments. The first argument is the UUID to search for in the `ServiceClassIDList` attribute. The second argument specifies the minimum security requirements needed for the connection. The third argument specifies whether the local device needs to be the master of this connection.

```
Connection getConnection(String uuidValue) throws
IOException {
    String connString;
    try {
        // Retrieve the LocalDevice and DiscoveryAgent objects.
        LocalDevice local = LocalDevice.getLocalDevice();
        DiscoveryAgent agent = local.getDiscoveryAgent();

        // Retrieve the connection string to the service
        UUID searchUUID = new UUID(uuidValue, false);
        connString = agent.selectService(searchUUID,
            ServiceRecord.NOAUTHENTICATE_NOENCRYPT, false);
    } catch (BluetoothStateException e) {
        throw new IOException("BluetoothStateException: " +
            e.getMessage());
    }
    if (connString == null) {
        throw new IOException(
            "Failed to locate a device with the UUID " +
            uuidValue);
    }
    return Connector.open(connString);
}
```

The `getConnection()` method uses the `selectService()` method to locate a service using the UUID value specified. If a service is found, the code calls `Connector.open()` to establish a connection to the service found. If a service is not found, the code throws an `IOException` to signal that a connection could not be established.

JABWT does not specify how `selectService()` finds a device with the service requested. Therefore an inquiry can occur during the call to `selectService()`. Because of this possibility, the `selectService()` method should be called in a separate thread

because an inquiry can last 10 seconds or more. This procedure prevents the application from appearing frozen while the `selectService()` method is called.

Generating a UUID for Your Service

For testing your server application, you can make up an arbitrary series of hexadecimal digits for the UUID needed in the server's connection string. However, when you are ready to package your application, you should provide a UUID that is truly unique so that clients can use this UUID to locate your server application on Bluetooth devices. For example, the `defineGameService()` method shown earlier in this chapter used the UUID `0FA1A7AC16A211D7854400B0D03D76EC`. A process has been specified for generating UUIDs that have a very high probability of being unique [35]. This process generates a UUID using a timestamp and the Ethernet address of the computer used to generate the UUID. It would have been useful to provide a method for generating UUID as part of the JABWT. However, this UUID-generating method would have been used only during development and would not have been used for actually running JABWT applications. This method was omitted from the API to keep the JABWT implementation as small as possible for J2ME devices.

Several utilities are available for generating UUIDs. These utilities tend to be operating system specific because of the need to access the Ethernet card address. On Windows, UUIDs can be generated with the tool `Guidgen.exe` that comes with Microsoft® Visual Studio®. On Linux a `uuidgen` function is available as part of the `e2fsprogs` package for second extended (ext2) Linux file systems. For example, the command

```
uuidgen -t
```

returns a result such as the following:

```
0fa1a7ac-16a2-11d7-8544-00b0d03d76ec
```

The last twelve hexadecimal digits are the network card address. As can be seen from the example output from `uuidgen`, it is conventional to include hyphens between certain digits of a UUID. However, hyphens are not allowed in the String representations of JABWT UUIDs; the hyphens must be removed before the UUID can be used in a JABWT connection string.

The Bluetooth SIG has reserved a range of UUIDs for use by the Bluetooth specifications. This reserved range starts at the Bluetooth base value of 00000000-0000-1000-8000-00805F9B34FB and includes all 2³² values up to FFFFFFFF-0000-1000-8000-00805F9B34FB. Applications should use values in this range only for the purposes described in the Bluetooth specifications.

7.3.9 Connect-Anytime Services

Some devices may provide a capability for starting selected server applications on demand when a client application attempts to connect to a server application not currently running. Server applications with this capability are called *connect-anytime services* in the JABWT specification. Section 7.2.1 discusses run-before-connect services. This section discusses connect-anytime services. An implementation of JABWT need not support both run-before-connect services and connect-anytime services. In fact, the first JABWT implementations support only run-before-connect services.

Connect-anytime services were described in the JABWT specification with the idea that they would take advantage of the MIDP 2.0 [26] capabilities for automatic startup of server MIDlets when clients attempt to connect. The relevant MIDP 2.0 class is `javax.microedition.io.PushRegistry`. JABWT does not explicitly mention any of the MIDP 2.0 mechanisms for automatically starting applications, because the JABWT specification was finalized before the proposed MIDP 2.0 APIs could be discussed publicly under the rules of the Java Community Process.

The run-before-connect service illustrated in Figure 7.2 is a proven scheme with demonstrated implementations. By contrast, the connect-anytime service sketched in Figure 7.5 is currently unproved, and the details remain to be worked out.

In Figure 7.5 the service record is added to the SDDb when the server application is first installed on the device. The service record remains visible to clients as long as the application has not been totally removed from the device. When a client discovers the service record and attempts to connect with a server installed but not currently running, the application management system starts the server application and hands off the client connection to the server. Clients need to do nothing special to connect to a connect-anytime service that is not currently running. A client simply retrieves the service record of the

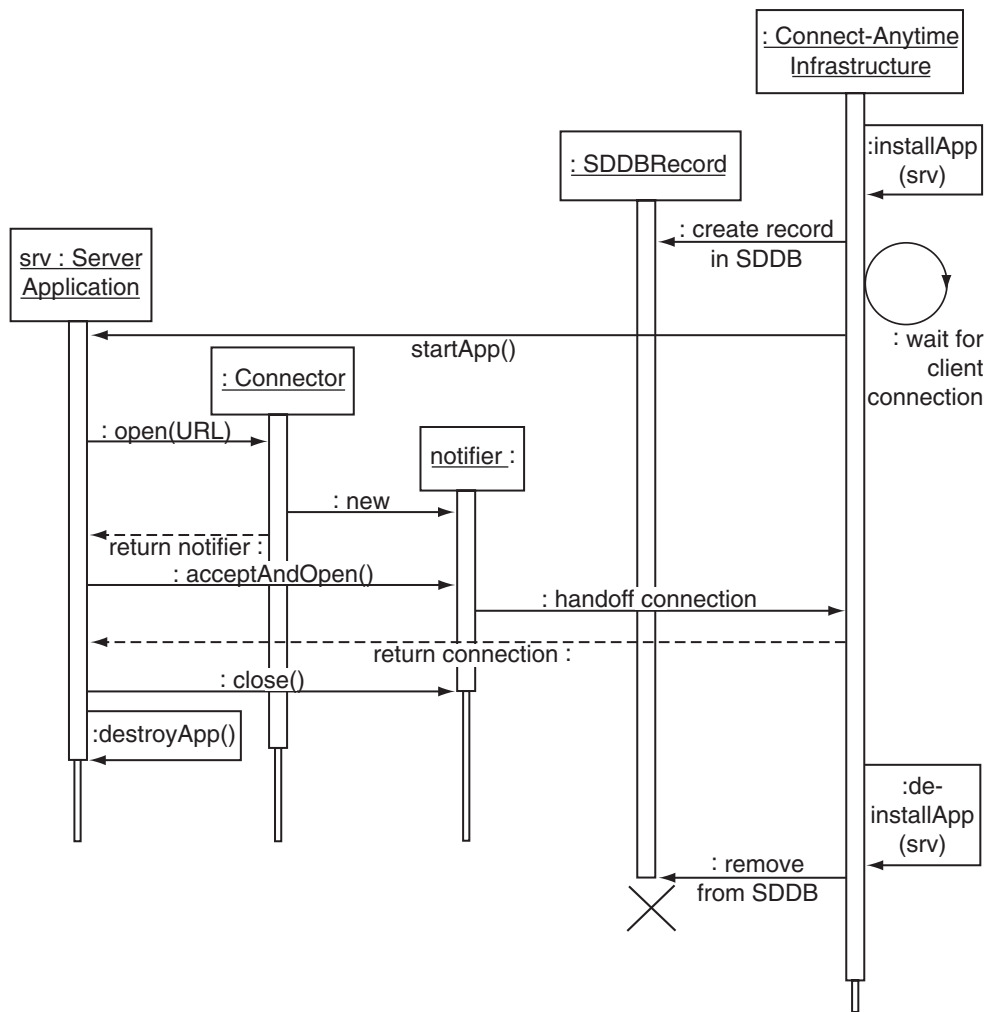


Figure 7.5 Lifecycle of a service record for a connect-anytime service.

connect-anytime service from the server and then attempts to connect using the `Connector.open()` string in the usual way.

There are several potential advantages of connect-anytime services. Device users frequently are responsible for starting run-before-connect services. This requires effort on the users' part, and they may sometimes forget to start a service. Connect-anytime services may be easier for users because users have only to start the client, and then the server will be

started automatically. The infrastructure for connect-anytime services is likely to ask user permission before starting up a service, but granting this permission should be easier for users than initiating the server startup process on their own. Also, if a device provides several services, it may be less computationally expensive to have the connect-anytime infrastructure listening for all service requests than it is to have all of the services running at all times.

Although Figure 7.5 shows the service record for a connect-anytime service being added to the SDDb at application installation, that is only one possibility. The JABWT specification provides a fair amount of latitude for when service records for connect-anytime services are actually added to the SDDb. The key point is that JABWT allows an option whereby clients can connect to a service not currently running, if it is possible to start the server application to complete the connection.

7.4 Summary

A Bluetooth client application communicates with a Bluetooth server application on another device to use the services provided by the server. Service discovery is the process of identifying the services offered by nearby Bluetooth devices by retrieving their service records. This chapter describes the JABWT methods used for locating service records that contain a particular collection of UUIDs and for retrieving some of the attributes of those service records.

Service registration is the process of creating a service record describing a service and adding it to the SDDb, where it can be discovered by clients. For many server applications, developers do not need to be concerned about service records. The JABWT implementation creates and registers a service record automatically. This service record advertises the service to potential clients and provides the information needed to construct a connection string that clients can use to access the service. In many cases, this automatically generated service record is sufficient.

In cases in which the automatically generated service record is inadequate, JABWT provides capabilities that allow developers to modify the service records. This chapter describes the service records automatically generated by JABWT and the procedure for modifying those service records when necessary.

8

CHAPTER

L2CAP

This chapter covers the following topics:

- What is L2CAP?
- What kinds of applications are appropriate for L2CAP communications?
- What support for L2CAP communications does JABWT provide?
- How is an L2CAP channel configured by a JABWT application?
- Why do L2CAP applications need to provide flow control?

8.1 Overview

L2CAP stands for *logical link control and adaptation protocol*. L2CAP is a multiplexing layer that allows several higher-level protocols or applications to use Bluetooth communications. Figure 8.1 shows where L2CAP fits in the Bluetooth protocol stack. The figure shows a common hardware configuration with separate host hardware and a Bluetooth radio module. In this configuration, the L2CAP layer is on the host side of the HCI. Also in this configuration, L2CAP uses HCI to communicate with the baseband layer in the Bluetooth radio module. All Bluetooth data communications use L2CAP, but Bluetooth voice communications do not.

As Figure 8.1 shows, several protocols sit on top of the L2CAP layer and use L2CAP to provide access to the Bluetooth hardware in the Bluetooth radio module. The SDP and the RFCOMM protocol are two of these higher-level protocols. JABWT provides access to both SDP and RFCOMM. Chapter 7 describes how JABWT applications use the SDP protocol to discover service records. Chapter 4 describes how JABWT applications use the RFCOMM protocol for serial port communications.

JABWT does not provide access to the other two protocols shown above L2CAP in Figure 8.1. TCS binary defines the call-control signaling

that establishes speech and data calls between Bluetooth devices. The BNEP [6] is an optional protocol developed after Bluetooth specification version 1.1 but based on the 1.1 version of the specification. The BNEP can be used to transmit IP packets over L2CAP and supports the PAN profile [36].

The PAN profile is one of three Bluetooth profiles that provide access to the Internet for Bluetooth devices. Two other profiles for Internet access are the LAN Access Profile [3] and the Dial-up Networking Profile [3]. The LAN Access Profile and the Dial-up Networking Profile use the RFCOMM protocol rather than the BNEP as their entry point into the Bluetooth protocol stack. The LAN Access Profile addresses the case in which a data terminal such as a laptop uses Bluetooth wireless technology to communicate with a LAN access point that serves as a gateway to a LAN. This is one of the use cases also addressed by the PAN profile. The PAN profile supersedes the LAN Access Profile, so the LAN Access Profile is now considered obsolete. The Dial-up Networking Profile describes the case in which a data terminal such as a laptop uses Bluetooth wireless technology to communicate with a cell phone or modem that provides a dial-up connection to a LAN.

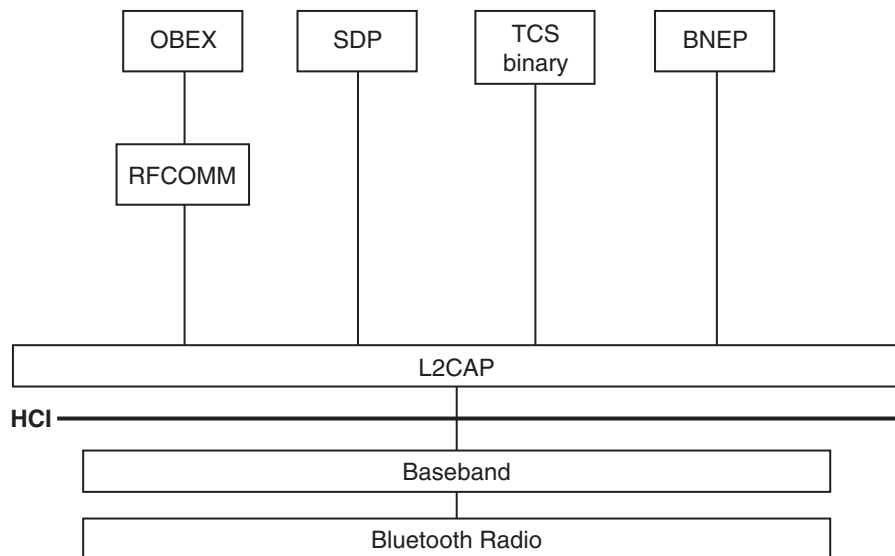


Figure 8.1 Position of L2CAP in the Bluetooth stack.

8.1.1 L2CAP Channels and L2CAP Packets

Figure 8.2 illustrates the multiplexing service that L2CAP provides. The left side of Figure 8.2 represents one Bluetooth device, and the right side represents another Bluetooth device. On the left side, an L2CAP server application and the RFCOMM protocol both are using L2CAP to provide Bluetooth communications. L2CAP establishes L2CAP channels that connect these higher-level entities to their counterparts on the remote device. In Figure 8.2, two L2CAP channels are represented. One L2CAP channel runs between the L2CAP server application and the L2CAP client application. Another L2CAP channel runs between the RFCOMM protocol layers on the two devices. In Figure 8.2 these L2CAP channels are represented as highways. The arrows traveling over the highways represent L2CAP packets. L2CAP provides full-duplex communications, so the L2CAP packets can travel in both directions.

The length of the arrow represents the size of the L2CAP packet. There is a limit to the size of the L2CAP packet that can be transferred over a particular L2CAP channel in a particular direction. This limit is called the *maximum transmission unit* (MTU). In Figure 8.2, the arrows heading to the L2CAP client are longer than the arrows heading from

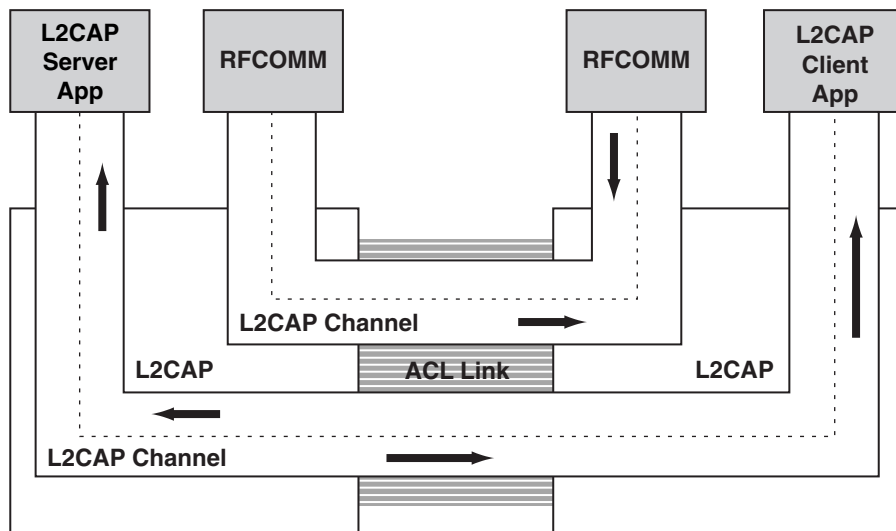


Figure 8.2 L2CAP channels transmit L2CAP packets to multiple destinations.

the L2CAP client. This configuration is intended to suggest that the MTU for the L2CAP packets received by the L2CAP client is larger than the MTU for packets received by the L2CAP server. When the L2CAP channel is established, the MTU values for travel in both directions are negotiated by the two L2CAP components. Section 8.2.3 provides additional details about MTUs.

Figure 8.2 shows the ACL link between the two devices. The baseband layer of the protocol stack establishes the ACL link. There is exactly one ACL link between two Bluetooth devices communicating with each other. The ACL link and the baseband functions provide the infrastructure needed to support the high-level, “logical” abstractions of L2CAP channels and L2CAP packets that L2CAP presents to higher-level protocols and applications.

L2CAP packets have to be converted into one or more baseband packets for transmission over the ACL link. The receiving device then reassembles these baseband packets into L2CAP packets. There are various sizes of baseband packets, but the largest payload is 339 bytes. This is much smaller than the largest payload possible for an L2CAP packet, 65,535 bytes. Because baseband packets are much smaller than packet sizes used by higher-level protocols and applications, the segmentation and reassembly process hides the details about Bluetooth baseband packets from the higher levels of the stack and from applications. By presenting abstractions such as L2CAP channels and L2CAP packets to higher levels, L2CAP makes it easier for higher-level protocols and applications to use Bluetooth communications. This adaptation function is one of the important contributions of L2CAP.

The L2CAP channels shown in Figure 8.2 are what are known as *connection-oriented channels*. They support L2CAP packet transmission in both directions, and the L2CAP packets transmitted are intended for use only by the single device at the other end of the ACL link. L2CAP also provides *connectionless channels*. Connectionless channels allow only one-way traffic, and they are intended for broadcasting L2CAP packets to a group of nearby devices. JABWT provides no support for connectionless channels.

8.1.2 Reasons for Using L2CAP

A number of Bluetooth protocols are defined on top of the L2CAP protocol. Figure 8.1 shows four of these protocols: RFCOMM, SDP, TCS

binary, and BNEP. However, the number of these protocols keeps growing as new Bluetooth profiles are developed that define new protocols based on L2CAP. The first column of Table 8.1 lists the current Bluetooth protocols that are layered on top of L2CAP. (Additional protocols and profiles are currently under development by the Bluetooth SIG.) The second column of Table 8.1 lists the current Bluetooth profiles that use the protocol listed in the first column. The Bluetooth protocols and profiles above the line dividing Table 8.1 are candidates for implementation with the JABWT L2CAP API. The profiles below the line that use the RFCOMM protocol are candidates to be implemented with the JABWT RFCOMM API. The JABWT service discovery API provides the key capabilities of the SDAP, so there should be no need to use JABWT to implement the SDAP.

The protocols and profiles above the line in Table 8.1 do not use RFCOMM or OBEX, so if they can be implemented at all with JABWT,

Table 8.1 Protocols and Profiles above Line Are Candidates for L2CAP API

Protocols That Use L2CAP Directly	Profiles Using the Protocol
Telephony Control Protocol Specification (TCS-BIN) [1]	Cordless Telephony Profile [3] Intercom Profile [3]
Bluetooth Network Encapsulation Protocol (BNEP) [6]	Personal Area Networks [36] Extended Service Discovery Profile [37]
Hardcopy Control Channel [7]	Hardcopy Cable Replacement Profile [7]
Hardcopy Notification Channel [7]	Hardcopy Cable Replacement Profile
Audio/Video Control Transport Protocol [8]	Audio/Video Remote Control Profile [38]
Audio/Video Distribution Transport Protocol [9]	Generic Audio/Video Distribution Profile [39] Advanced Audio Distribution Profile [40]
RFCOMM [1]	Serial Port Profile [3] Dial-up Networking Profile [3] FAX Profile [3] Headset Profile [3] Hands-Free Profile [41]
Service Discovery Protocol [1]	Service Discovery Application Profile [3]

they will use the L2CAP API. Because JABWT does not provide an interface to all of the L2CAP features, it may not be possible to implement all of the protocols and profiles in Table 8.1 with JABWT. For example, the first protocol in the table, TCS-BIN makes use of both connection-oriented and connectionless L2CAP channels. Because JABWT does not support connectionless L2CAP channels, it would not be possible to implement all of the TCS-BIN protocol with JABWT.

In the case of TCS-BIN, the JSR-82 expert group made an explicit decision that telephony control would likely be provided by device manufacturers, so it was not necessary for JABWT to provide an API that could be used to implement TCS-BIN. However, the other protocols and profiles above the line were defined by the Bluetooth SIG after definition of JABWT, which was based on the 1.1 version of the Bluetooth specification. Consequently, it may or may not be possible to implement these protocols and profiles with JABWT. Developers have to study the specifications carefully and assure themselves that JABWT support for L2CAP is sufficient to implement a particular protocol and profile.

For example, JABWT and the Bluetooth audio/video profiles have incompatible requirements regarding flush timeout. The baseband layer, a layer below L2CAP in the Bluetooth stack (Figure 8.1), offers the option of retransmitting packets until they are received successfully. The retransmit option is controlled by a parameter called *flush timeout*, which indicates how long the baseband attempts to retransmit a packet before giving up and flushing the packet. JABWT currently requires that L2CAP communications use a flush timeout of 0xFFFF, which means that baseband should never give up. The baseband should continue to retransmit until either the packet is acknowledged or the ACL link terminates. On the other hand, the Bluetooth audio/video profiles specify that applications using the audio/video profiles be allowed to set the value of flush timeout. These audio/video profiles recommend that small values should be used for flush timeout. Small values for flush timeout help ensure that most of the L2CAP bandwidth is devoted to the initial transmission of audio/video data and that retransmission is minimized. The next revision of the JABWT specification is expected to remove the requirement for using a flush timeout of 0xFFFF. The revised JABWT specification could introduce a way for applications to request a particular flush timeout. This would remove the current obstacle to using JABWT to implement the audio/video profiles.

In addition to the standardized applications defined as Bluetooth profiles, custom applications based on the L2CAP API are possible. In the case of custom applications, JABWT developers have a choice about whether to use the L2CAP API, the RFCOMM API, or the OBEX API. For most custom applications, the stream-oriented APIs provided by the higher-level protocols RFCOMM and OBEX would have advantages. However, if a custom application requires control over which bytes are sent together in a single packet, then the L2CAP API is the best choice. This might be the case if the application needs to define a new packet-based protocol analogous to the packet-based protocols listed in the first column of Table 8.1. If a stream-based protocol can be used, RFCOMM and OBEX are better options. OBEX itself is a good example of a stream-based protocol [1, 29] that can be defined over RFCOMM.

8.2 API Capabilities

This section describes the support that JABWT provides for L2CAP communications. The Java interfaces defined by JABWT for L2CAP are described. Examples of connection strings used to open an L2CAP connection are provided. Guidelines are proposed for configuring L2CAP channels with MTU parameters in connection strings and for selecting appropriate sizes for the byte arrays used to send and receive L2CAP packets. Issues related to flow control are discussed.

8.2.1 JABWT for L2CAP

Figure 8.3 shows the two interfaces defined in JABWT for L2CAP communications, `L2CAPConnection` and `L2CAPConnectionNotifier`. An L2CAP server uses an `L2CAPConnectionNotifier` to wait for an L2CAP client to establish a connection. The notifier then returns an `L2CAPConnection` object to provide access to the L2CAP channel between the client and the server. The `L2CAPConnection` interface can be used to send data between the client and the server using the L2CAP protocol.

When an L2CAP client is successful in opening a connection to an L2CAP server, the result returned from `Connector.open()` is an `L2CAPConnection` that gives the client access to the L2CAP channel. Again, the client uses the `L2CAPConnection` to send and receive data.

The L2CAP client does not use the `L2CAPConnectionNotifier` interface.

For serial port (RFCOMM) communications, the JABWT specification did not need to define any new classes or interfaces beyond those in the GCF defined by CLDC [13]. GCF already provided the stream-oriented interfaces `StreamConnection` and `StreamConnectionNotifier`. However, these CLDC interfaces were not useful for L2CAP because L2CAP communications are based on packets not on streams.

CLDC has a `DatagramConnection` interface that is packet oriented rather than stream oriented, and `DatagramConnection` was considered for possible use with L2CAP. As it turns out, `send()` and `receive()` methods were defined for `L2CAPConnection`. `DatagramConnection` has these same methods, but the method arguments are different. For `DatagramConnection` these methods take a `Datagram` argument, whereas the arguments to the `L2CAPConnection` versions of the methods are byte arrays. Each `Datagram` object sent over a `DatagramConnection` contains a destination address, because each `Datagram` can be going to a different recipient. However, this overhead associated with `Datagram` is not needed for L2CAP packets. Each `L2CAPConnection` instance defines a unique sending application and a unique receiving application. Because the address information is contained in the connection, there is no need to provide an address as part of the argument to `send()` or `receive()`.

The `receive()` method blocks until either an L2CAP packet is read or the L2CAP channel is closed. The `ready()` method defined by the `L2CAPConnection` interface makes it possible to check whether a call to `receive()` will block. The `ready()` method returns true if an L2CAP packet is available to be read immediately by `receive()` without blocking.

8.2.2 Open an L2CAP Connection

Here are some examples of legal arguments to `Connector.open()` for L2CAP clients:

```
"btl2cap://0050CD00321B:1001"
"btl2cap://0050CD00321B:1001;receiveMTU=512"
"btl2cap://0050CD00321B:1001;receiveMTU=512;"
+ "transmitMTU=512"
```

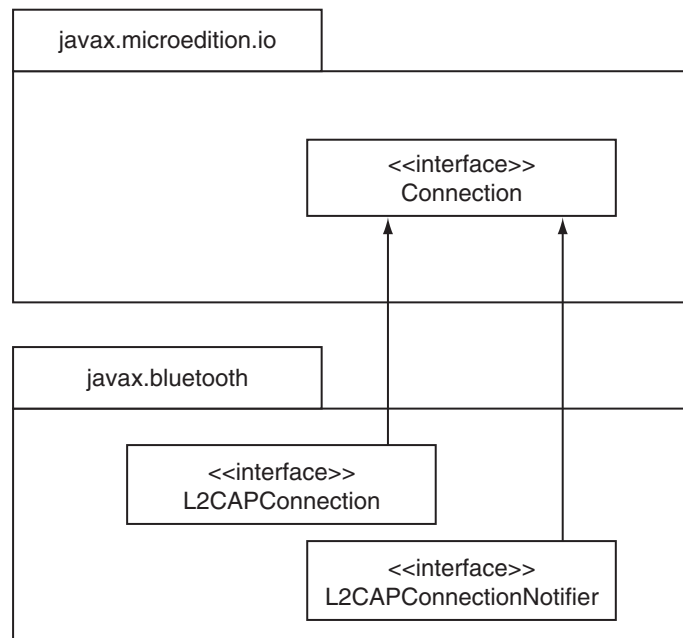


Figure 8.3 JABWT defines two interfaces for L2CAP communications.

```

"bt l2cap://0050CD00321B:1001;authenticate=true;"
+ "encrypt=true"
"bt l2cap://0050CD00321B:1001;master=true"

```

The protocol name `bt l2cap` is “bee tee *el* two cap,” not “bee tee *one* two cap.” The entry `0050CD00321B` in these examples is the Bluetooth address of the server device. The entry `1001` in these examples is the PSM value for the server application. The PSM is obtained from the service record. The PSM tells the L2CAP layer on the remote device which server application the client wants as the destination of the new L2CAP channel. Higher-level protocols such as RFCOMM and SDP have L2CAP PSM values permanently assigned to them. A PSM has to be dynamically generated for an application, and it is possible that a different PSM will be assigned to the same server application the next time it starts up. This situation is similar to that of `bt spp` server channel identifiers, which can vary from device to device and can even vary over time for the same service. One difference between PSMs and service

channel identifiers in JABWT is that PSMs in `bt12cap` connection strings are interpreted as hexadecimal numbers, whereas service channel identifiers in `btspp` and `btgoep` connection strings are interpreted as decimal numbers.

The security parameters `authenticate` and `encrypt` are familiar from Chapter 4. The parameter `master` used to request the master role in the resulting Bluetooth network also is familiar. The parameters `receiveMTU` and `transmitMTU` are unique to L2CAP. The parameter `receiveMTU` indicates the size in bytes of the payload of the largest L2CAP packet that the client is willing to receive from the server. The parameter `transmitMTU` indicates the size in bytes of the payload of the largest L2CAP packet that the client will send to the server. These parameters are discussed further in the next section.

L2CAP client applications typically get a connection string from one of the instance methods:

- `ServiceRecord.getConnectionURL(int requiredSecurity, boolean master)`
- `DiscoveryAgent.selectService(UUID uuid, int security, boolean master)`

However, if either of the parameters `transmitMTU` or `receiveMTU` is to be used, it must be appended to the Strings returned from these methods. There is no option to include these MTU parameters in the arguments to the methods as there is for the security parameters and the `master` parameter.

Here are some examples of legal arguments to `Connector.open()` for L2CAP servers:

```
"bt12cap://localhost:9C68A2AA1EC011D79E6C00B0D03D76EC"
"bt12cap://localhost:9C68A2AA1EC011D79E6C00B0D03D76EC;"
    + "name=L2CAPEx"
"bt12cap://localhost:9C68A2AA1EC011D79E6C00B0D03D76EC;"
    + "receiveMTU=512"
"bt12cap://localhost:9C68A2AA1EC011D79E6C00B0D03D76EC;"
    + "receiveMTU=512;"
    + "transmitMTU=1024"
"bt12cap://localhost:9C68A2AA1EC011D79E6C00B0D03D76EC;"
    + "authenticate=true;"
```

```
+ "encrypt=true;"  
+ "authorize=true"  
"bt l2cap://localhost:9C68A2AA1EC011D79E6C00B0D03D76EC;"  
+ "master=true"
```

The only new parameters for L2CAP servers are `receiveMTU` and `transmitMTU`. They have the same meaning for the server as they do for the client. A PSM value is not part of the server's connection string. The PSM is generated automatically and inserted into the service record by the JABWT implementation. This is similar to the situation for `bt spp` servers, in which the server channel identifier also is automatically generated and inserted into the service record (see Chapter 7).

8.2.3 L2CAP Channel Configuration

There may be a limit on the size of L2CAP payload the Bluetooth stack can receive or the size of payload a JABWT application is prepared to receive. The largest L2CAP packet payload the Bluetooth stack can receive is returned by `LocalDevice.getProperty("bluetooth.l2cap.receiveMTU.max")`.

The application should decide whether it wants to handle payloads as large as the stack can handle or something smaller. The answer is likely to depend on the nature of the application and the Java heap space expected to be available to applications.

On the basis of these considerations, the application can communicate `receiveMTU` to the remote device when it creates a connection using the `receiveMTU` parameter, `Connector.open("bt l2cap://...;receiveMTU=1024")`.

For some applications it is important to be able to send L2CAP packets up to a particular size. For example, the BNEP needs to be able to transmit the maximum Ethernet packet payload, 1500 bytes, plus all of the associated BNEP headers in a single L2CAP packet. Consequently, BNEP needs to be able to send L2CAP packets with payloads of at least 1691 bytes [6]. This requirement on the size of outgoing L2CAP packets can be declared by a JABWT application with the connection string parameter `transmitMTU=1691`.

In general, it is better not to specify `receiveMTU` or `transmitMTU` values in the connection string unless absolutely necessary. MTU values

are assigned automatically if no MTU values are mentioned in the connection string. The automatic assignment usually is the default MTU of 672 defined by the L2CAP specification. However, there are cases, such as the BNEP case discussed earlier, in which setting a particular value is required.

The documentation on MTUs can be confusing. Many of the details about `receiveMTU` and `transmitMTU` in the JABWT specification [20] are relevant only to implementers of the JABWT specification. The Bluetooth L2CAP specification [1] describes MTUs from the point of L2CAP. However, L2CAP allows a back-and-forth negotiation process for the MTU values for an L2CAP connection that does not really apply to JABWT applications. The `receiveMTU` and `transmitMTU` values specified in connection strings by JABWT applications should not be viewed as initial proposals in a back-and-forth negotiation. Instead they should be viewed as non-negotiable requirements.

For application developers, we boil down the essentials about MTUs to four rules for code development and one potential pitfall to be aware of even if all four rules are followed.

MTU Rule 1

The values for `receiveMTU` and `transmitMTU` in your L2CAP connection string must be no smaller than `L2CAPConnection.MINIMUM_MTU`, which is 48, the minimum MTU allowed by the Bluetooth L2CAP specification. The values for `receiveMTU` and `transmitMTU` also must be no larger than 65,535, the maximum payload size in an L2CAP packet.

MTU Rule 2

The value for `receiveMTU` in your L2CAP connection string must be smaller than or equal to `LocalDevice.getProperty("bluetooth.l2cap.receiveMTU.max")`, the largest L2CAP packet that can be received by the Bluetooth stack on the device on which your application is currently running. Applications can use this property at runtime to tailor their MTU values to the limits of any Bluetooth stack in use.

MTU Rule 3

For transmitting outgoing packets over an L2CAP connection with `send(byte[] outBuf)`, the byte array `outBuf` must be no larger than

`L2CAPConnection.getTransmitMTU()`. If `outBuf` is larger than this, bytes are discarded before the L2CAP packet is sent. If `transmitMTU` was declared in the L2CAP connection string, then `getTransmitMTU()` has that same value.

MTU Rule 4

To receive incoming packets with `receive(byte[] inBuf)`, allocate a byte array of size `L2CAPConnection.getReceiveMTU()`.

If you use an `inBuf` smaller than `L2CAPConnection.getReceiveMTU()`, any bytes received in the L2CAP packet that do not fit in `inBuf` are discarded. If `receiveMTU` was declared in the L2CAP connection string, you could allocate a byte array of size `receiveMTU`. However, if the remote device declares a `transmitMTU` in its connection string that is smaller than `receiveMTU`, then `L2CAPConnection.getReceiveMTU()` could be smaller than `receiveMTU`. It never is larger than `receiveMTU`.

If you follow all four MTU rules, it is still possible that at runtime a particular L2CAP client and server will be unable to form a connection because of incompatible MTU values. For example, suppose Application A specifies MTU values in its connection string as follows:

```
"btL2cap://...;receiveMTU=receiveMTUA;transmitMTU=
transmitMTUA"
```

Also, suppose Application B specifies MTU values in its connection string as follows:

```
"btL2cap://...;receiveMTU=receiveMTUB;transmitMTU=
transmitMTUB)"
```

MTU Mismatch Pitfall

The applications fail to connect because of inappropriate MTU values,

- If the largest packet Application A will send, `transmitMTUA`, is larger than the largest packet Application B can receive, `receiveMTUB`, or
- If the largest packet Application B will send, `transmitMTUB`, is larger than the largest packet Application A can receive, `receiveMTUA`

The basic problem that leads to the MTU mismatch pitfall is that the L2CAP protocol does not provide any way to inquire at runtime about the MTU requirements of the remote device other than trying to make a connection and seeing whether you succeed. (Server applications can use custom service attributes in their service records to communicate their MTU requirements, but we are not aware of any precedent for doing this in the Bluetooth profiles.)

If you are writing both the client and server applications, you can avoid this pitfall by not specifying MTU values at all in the connection string or by making `transmitMTUA = receiveMTUB` and `receiveMTUA = transmitMTUB`. If you have to interoperate with a variety of implementations, and they use different MTU values, the best strategy is to omit the `transmitMTU` parameter from your connection string. By omitting a `transmitMTU`, you avoid a mismatch with the `receiveMTU` of the remote device.

Omitting a `receiveMTU` in your connection string does not provide the same benefit. The L2CAP channel configuration process requires that each application propose an MTU value for incoming L2CAP packets. If there is no `receiveMTU` in the connection string, then the JABWT implementation supplies a value for `receiveMTU` by using the constant `L2CAPConnection.DEFAULT_MTU`, which has the value 672.

8.2.4 No Flow Control in L2CAP

L2CAP does not provide any flow control mechanism nor does it provide any mechanism for ensuring reliable transmission of L2CAP packets. The baseband layer, which is below L2CAP in the Bluetooth stack, provides flow control for the ACL link as a whole. Unfortunately, it is not sufficient to rely on the flow control provided by the lower baseband level. The problem is that L2CAP is a multiplexing layer that provides multiple L2CAP channels headed to multiple higher-level protocols or applications. In cases in which L2CAP packets arrive faster than they can be processed by one of the higher-level protocols and applications, the L2CAP buffers fill up. When buffers are going to overflow, the only options available to L2CAP are the following:

- Let the lower-level flow-control mechanisms kick in and shut off all incoming packets over this ACL link. Baseband flow-control

shuts off L2CAP packets over all the L2CAP channels, not just the L2CAP channels that are having trouble keeping up.

- Discard some L2CAP packets because there is no room for them in the L2CAP buffers. The lower stack layers will not retransmit these packets, because they have already been acknowledged as successfully received.

Simply discarding L2CAP packets is an unattractive option that would lead to data corruption or hung communications. However, the other option whereby the lower-level flow control shuts off all of the L2CAP channels can lead to deadlock in certain situations. The same problem of a multiplexing layer over a reliable communication layer arises in infrared data communications. The deadlock scenario for the IrDA protocol stack is discussed by Williams and Millar [42].

A summary of the deadlock scenario in L2CAP terms is as follows: Suppose a higher-level application uses two L2CAP channels. One of the L2CAP channels is used as a data channel, and the other L2CAP channel is used as a signaling channel. It is possible that the L2CAP buffers are overflowing because the application is waiting to receive an L2CAP packet on the signaling channel before it processes the packets on the data channel. If this is the case, and if baseband flow control shuts off the entire ACL link, then the L2CAP packet on the signaling channel cannot get through. The application continues to wait for this packet, so it does not process the packets on the data channel. However, processing packets on the data channel is the only thing that will free L2CAP buffers and get packets flowing again on the ACL link.

Because L2CAP has no flow control, the protocols and profiles that use L2CAP communications need to have their own mechanisms for flow control. For example, the RFCOMM protocol offers a credit-based flow control mechanism. The Hardcopy Cable Replacement Profile [7], which is one of three Bluetooth profiles for printing, also uses a credit-based flow control mechanism. The Bluetooth Extended Service Discovery Profile [37] uses an end-to-end window flow control mechanism.

If a flow control mechanism were to be added to L2CAP, it would not be necessary for every protocol and profile based on L2CAP to provide its own flow control. For this reason, there is a good chance that the Bluetooth SIG will provide a version of L2CAP with flow control in an upcoming version of the Bluetooth specification.

8.2.5 Types of Applications Using L2CAP

L2CAP applications can be implementations of standard Bluetooth profiles, or they can be nonstandardized, custom applications. The issues that come up are somewhat different depending on which kind of application is planned.

Implementing Bluetooth Profiles Using L2CAP

Certain Bluetooth profiles use L2CAP as their entry point into the Bluetooth protocol stack. Developers who intend to implement one of these Bluetooth profiles with JABWT applications need to study the profile specifications closely so that their applications can pass any Bluetooth qualification tests [10] for this profile and can successfully interoperate with other devices that support these profiles.

Some Bluetooth profiles place requirements on MTU values configured for L2CAP channels. For example, the Hardcopy Cable Replacement Profile establishes two L2CAP channels: a control channel and a data channel. The profile requires that the MTU for the control channel be at least 128 bytes in both directions. The profile recommends that the MTU for the data channel be larger than the minimum (48 bytes) but does not require a particular value.

Implementing Custom Applications Using L2CAP

L2CAP applications that do not claim to conform to a Bluetooth profile do not need to undergo the Bluetooth qualification process. However, developers planning a custom application will still benefit from studying the Bluetooth profiles that use L2CAP. These profiles provide useful examples of how to best use L2CAP.

You should consider how your application will provide flow control. Several flow control schemes have been adopted by the Bluetooth profiles and protocols. For example, SDP entails a simple scheme that requires that only one SDP request from an SDP client to an SDP server can be outstanding at any point in time. Until the server responds to this request, the client is not allowed to issue another request over this same L2CAP channel.

Other current approaches for flow control over L2CAP are referenced in Section 8.2.4. It is worth consulting those references to see what options are available.

8.3 Programming with the API

This section shows example code for MIDP applications that use JABWT L2CAP communications. The example code illustrates the use of JABWT and some design considerations for L2CAP applications. Not all of the code needed to produce a running application is presented here. The complete code is available in Appendix A.

8.3.1 Example: Echo L2CAP Packets

The example code shows both an L2CAP server and an L2CAP client. The L2CAP server echoes back any L2CAP packets sent by the L2CAP client. The client sends 50,000 bytes in a series of L2CAP packets. The size of the packets sent by the client is determined by the value of `getTransmitMTU()` for the connection. The payload of every packet the server receives is immediately sent back to the client. Both the client and the server keep a count of the total number of bytes sent or received over this connection. This byte count is reported when the packet exchange is complete.

We first look at the MIDlet method `openL2CAPConnection()`, which computes the connection string for either the server or the client and then starts it running. This method takes as arguments `receiveMTU` and `transmitMTU` and two boolean arguments that indicate whether those arguments should be added as parameters to the connection string. There is also an argument indicating whether a client or a server should be started. In the case in which a client should be started, `openL2CAPConnection()` uses the `selectService()` method to obtain a connection string for a server application. The `selectService()` method attempts to find a server application that uses a particular UUID in its service record. Having computed the connection string in the `url` variable, the `openL2CAPConnection()` method starts a thread to execute the client. In the case in which the method is starting a server, computing the connection string is just a matter of concatenating strings for the `bt l2cap` scheme, a UUID for the `ServiceClassID` (see Chapter 7), and any MTU parameters.

```
void openL2CAPConnection(boolean isClient,
                        boolean receiveMTUInput,
                        int receiveMTU,
                        boolean transmitMTUInput,
                        int transmitMTU) {
```

```

String url = null;
String paramString = "";

if (receiveMTUInput) {
    paramString += ";receiveMTU=" + receiveMTU;
}
if (transmitMTUInput) {
    paramString += ";transmitMTU=" + transmitMTU;
}
if (isClient) {
    displayField.setText(
        "searching, please wait...");
    DiscoveryAgent agent =
        device.getDiscoveryAgent();
    try {
        url
            = agent.selectService(uuid,
                ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
                false);
    } catch (BluetoothStateException e) {
        displayError("Error",
            "BluetoothStateException: " +
            e.getMessage());
    }
    if (url == null) {
        displayError("Error",
            "failed to find server!");
        return;
    }
    url += paramString;
    new L2capClient(this).start(url);
} else {
    url = "btl2cap://localhost:" + uuid.toString();
    url += paramString;
    new L2capServer(this).start(url);
}
}

```

The L2capServer runs in its own thread, and it establishes an L2CAPConnection in its run() method.

```
public class L2capServer extends EchoParticipant
    implements Runnable {
    L2capMtuMIDlet parent;
    private String url;

    public L2capServer(L2capMtuMIDlet parent) {
        this.parent = parent;
        this.out = parent.displayField;
    }

    public void start(String url) {
        this.url = url;
        new Thread(this).start();
    }

    public void run() {
        LocalDevice device = null;
        L2CAPConnectionNotifier notifier = null;
        try {
            device = LocalDevice.getLocalDevice();
            /* Request that the device be made discoverable */
            device.setDiscoverable(DiscoveryAgent.GIAC);
        } catch (BluetoothStateException e) {
            parent.displayError("Error",
                               "BluetoothStateException: " +
                               e.toString());

            return;
        }
        try {
            notifier = (L2CAPConnectionNotifier)
                Connector.open(url);
        } catch (IllegalArgumentException e) {
            parent.displayError("Error",
                               "IllegalArgumentException in " +
                               "Connector.open()");
        } catch (IOException e) {
            parent.displayError("Error",
                               "IOException: " +
                               e.getMessage());
        }
    }
}
```


The statement that adds the service record to the SDDb and waits for a client to connect is

```
L2CAPConnection conn = notifier.acceptAndOpen();
```

If this statement does not throw an exception, it returns an instance of an `L2CAPConnection`, which provides access to the L2CAP channel between the `L2capServer` and the `L2capClient`.

The definition of the `echoReceivedL2capPackets()` method that actually sends and receives the bytes is described below in the `EchoParticipant` class. The `L2capServer` extends the `EchoParticipant` class, so it inherits this method.

The code for the `L2capClient` is shown next. Here the key statement in the `run()` method is

```
conn = (L2CAPConnection)Connector.open(url);
```

Again, the `openL2CAPConnection()` method provides the `url`; its value varies with the Bluetooth device address of the server and the PSM assigned to the server application. In this first example it has the value

```
"bt12cap://0050CD00321B:1001;authenticate=false;"
+ "encrypt=false;master=false;"
+ "receiveMTU=672;transmitMTU=672"
```

The `Connector.open(url)` statement attempts to form an L2CAP connection to the echo service described by the `url` argument.

```
public class L2capClient extends EchoParticipant
    implements Runnable {
    L2capMtuMIDlet parent;
    private String url;

    public L2capClient(L2capMtuMIDlet parent) {
        this.parent = parent;
        this.out = parent.displayField;
    }

    public void start(String url) {
        this.url = url;
```

```

        new Thread(this).start();
    }

    public void run() {
        L2CAPConnection conn = null;
        out.setLabel("["+url+"]");
        try {
            conn = (L2CAPConnection)Connector.open(url);
        } catch (IllegalArgumentException e) {
            parent.displayError("Error",
                                "IllegalArgumentException in "
                                + "Connector.open()\n"
                                + e.getMessage());
        } catch (BluetoothConnectionException e) {
            String problem = "";
            if (e.getStatus() ==
                BluetoothConnectionException.UNACCEPTABLE_PARAMS)
                {problem = "unacceptable parameters\n";
            }
            parent.displayError("Error",
                                "BluetoothConnectionException: "
                                + problem + "msg=" +
                                e.getMessage() +
                                "\nstatus= " + e.getStatus());
        } catch (IOException e) {
            parent.displayError("Error",
                                "IOException: " + e.getMessage());
        }
        if (conn == null) {
            return;
        }
        try {
            sendL2capPacketsForEcho(conn);
            conn.close();
        } catch (IOException e) {
            parent.displayError("Error",
                                "IOException: " + e.getMessage());
        }
    }
}

```

If a connection cannot be formed between the L2CAP client and the L2CAP server because of incompatible MTU values (see the MTU mismatch pitfall in Section 8.2.3) then a `BluetoothConnectionException` is thrown with a status of `BluetoothConnectionException.UNACCEPTABLE_PARAMS`. The error handling code for the `L2capClient` class checks for a `BluetoothConnectionException` with that status. The `BluetoothConnectionException` class defines five other constants in addition to `UNACCEPTABLE_PARAMS`. These constants describe different reasons that a connection attempt might fail. As shown in the example code, the `getStatus()` method is used to retrieve the constant that applies to a particular exception.

Once the `L2CAPConnection` is established, it is passed to the method `sendL2capPacketsForEcho()`, which sends and receives the bytes over the L2CAP channel. The `sendL2capPacketsForEcho()` method is inherited from `EchoParticipant`, which is the next class we examine. This class has two methods. The method `sendL2capPacketsForEcho()` is used by the client for generating the L2CAP packets. The method `echoReceivedL2capPackets()` is used by the server for echoing back the bytes received from the client.

```
public class EchoParticipant {

    protected StringItem out;
    private int bytesToSend = 100000;

    void sendL2capPacketsForEcho(L2CAPConnection conn)
        throws IOException {

        byte[] sbuf = new byte[conn.getTransmitMTU()];
        byte[] rbuf = new byte[conn.getReceiveMTU()];
        for (int i=0; i < sbuf.length; i++) {
            sbuf[i] = (byte)i;
        }
        int count = 0;
        long start = System.currentTimeMillis();
        while (count < bytesToSend) {
            conn.send(sbuf);
            count += sbuf.length;
            count += conn.receive(rbuf);
        }
    }
}
```

```

        /* Display the bytes sent and received so far */
        out.setText(Integer.toString(count));
    }
    /* Let the echoer know we are done sending bytes */
    conn.send("DONE".getBytes());
    conn.receive(rbuf);
    long end = System.currentTimeMillis();
    out.setText("Done (transferred "+count+" bytes)\n"
        + "Elapsed time " + (end - start)/1000
        + "sec");
}

void echoReceivedL2capPackets(L2CAPConnection conn)
    throws IOException {

    byte[] ibuf = new byte[conn.getReceiveMTU()];
    int bytesIn;
    int count = 0;
    for (;;) {
        bytesIn = conn.receive(ibuf);
        byte[] obuf = new byte[bytesIn];
        System.arraycopy(ibuf, 0, obuf, 0, bytesIn);
        conn.send(obuf);
        if ((bytesIn == 4) && (new
            String(obuf)).equals("DONE")) {break;}
        count += 2 * bytesIn;
        /* Display the bytes received and sent so far */
        out.setText(Integer.toString(count));
    }
    out.setText("Done (transferred " + count + " bytes)");
}
}

```

The key parts of both methods are the statements that send and receive L2CAP packets. The method `sendL2capPacketsForEcho()` does

```
conn.send(sbuf);
```

followed by

```
count += conn.receive(rbuf);
```

The method `echoReceivedL2capPackets()` reverses the order of these operations.

The method `sendL2capPacketsForEcho()` follows MTU Rule 3, which limits the size of packets sent to a maximum of `getTransmitMTU()`. This is shown in the two statements

```
byte[] sbuf = new byte[conn.getTransmitMTU()];
```

and

```
conn.send(sbuf);
```

The method `sendL2capPacketsForEcho()` also follows MTU Rule 4, which recommends allocating a byte array of size `getReceiveMTU()` to receive incoming packets:

```
byte[] rbuf = new byte[conn.getReceiveMTU()];
```

and

```
count += conn.receive(rbuf);
```

The method `echoReceivedL2capPackets()` also follows MTU Rules 3 and 4, although this is more difficult to see for MTU Rule 3. The relevant statements are

```
bytesIn = conn.receive(ibuf);  
byte[] obuf = new byte[bytesIn];  
...  
conn.send(obuf);
```

The size of the byte array sent is based on the size of the byte array received. How do we know that the byte array `obuf` in `conn.send(obuf)` is not larger than `transmitMTU`? Because all MTU values are specified as 672 bytes in both the server and client connection strings, we know that the packet received will not be larger than 672 bytes. We can conclude that `obuf`, the byte array sent, also will be no larger than 672 bytes.

The client knows it is finished sending packets when its count exceeds 100,000 bytes. The `L2capClient` sends a special 4-byte packet

corresponding to the ASCII values for the character string "DONE" to inform the L2CAPServer that the client is finished sending bytes.

8.3.2 User Interface for MTU Values

The example code in this section extends the L2CAP echo program of the previous section with a user interface that lets you enter values for `receiveMTU` and `transmitMTU` for both the L2CAP server and the L2CAP client. These MTU values are then used in the connection strings. If an empty value is provided for one of these MTU values, that is, if the field in the user interface is cleared, the corresponding MTU parameter is not included in the connection string passed to `Connector.open()`. This user interface makes it easy to try various combinations of MTUs to see the effect of MTU Rules 1 and 2 and the MTU mismatch pitfall discussed in Section 8.2.3. This user interface also makes it possible to experience the effect of MTU size on the time required for the client and server to transmit 100,000 bytes over L2CAP.

Figure 8.4 shows the user interface for entering MTU values. Both the server and the client use the same user interface. Figure 8.4 (a) shows the first screen, which allows the user to enter a value for `receiveMTU`. Figure 8.4 (b) shows the second screen, which allows the user to enter a value for `transmitMTU`. The values shown in Figure 8.4 are compatible because $\text{transmitMTU} \leq \text{receiveMTU}$ for L2CAP packets sent from the client to the server.

The method `getReceiveMTUFromUser()` shown below creates the display shown in Figure 8.4 (a) for entering a value for `receiveMTU`. The constant `L2CAPConnection.MINIMUM_MTU` is used to display the lower bound on legal input values. The `LocalDevice` property `bluetooth.l2cap.receiveMTU.max` is used to display the upper bound on legal input values. The constant `L2CAPConnection.DEFAULT_MTU` is provided as the starting value of the input field.

```
private void getReceiveMTUFromUser(boolean isClient) {
    String maxRecMTUPlus1;
    String maxRecMTU
        = LocalDevice.getProperty(
            "bluetooth.l2cap.receiveMTU.max");
    if (maxRecMTU == null) {
        maxRecMTUPlus1 = "Unknown";
    } else {
```



Figure 8.4 User interface for MTU experiments. (A) A receiveMTU of 672 bytes is entered for the server. (B) A transmitMTU of 500 bytes is entered for the client (emulation only).

```

/* Get (max + 1) for display of (min - 1) < x <
(max + 1) */
maxRecMTUplus1
    = (new Integer(Integer.parseInt(maxRecMTU) +
    1)).toString();
}
String initialMTU
    = Integer.toString(L2CAPConnection.DEFAULT_MTU);
receiveMTUForm = new Form(isClient ? "L2CAP Client" :
    "L2CAP Server");
String recMtuFieldLabel

```

```

        = "ReceiveMTU \n" + (L2CAPConnection.MINIMUM_MTU
            - 1) + " < x < " + maxRecMTUPlus1;
receiveMTUForm.append(new TextField(recMtuFieldLabel,
    initialMTU, 10, TextField.NUMERIC));
receiveMTUForm.addCommand(new Command("Exit",
    Command.EXIT, 1));
receiveMTUForm.addCommand(new Command("Next",
    Command.ITEM, 1));
receiveMTUForm.setCommandListener(this);
display.setCurrent(receiveMTUForm);
}

```

We can cause problems for the L2capServer if we enter the following MTU values in the user interface:

Client: receiveMTU=500, transmitMTU=672

Server: receiveMTU=672, transmitMTU=500

These values avoid the MTU mismatch pitfall, so the connection is formed. The client sends 672-byte L2CAP packets to the server. The server can receive these packets because 672 bytes is the same as the server's receiveMTU. However, when the server attempts to echo the bytes back to the client, a 672-byte packet is larger than the server's transmitMTU of 500 bytes.

There are several options for dealing with this problem. For the sake of simplicity, we adopt an approach that does only one send(outBuf) for each receive(inBuf) and uses an outBuf of size transmitMTU. The extra 172 bytes are not echoed back to the client. This allows us to have just one send() for every receive() and keeps the example code a little simpler. In theory, using an outBuf of 672 bytes with all the received bytes should lead to the same result as using an outBuf of size transmitMTU. The excess 172 bytes should be automatically discarded by the JABWT implementation. However, this would violate MTU Rule 3. Following MTU Rule 3 here makes it clear in the code that the failure to echo all the bytes is intentional.

The shaded statements below show the changes made to the method echoReceivedL2capPackets() shown in Section 8.3.1. The shaded statements limit the number of bytes echoed from each incoming L2CAP packet to just transmitMTU bytes.

```

void echoReceivedL2capPackets(L2CAPConnection conn)
    throws IOException {

    byte[] ibuf = new byte[conn.getReceiveMTU()];

    int transmitMTU = conn.getTransmitMTU();

    int bytesIn;

    int bytesOut;

    int count = 0;
    for (;;) {
        bytesIn = conn.receive(ibuf);

        bytesOut = Math.min(bytesIn, transmitMTU);
        byte[] obuf = new byte[bytesOut];
        System.arraycopy(ibuf, 0, obuf, 0, bytesOut);

        conn.send(obuf);
        if ((bytesIn == 4) && (new String(obuf)).equals
            ("DONE")) {
            break;}

        count += bytesIn + bytesOut;

        /* Display the bytes received and sent so far */
        out.setText(Integer.toString(count));
    }
    out.setText("Done (transferred " + count + " bytes)");
}

```

8.3.3 L2CAP Clients and Servers Have the Same Capabilities

The example code in this chapter might leave the erroneous impression that L2CAP server applications have to be passive and are incapable of initiating communications. This is not the case. Although the client initiates the L2CAP connection, once that connection is formed, both sides have access to an instance of an `L2CAPConnection`, so both sides have

the same capabilities. Clients and servers can both send packets whenever they want to. An easy experiment that illustrates this point is to exchange these two statements in the example code:

```
sendL2capPacketsForEcho(conn);
```

and

```
echoReceivedL2capPackets(conn);
```

This exchange changes the example code so that instead of the server echoing the packets sent by the client, the client echoes the packets sent by the server.

8.3.4 Flow Control

L2CAP provides no flow control, so Bluetooth protocols and profiles that use L2CAP typically provide their own flow control. Let's consider the echo example of this chapter from the point of view of flow control. Suppose the client device is capable of sending L2CAP packets much faster than the server device is capable of echoing them back. Because our example code waits for a packet to be returned before it tries to send the next packet, the client is paced by the server's ability to echo the packets. This should keep the client from getting ahead of the server and overflowing the server's buffers.

However, suppose that instead of a symmetric, two-way echoing application, the data transfer is one way. In that case, a different flow control scheme is required. In addition to the L2CAP packets that transmit data and travel in one direction it would be necessary to send back L2CAP packets containing control signals to stop and start the data flow.

The next code example illustrates a one-way data transfer using a credit-based flow control scheme.

The code for the complete example is too lengthy to be shown here, so only selected methods involved with credit-based flow control are shown. The complete code is available in Appendix A. Credit-based flow control is used in the Bluetooth specification for RFCOMM and the Hardcopy Cable Replacement Profile. In the credit-based flow control scheme illustrated in this example, the L2CAP server starts by issuing four credits to the L2CAP client. The client can send as many L2CAP

packets as it has credits, so the client can then send four packets to the L2CAP server. When the client's credit count reaches zero, the client must stop sending L2CAP packets and wait for additional credits from the server.

In the `CreditBased1WayXfer` class shown below, the key section that accomplishes the flow control is

```
if (availableCredits > 0) {  
    conn.send(sbuf);  
    availableCredits--;
```

The L2CAP client can use the JABWT `send( )` method to send an L2CAP packet to the server as long as it has available credits. However, each packet sent uses up a credit. When the credits reach zero, the L2CAP client has to stop sending data until more credits are received.

```
public class CreditBased1WayXfer {  
  
    // number of L2CAP packets the receiver has  
    // authorized to be sent  
    int availableCredits;  
    protected StringItem out;  
    private int bytesToSend = 50000;  
  
    void sendL2capPackets(L2CAPConnection conn) throws  
        IOException {  
  
        boolean sentDone = false;  
        byte[] sbuf = new byte[conn.getTransmitMTU()];  
        int receiveMTU = conn.getReceiveMTU();  
        byte[] rbuf = new byte[receiveMTU];  
        for (int i=0; i < sbuf.length; i++) {  
            sbuf[i] = (byte)i;  
        }  
        int count = 0;  
        long start = System.currentTimeMillis();  
        //listen for credits authorizing sending packets
```

```

        receiveCredits(conn, receiveMTU);
        while (count < bytesToSend) {
            if (availableCredits > 0) {
                conn.send(sbuf);
                availableCredits--;
                count += sbuf.length;
                // Display the number of bytes sent so far
                out.setText(Integer.toString(count));
            }
            maybeReceiveCredits(conn, receiveMTU);
        }
        // Let the receiver know we are done sending bytes
        while (!sentDone) {
            if (availableCredits > 0) {
                conn.send("DONE".getBytes());
                sentDone = true;
            } else {
                maybeReceiveCredits(conn, receiveMTU);
            }
        }
        long end = System.currentTimeMillis();
        out.setText("Done (transferred "+count+" bytes)\n"
            + "Elapsed time " + (end -
                start)/1000 + "sec");
    }
}

```

The `sendL2capPackets()` method above uses the two application methods `receiveCredits()` and `maybeReceiveCredits()` to listen for L2CAP packets that deliver additional credits from the server. The definition of the `receiveCredits()` method is shown next. This method reads an L2CAP packet from the server and interprets the byte array in that packet as an integer. That integer is added to `availableCredits` to increase the credits available for use by the client.

```

// Read an L2CAP packet. If it has four bytes, then
// interpret those

```

```

// bytes as new credits for sending L2CAP packets.
void receiveCredits(L2CAPConnection conn,
    int receiveMTU) {
    int incomingBytes;
    int newCredits;
    byte[] rbuf = new byte[receiveMTU];
    try {
        incomingBytes = conn.receive(rbuf);
    } catch (IOException ignore) {
        return;
    }
    // assume four bytes are used to encode new
    // credits
    if (incomingBytes != 4) {
        return;
    }
    availableCredits +=
        CreditBased1WayXfer.byteArray2Int(rbuf);
}

// Convert a four-byte array to an int. The byte
// array is assumed
// to have a big Endian byte order.
public static int byteArray2Int(byte[] argBytes) {
    int result = 0;
    // big-endian conversion
    for (int i = 0, j = 0; i < 4; i++, j++) {
        result = result + (((int)argBytes[i] << 24)
            >>> (j * 8));
    }
    return result;
}

```

The L2CAP server issues an additional credit to the client only when the server frees up buffer space to hold one additional L2CAP data packet. It is unclear how long it takes the server to process one of the previously sent L2CAP packets, free up the space needed to receive another packet from the client, and send an L2CAP packet back to the client to issue the

additional credit. Because `receiveCredits()` uses the blocking JABWT method `receive()` to read an L2CAP packet from the server, there is always the risk that a call to `receive()` can become stuck waiting for this L2CAP packet from the server. The credit-based flow control scheme is meant to block only when the client has no more credits, so we want the client to listen for additional credits without blocking.

The key to keeping the client from getting stuck in `receive()` is to use the JABWT `ready()` method to test whether an L2CAP packet is available for the client to read. If `ready()` returns `true`, `receive()` returns an L2CAP packet without blocking. The client's `maybeReceiveCredits()` method is shown next. It uses the `ready()` method to check for additional credits issued by the server without blocking.

```
// If there is an L2CAP packet waiting to be read,
// then call
// receiveCredits. Otherwise return without
// blocking.
void maybeReceiveCredits(L2CAPConnection conn,
    int receiveMTU) {
    try {
        if (conn.ready()) {
            receiveCredits(conn, receiveMTU);
        }
    } catch (IOException ignore) {
    }
}
```

All of the example code we have looked at so far for credit-based flow control has been client code. The L2CAP server in this example has two threads: one thread to move the L2CAP packets received from the client to a buffer storage location and a second thread to process the buffered L2CAP packets and issue credits.

The `receiveL2capPackets()` method shown next is used in the thread that receives the incoming L2CAP packets and stores them. It is derived from the `echoReceivedL2capPackets()` used in the echo examples earlier in this chapter. The `receiveL2capPackets()` method has been modified to store the bytes obtained from an L2CAP packet sent by the client to one of four `L2capPacketBuffers` maintained

by the server. An `L2capPacketBuffer` has room to store `getReceiveMTU()` bytes. These bytes are processed later by the second thread. The code in `echoReceivedL2capPackets()` for counting incoming bytes and for echoing those bytes back to the client has been removed from `receiveL2capPackets()`.

```
public class CreditBased1WayXfer {
...

    void receiveL2capPackets(L2CAPConnection conn,
                            L2capBuffers buffers,
                            CreditIssuer issuer)
        throws IOException {

        byte[] ibuf = new byte[conn.getReceiveMTU()];
        L2capPacketBuffer packetBuffer;
        int transmitMTU = conn.getTransmitMTU();
        int bytesIn;

        for (;;) {
            packetBuffer =
                buffers.nextAvailablePacketBuffer();
            if (packetBuffer != null) {
                bytesIn = conn.receive(ibuf);
                packetBuffer.storeBytes(ibuf, bytesIn);
                if ((bytesIn == 4) &&
                    (new String(ibuf, 0,
                                bytesIn)).equals("DONE")) {
                    break;
                }
            } else {
                if (conn.ready()) {
                    System.out.println("Should not get here. No " +
                                        "L2capPacketBuffer " +
                                        "available " +
                                        "to receive incoming " +
                                        "packet.");
                }
            }
        }
    }
}
```

```

    }
    issuer.setDoneProcessing();
    out.setText("Done (transferred " + issuer.count +
        " bytes)");
}
...
}

```

The following example code shows the `run()` method for the `CreditIssuer` thread that processes the buffered packets and issues credits to the client. The first thing that happens in the `CreditIssuer.run()` method is that an L2CAP packet is sent to the client containing a number represented as a byte array. The number in the L2CAP packet is the number of starting credits that the server issues to the client. This number is determined by the number of client packets that the server has buffer space available to receive.

The `CreditIssuer` then continuously loops over the buffers to process any packets that have been received and stored there by the other server thread. The only processing of packets done in the example is to count the number of bytes in each packet and keep a total count of bytes received from the client over this L2CAP channel.

```

public class CreditIssuer implements Runnable {

...

    public void run() {

        L2capPacketBuffer packetBuffer;
        int freedBuffers;
        //Issue one credit for each L2capPacketBuffer
        int totalCredits =
            L2capBuffers.NUMBER_OF_PACKET_BUFFERS;
        byte[] issueCreditsPacketPayload =
            CreditBased1WayXfer.int2ByteArray(totalCredits);
        try {
            conn.send(issueCreditsPacketPayload);
        } catch (IOException e) {
            System.out.println("IOException when issuing "
                + "initial credits");
        }
    }
}

```

```

        return;
    }
    while (!doneProcessing) {
        freedBuffers = 0;
        while ((packetBuffer =
            buffers.nextUsedPacketBuffer())
            != null) {
            count += packetBuffer.getNumBytesStored();
            packetBuffer.eraseStoredBytes();
            freedBuffers++;
        }

        if (freedBuffers > 0) {
            try {
                conn.send(CreditBased1WayXfer.int2ByteArray(
                    freedBuffers));
            } catch (IOException e) {
                System.out.println("IOException " +
                    e.getMessage());
            }
        }
        try {
            Thread.sleep(sleepTime);
        } catch (InterruptedException ignore) {
        }
    }
}
...
}

```

After the bytes in a buffered L2CAP packet have been counted, the `CreditIssuer` thread calls the application method `eraseStoredBytes()` to make this `L2capPacketBuffer` available to store future L2CAP packets. `CreditIssuer` keeps track of how many buffers it has freed up and uses the following statement to send an L2CAP packet to the client to issue one additional credit for each buffer freed:

```

conn.send(CreditBased1WayXfer.int2ByteArray(
    freedBuffers));

```

The statement `Thread.sleep(sleepTime)` at the bottom of the while loop makes it possible to introduce an arbitrary delay into the `CreditIssuer` thread. Experimenting with various delays shows how credit-based flow control adjusts the client's data-transmission rate to match the server's packet-processing rate.

8.4 Summary

L2CAP is one of three APIs for Bluetooth communication that are available to JABWT applications. JABWT provides a packet-based API for L2CAP as opposed to the stream-based APIs available for serial port and OBEX.

L2CAP communications are the right choice for an application if

- The application implements a Bluetooth profile that uses the L2CAP protocol and that Bluetooth profile does not use one of the higher-level protocols RFCOMM or OBEX, or
- The application implements a new custom protocol that is packet oriented

The `L2CAPConnection` interface provides methods for sending and receiving L2CAP packets over an L2CAP channel. Applications can use the connection string parameters `receiveMTU` and `transmitMTU` to define their requirements for maximum payload sizes of the L2CAP packets. This chapter presents four rules regarding MTU values for use in JABWT programs.

L2CAP provides no flow control, so JABWT applications that use the L2CAP API need to provide their own flow control. Without some form of flow control, L2CAP applications could encounter packet loss or deadlock. This chapter provides example code for two flow control schemes:

- Simple flow control that waits for a response to packet n before sending packet $n + 1$
- Credit-based flow control

References

1. Bluetooth SIG. Specification of the Bluetooth System, Core, v1.1, www.bluetooth.com, 2000.
2. J. Larmouth. *Understanding OSI*. International Thomson Publishing, www.isi.salford.ac.uk/books/osi/osi.html, 1996.
3. Bluetooth SIG. Specification of the Bluetooth System, Profiles v1.1, www.bluetooth.com, 2000.
4. Miller, B. A. and C. Bisdikian. *Bluetooth Revealed*, 2nd ed. Upper Saddle River: Prentice-Hall, 2001.
5. Bray, J., and C. F. Sturman. *Bluetooth 1.1: Connect without Cables*, 2nd ed. Upper Saddle River: Prentice-Hall, 2001.
6. Bluetooth SIG, Bluetooth Network Encapsulation Protocol (BNEP) Specification, Revision 1.0, 2003.
7. Bluetooth SIG, Hardcopy Cable Replacement Profile Interoperability Specification, Revision 1.0a, 2002.
8. Bluetooth SIG, Audio/Video Control Transport Protocol Specification, Revision 1.0, 2003.
9. Bluetooth SIG, Audio/Video Distribution Transport Protocol Specification, Revision 1.0, 2003.
10. Bluetooth SIG. Bluetooth Qualification Program Website qualweb.opengroup.org.
11. Topley, K. *J2ME in a Nutshell*. Sebastopol: O'Reilly, 2002.
12. Riggs, R., A. Taivalsaari, and M. VandenBrink. *Programming Wireless Devices with the Java™ 2 Platform, Micro Edition*. Boston: Addison-Wesley, 2001.
13. Java Community Process. J2ME Connected, Limited Device Configuration (JSR-30), www.jcp.org/jsr/detail/30.jsp, 2000 .
14. Java Community Process. J2ME Connected Device Configuration (JSR-36), www.jcp.org/jsr/detail/36.jsp, 2001.
15. Lindholm, T., and F. Yellin. *The Java™ Virtual Machine Specification, Second Edition*. Boston: Addison-Wesley, 1999.
16. Java Community Process. Mobile Information Device Profile for the J2ME Platform (JSR-37), www.jcp.org/jsr/detail/37.jsp, 2000.
17. Java Community Process. J2ME Foundation Profile (JSR-46), www.jcp.org/jsr/detail/46.jsp, 2001.
18. Java Community Process. Personal Profile Specification (JSR-62), www.jcp.org/jsr/detail/62.jsp, 2002.
19. Java Community Process. Personal Basis Profile Specification (JSR-129), www.jcp.org/jsr/detail/129.jsp, 2002.

20. Java Community Process. Java APIs for Bluetooth Wireless Technology (JSR-82), www.jcp.org/jsr/detail/82.jsp, 2002.
21. Java Community Process. Generic Connection Framework Optional Package for J2SE (JSR-197), www.jcp.org/jsr/detail/197.jsp, 2003.
22. JINI Networking Technology Home Page, www.sun.com/jini/.
23. Project JXTA Home Page, www.jxta.org.
24. Kumar, C B., and P. Kline. "Bringing the Benefits of Java to Bluetooth." *Embedded Systems, the European Magazine for Embedded Design* 6 (2002) no. 42.
25. Java Community Process., J2ME Connected, Limited Device Configuration 1.1 (JSR-139), www.jcp.org/jsr/detail/139.jsp, 2003.
26. Java Community Process. Mobile Information Device Profile 2.0 (JSR-118), jcp.org/jsr/detail/118.jsp, 2002.
27. Bluetooth SIG, Basic Printing Profile, Revision 0.95a, 2003.
28. Bluetooth SIG, Basic Imaging Profile, Revision 0.95c, 2003.
29. Infrared Data Association®, IrDA® Object Exchange Protocol–OBEX™, version 1.3, 2003.
30. Bluetooth SIG. Bluetooth Assigned Numbers, www.bluetooth.org/foundry/assignnumb/document/assigned_numbers.
31. Fowler, M., and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2nd ed. Boston: Addison-Wesley, 2000.
32. International Organization for Standardization. Code for the presentation of names of languages. ISO 639:1988 (E/F), Geneva, 1988.
33. Internet Assigned Numbers Authority. www.iana.org/assignments/character-sets.
34. The Internet Mail Consortium. vCard – The Electronic Business Card, Version 2.1, 1996. www.imc.org/pdi/vcard-21.txt.
35. The Open Group. DCE 1.1: Remote Procedure Call, Appendix A. Document Number C706, www.opengroup.org/onlinepubs/009629399/apdx.htm, Reading, UK, 1997.
36. Bluetooth SIG, Personal Area Networking Profile, Revision 1.0, 2003.
37. Bluetooth SIG, Bluetooth Extended Service Discovery Profile (ESDP) for Universal Plug and Play™ (UPnP™). Revision 0.95a, 2003.
38. Bluetooth SIG, Audio/Video Remote Control Profile, Revision 1.0, 2003.
39. Bluetooth SIG, Generic Audio/Video Distribution Profile, Revision 1.0, 2003.
40. Bluetooth SIG, Advanced Audio Distribution Profile, Revision 1.0, 2003.
41. Bluetooth SIG, Hands-Free Profile, Revision 1.0, 2003.
42. Williams, S., and I. Millar. "The IrDA platform," in *Insights into Mobile Multimedia Communication*, ed. D. Bull, C. Canagarajah, and A. Nix. San Francisco: Morgan Kaufmann, 1998. www.irda.org/design/irda_platform.pdf, Infrared Data Association, Walnut Creek, California, 1996.

Index

- ACL link, 10, 208
- Advanced Audio Distribution Profile, 209
- Agere, 4
- API
 - device discovery, 111–113
 - JABWT, 27–30
 - Java, 19–20
 - L2CAP, 211–220
 - OBEX, 84–87
 - programming. *See* Programming with the API
 - RFCOMM, 56–58
 - service discovery, 142–160
- Application profiles, 12
- Application programming interface. *See* API
- Applications. *See* Example applications
- Asynchronous connectionless (ACL), 10
- Attribute ID base value, 171
- Audio, 135
- Audio/video, 136
- Audio/Video Control Transport Protocol, 209
- Audio/Video Distribution Transport Protocol, 209
- Audio/Video Remote Control Profile, 209
- Authentication
 - Bluetooth, 53, 54
 - OBEX, 86, 102–106
- Authorization, 55–56
- Automatic generation of service records, 146–147, 161–167
- Base-plus-offset scheme, 170
- Baseband layer, 218
- Baseband packets, 208
- Basic profiles, 12
- BCC implementation, 330–331
- Blåtand, Harald, 4–5
- Blocking/non-blocking stack calls, 328–330
- Bluetooth
 - attributes, 3–4
 - client and server model, 37–41
 - historical overview, 4–5
 - link, 75
 - name, 4–5
 - native device, 322
 - overview, 3
 - profiles, 7, 11–13
 - protocols, 9–11
 - qualification process, 13–14
 - specification, 7
 - subsystem of JABWT, as, 29–30
 - timeline, 5
 - websites, 22
- Bluetooth address, 109
- Bluetooth authentication, 53
- Bluetooth client and server model, 37–41
- Bluetooth client application, 40
- Bluetooth Control Center (BCC), 30, 41–42, 43
- Bluetooth controller, 8
- Bluetooth Core specification, 25
- Bluetooth development kits, 243
- Bluetooth device, 110
- Bluetooth Extended Service Discovery Profile, 219
- Bluetooth host, 8
- Bluetooth host and device classification, 8
- Bluetooth-KOSI, 327
- Bluetooth-KVM, 326–327
- Bluetooth link, 75
- Bluetooth Network Encapsulation Protocol (BNEP), 11, 206, 209
- Bluetooth Object Push Profile, 173–181
- Bluetooth profile hierarchy, 13
- Bluetooth Profile specification, 25
- Bluetooth profiles, 7, 11–13
- Bluetooth protocol stack, 7–10, 8, 323–325
- Bluetooth protocols, 9–11
- Bluetooth qualification process, 13–14, 332
- Bluetooth radio module, 8
- Bluetooth security, 52–56. *See also* Security
- Bluetooth server application, 39–40
- Bluetooth service, 37
- Bluetooth SIG, 4, 11
- Bluetooth specification, 7
- Bluetooth system requirements, 29–30
- Bluetooth TCK, 333
- Bluetooth use cases, 6–7
- Bluetooth vision, 5–7
- bluetooth.api.version**, 116
- bluetooth.connected.devices.max**, 116
- bluetooth.connected.inquiry**, 116
- bluetooth.connected.inquiry.scan**, 116
- bluetooth.connected.page**, 116
- bluetooth.connected.page.scan**, 116

- bluetooth.l2cap.receiveMTU.max**, 116
- bluetooth.master.switch**, 116
- BluetoothProfileDescriptorList**, 150, 175
- bluetooth.sd.attr.retrieveable.max**, 116
- bluetooth.sd.trans.max**, 116
- bt12cap**, 213
- Cached devices, 113, 122
- Capturing, 135
- CDC, 16
- Class of device record, 112
- CLDC, 16, 35–36, 321–322
- CLDC + MIDP + Bluetooth architecture diagram, 36
- CLDC TCK, 333
- Client and server mode, 37–41
- Client connection strings, 66
 - L2CAP, 212–213
 - OBEX, 89
 - RFCOMM, 61
- Communication, 35
- Computer, 136
- Configurations, 15–16
- Conflict resolution, 42–43
- Connect-anytime services, 201–203
- Connected, Limited Device Configuration (CLDC), 16, 35–36, 321–322
- Connected Device Configuration (CDC), 16
- Connection
 - client (RFCOMM), 65–75
 - JABWT implementation, 327–328
 - L2CAP, 212–215
 - message client application, 305–310
 - OBEX, 88–89
 - server (RFCOMM), 60–65
- Connection-oriented channels, 208
- Connection strings
 - client. *See* Client connection strings
 - L2CAP, 212–215
 - OBEX, 88, 89
 - RFCOMM, 58–61, 66
 - server. *See* Server connection strings
- Connectionless channels, 208
- Context switching, 328, 329
- Cordless Telephony Profile, 209
- Credit-based flow control scheme, 234–240
- Custom applications, 211
- Custom services, 142
- Data element alternative, 149
- Data element sequence, 149
- DataElement, 147, 149, 192
- DATSEQ**, 180
- Deadlock scenario, 219
- Default attributes, 181
- Default service records, 147
- Development tools, 44–45, 243
- Device discovery, 109–138
 - API capabilities, 111–113
 - cached devices, 113, 122
 - DeviceClass**, 134–138
 - discoverable mode, 110, 114
 - limited vs. general inquiry, 110
 - local device, 114–125
 - major device class, 112–113, 136
 - major service class, 112, 134–136
 - minor device class, 113, 136–137
 - overview, 109
 - pre-known devices, 113, 122
 - programming with the API, 113–137
 - remote device, 131–134
 - simple device discovery, 120–125
 - starting an inquiry, 125–130
- Device management, 35
- Device properties, 41, 43
- DeviceClass**, 134–138, 160
- Dial-up Networking Profile, 206, 209
- Discoverable mode, 110, 114
- Discovery, 35
- DiscoveryAgent**, 114, 120
- DiscoveryAgent.GIAC**, 114, 125
- DiscoveryAgent.LIAC**, 114, 125
- DocumentationURL**, 150, 167
- Echo client/server, 61–75
- Echo L2CAP packets, 221–229
- 802.11b, 2
- Encryption, 55
- Ericsson, 4
- Event handling, 328
- Example applications, 243–318
 - OBEX message application, 289–318. *See also* OBEX message application overview, 243
 - Tic-Tac-Toe MIDlet, 244–288. *See also* Tic-Tac-Toe MIDlet
- Extended Service Discovery Profile, 209, 219
- Extended Systems, 20
- FAX Profile, 209
- Flow control, 218–219, 234–240
- Flush timeout, 210
- Foundation Profile (FP), 17
- Frequency hopping, 58
- Friendly name, 43
- GAP, 12
- GCF, 20
- General discoverable device, 110, 114
- General inquiry, 110, 125
- Generic Access Profile (GAP), 12
- Generic Audio/Video Distribution Profile, 209
- Generic Connection Framework (GCF), 20
- Generic Object Exchange Profile (GOEP), 12
- Guidgen.exe**, 201
- Hands-Free Profile, 209
- Hardcopy Cable Replacement Profile, 209, 219, 220

- Hardcopy Control Channel, 209
- Hardcopy Notification Channel, 209
- HCI interface, 8, 10
- Headset Profile, 209
- “Hello, World” JABWT application, 45–50
- High-end routers, 1
- Historical overview, 4–5
- Host Controller Interface (HCI), 8
- IBM, 4, 20
- IconURL**, 150
- IEEE 802.11, 2
- Illustrations. *See* Example applications
- Imaging, 136
- Implementation components, 326
- Implementing JABWT, 319–335
 - adding J2ME/Bluetooth support, 322–325
 - BCC implementation, 330–331
 - blocking/nonblocking stack calls, 328–330
 - Bluetooth-KOSI, 327
 - Bluetooth-KVM, 326–327
 - CLDC, 321–322
 - connections, 327–328
 - context switching, 328, 329
 - event handling, 328
 - KVM extensions, 326–327
 - KVM-stack interface, 327–330
 - OBEX implementation, 331, 332
 - porting process, 320–322
 - RI, 333
 - stack, 323–325
 - TCK agent, 333–334
 - TCK compliance, 332–334
 - who should do it, 319
- Impronto Simulator, 44, 45, 243
- Information, 135
- Information age, 1
- Infrared, 2
- Infrared Data Association (IrDA), 2, 3
- Infrared Object Exchange protocol (IrOBEX), 77, 78. *See also* OBEX
- Inquiry, 109, 110, 125. *See also* Device discovery
- Inquiry access codes, 125
- Instant post card, 7
- Intel, 4
- Intercom Profile, 209
- Internet websites
 - Bluetooth wireless technology, 22
 - J2ME, 22
 - JSR-82, 22
 - qualification process, 14
 - RI/TCKs, 333
- IrDA, 2, 3
- IrOBEX, 77, 78. *See also* OBEX
- J2EE, 14
- J2ME, 2, 14–18
 - components, 16
 - configurations, 15–16
 - optional packages, 17–18
 - overview, 14
 - profiles, 16–17
 - websites, 22
- J2ME configurations, 15–16
- J2ME development tools, 44–45, 243
- J2ME device requirements, 28–29
- J2ME Wireless Toolkit, 44, 45, 243
- J2SE, 14, 20
- JABWT, 19, 23–24
 - API characteristics/ hardware requirements, 27–30
 - architecture, 35–41
 - BCC, 41–42, 43
 - Bluetooth stack, 33
 - Bluetooth system requirements, 29–30
 - conflict resolution, 42–43
 - device properties, 41, 43
 - goals, 23–24
 - implementation, 319–335. *See also* Implementing JABWT
 - J2ME device requirements, 28–29
 - Java packages, 34, 37
 - key characteristics, 32, 34
 - sample application (“Hello World”), 45–50
 - scope, 30–32
 - specification, 25, 27–28
 - system properties, 41
 - target devices, 24, 25
 - use cases, 26–27
- JABWT packages, 34
- JABWT specification
 - characteristics, 27–28
- JABWT TCK test, 333
- JABWT use cases, 26–27
- Java API, 19–20
- Java APIs for Bluetooth Wireless Technology. *See* JABWT
- Java Community Process (JCP), 19
- Java language, 18–19
- Java packages, 37
- Java Specification Request (JSR), 19
- Java virtual machine, 15
- javax.bluetooth**, 34, 37
- javax.microedition.io**, 37
- javax.obex**, 34, 37
- Java 2, Micro Edition. *See* J2ME
- Java 2 Platform, Enterprise Edition (J2EE), 14
- Java 2 Platform, Standard Edition (J2SE), 14, 20
- Java 2 platforms, 15
- JCP, 19
- Jini, 20–22
- JSR, 19
- JSR-82, 20, 22, 333
- JSR-82 TCK setup, 333, 334
- JSR-82 TCK test, 333, 334
- JSR-139, 24
- JSR-197, 20, 24, 29
- JXTA, 20–22
- Kilobyte Virtual Machine (KVM), 16
- Kiosk, 26
- KOSI, 326
- KToolBar, 45
- KVM, 16
- KVM Extensions, 326, 327

- KVM operating system interface (KOSI), 326
- KVM-stack interface, 327–330
- L2CAP, 205–241
 - API, 211–220
 - application, 220
 - Bluetooth stack, and, 206
 - channel configuration, 215–218
 - channels/packets, 207–208
 - clients/servers, 233
 - custom applications, 211, 220
 - deadlock scenario, 219
 - echo L2CAP packets, 221–229
 - flow control, 218–219, 234–240
 - flush timeout, 210
 - interfaces defined in JABWT, 211–212
 - MTU, 207, 215–218, 229–233
 - open a connection, 212–215
 - profiles/protocols, 206, 209
 - programming with the API, 221–240
 - PSM, 213–215
 - service record, 153–155
 - user interface for MTU values, 229–233
 - why used, 208–211
- L2CAP applications, 220
- L2CAP channel configuration, 215–218
- L2CAP channels, 207–208
- L2CAP clients/servers, 233
- L2CAP packets, 207–208
- L2CAPConnection**, 211
- L2CAPConnectionNotifier**, 211
- LAN Access Profile, 206
- LAN/network access point, 136
- LanguageBaseAttributeIDList**, 171
- Licensee closed classes (LCCs), 18
- Licensee open classes (LOCs), 18
- Limited discoverable device, 110, 114
- Limited Discoverable Mode, 135
- Limited inquiry, 110, 125
- Link, 75
- Link Manager Protocol (LMP), 10
- List-of-lists structure, 152
- LMP, 10
- LocalDevice**, 114
- LocalDevice.getProperty**, 41, 116
- LOCs, 18
- Logical link control and adaptation protocol. *See* L2CAP
- Major device class, 112–113, 136
- Major service class, 112, 134–136
- Master and slave configuration, 57–58
- Maximum transmission unit (MTU), 207, 215–218, 229–233
- Message application. *See* OBEX message application
- Message client application, 301–318
- Message server application, 289–301
- MIBenum, 171
- Microsoft, 4
- MIDlet, 17
- MIDP, 17, 35–36
- MIDP device, 323
- Minor device classes, 113, 136–137
- Miscellaneous device class, 136
- Mitsubishi Electric, 20
- Mobile Information Device Profile (MIDP), 17, 35–36
- Modifying a service record, 158–160, 167–170
- Motorola, 4, 20, 333
- MTU, 207, 215–218, 229–233
- MTU mismatch pitfall, 217
- Native Bluetooth Device, 322
- Networking, 135
- Newbury Networks, 20
- Nokia, 4, 20
- Non-blocking stack calls, 328–330
- Not discoverable, 110, 114
- OBEX, 77–108
 - API, 84–87
 - authentication, 86, 102–106
 - basic operations, 79
 - common headers, 80
 - connection strings, 88, 89
 - establishing a connection, 88–89
 - example session, 81–84
 - headers, 80, 90–92
 - overview, 77–78
 - programming with the API, 87–106
 - protocol description, 79–80
 - receiving request from client, 98–102
 - sending request to server, 92–98
 - sessions, 79–84, 107
 - special operations, 80
 - use cases, 78–79
- OBEX authentication, 86, 102–106
- OBEX CONNECT operation, 81
- OBEX connection string, 88, 89
- OBEX DISCONNECT operation, 83
- OBEX GET operation, 96–97
- OBEX implementation, 331, 332
- OBEX message application, 289–318
 - establishing connection to server, 305–310
 - message client application, 301–318
 - message server application, 289–301
 - receiving messages from client, 296–298
 - retrieving message from server, 314–318
 - sending message to server, 310–314

- OBEX message application
(*continued*)
 - sending messages to client, 299–301
- OBEX PUT operation, 83, 95–96
- OBEX SETPATH operation, 82
- OBEX TCK, 333
- Object Push service record, 173–181
- Object Transfer, 135
- onXXX() method, 106, 107
- Optional packages, 17–18
- Pairing, 52–53
- PAN, 5
- PAN profile, 206, 209
- Parthus Technologies, 20
- PauseCanvas**, 252–258
- PauseTimerTask**, 252–258
- PBP, 17
- PBX, 1
- Peer-to-peer applications, 40
- Peer-to-peer networking, 26
- Peripheral, 136
- Personal area network (PAN), 5
- Personal Area Networks, 209
- Personal Basic Profile (PBP), 17
- Personal Profile (PP), 17
- PersonalJava, 17, 20
- Phone, 136
- Porting process, 320–322
- Positioning, 135
- PP, 17
- Pre-known devices, 113, 122
- Predefined devices, 113
- Printing, 79
- Printing service, 37
- Private branch exchange (PBX), 1
- Profiles
 - application, 12
 - basic, 12
 - Bluetooth, 7, 11–13
 - J2ME, 16–17
 - L2CAP, 206, 209
 - transport, 12
- Programming with the API. *See also* Example applications
 - device delivery, 113–137
 - L2CAP, 221–240
 - OBEX, 87–106
 - RFCOMM, 58–75
 - service discovery, 161–203
- Protocol and service
 - multiplexer (PSM), 154, 213–215
- Protocol stack, 7–10, 323–325
- ProtocolDescriptorList**, 150, 152
- Protocols
 - Bluetooth, 9–11
 - L2CAP, 206, 209
- Provisioning, 78–79
- PSM, 154, 213–215
- Qualification process, 13–14, 332
- Radio frequency (RF), 2
- Reference implementation (RI), 333
- RemoteDevice**, 131–134
- Rendering, 135
- Research in Motion, 20
- RF, 2
- RFCOMM, 11, 51–76
 - API capabilities, 56–58
 - Bluetooth security, 52–56
 - establishing client connection, 65–75
 - establishing server connection, 60–65
 - master and slave configuration, 57–58
 - overview, 51–56
 - programming with the API, 58–75
 - valid parameters for connection strings, 59
- RI, 333
- Rococo Impronto Simulator, 44, 45
- Rococo Software, 20, 44
- Run-before-connect services, 143–144
- SCO link, 10
- SDAP, 12, 209
- SDDB, 38
- SDP, 10
- searchServices()**, 181–190
- Security, 52–56
 - authentication, 53
 - authorization, 55–56
 - encryption, 55
 - pairing, 52–53
- Serial Port Profile (SPP), 12, 51, 209
- Serial ports, 11
- Server connection strings, 60–61
 - L2CAP, 214–215
 - OBEX, 89
 - RFCOMM, 60–61
- Service class, 134–136
- Service discovery, 139–204
 - API capabilities, 142–160
 - connect-anytime services, 201–203
 - custom services, 142
 - DataElement, 147, 149, 192
 - device service classes, 160
 - overview, 139–142
 - programming with the API, 161–203
 - retrieving additional attributes, 194–198
 - run-before-connect services, 143–144
 - service record. *See* Service record
 - service registration, 141–142, 161, 203
 - service search, 181–190
 - simple programming example, 199
 - UUID. *See* UUID
- Service Discovery Application Profile (SDAP), 12, 209
- Service discovery database (SDDB), 38
- Service Discovery Protocol (SDP), 10
- Service record, 141
 - add, to SDDB, 155–157, 158
 - attribute IDs, 150
 - automatic generation, 146–147, 161–167
 - DataElement, 147, 149, 192

- Service record (*continued*)
 - defined, 143
 - L2CAP, for, 153–155
 - modifications to, 158–160, 167–170
 - multiple language, 170–173
 - OBEX over RFCOMM, for, 155
 - Object Push, 173–181
 - remove, from SDDb, 157, 158
 - serial port, for, 147–153
 - working with, 190–194
- Service registration, 141–142, 161, 203
- Service search, 181–190
- ServiceAvailability**, 150
- ServiceClassIDList**, 150, 151
- ServiceDescription**, 150, 167
- ServiceID**, 150
- ServiceInfoTimeToLive**, 150
- ServiceName**, 150, 152–153
- ServiceRecordHandle**, 150, 153
- SERVICE_SEARCH_COMPLETED**, 183
- SERVICE_SEARCH_DEVICE_NOT_REACHABLE**, 183
- SERVICE_SEARCH_ERROR**, 183
- SERVICE_SEARCH_NO_RECORDS**, 183
- SERVICE_SEARCH_TERMINATED**, 183
- Sharp Laboratories of America, 20
- Short-form UUIDs, 152
- Short-range wireless standards, 2
- Smart Fusion, 20
- Smart Network Devices, 20
- Sony Ericsson Mobile Communications, 20
- SPP, 12, 51, 209
- Stack. *See* Bluetooth protocol stack
- Stream-based protocol, 211
- Sun Microsystems, 20, 333
- Supported Formats List**, 177
- Symbian, 20
- Synchronous connection oriented (SCO), 10
- System properties, 41
- TCK agent, 333–334
- TCK compliance, 332–334
- TCS-BIN, 209, 210
- TCS binary, 11
- Technology Compatibility Kit (TCK) compliance, 332–334
- Telecordia, 20
- Telephony, 135
- Telephony Control Protocol Specification, Binary (TCS binary or TCS-BIN), 11, 209, 210
- Three-in-one phone usage model, 6
- 3Com, 4
- Tic-Tac-Toe MIDlet, 244–288
 - creating the game thread, 258–267
 - implementation, 245–252
 - overview, 244–245
 - PauseCanvas**, 252–258
 - PauseTimerTask**, 252–258
 - playing the game, 267–288
 - TTTCanvas**, 258–267
 - TTTGame**, 258–267
- Timeline, 5
- Toshiba, 4
- Transport profiles, 12
- TTTCanvas**, 258–267
- TTTGame**, 258–267
- Unicode Transformation Format 8 (UTF-8), 171
- Universally unique identifier. *See* UUID
- Upper-layer stack, 8
- Use cases
 - Bluetooth, 6–7
 - JABWT, 26–27
- User interface for MTU values, 229–233
- UTF-8, 171
- UUID, 143, 149
 - conversion, 151
 - generating, 200–201
 - short-form, 152
 - 16-bit vs. 128-bit, 151
- uuidgen**, 200
- Vaultus, 20
- Vending machine, 27
- Voice over Internet Protocol (VoIP), 2
- Websites. *See* Internet websites
- Wireless communication technologies, 3
- Wireless connectivity, 1–3
- Wireless local area network (WLAN), 2, 3
- Wireless telematics, 6–7
- Wireless Toolkit, 44, 45, 243
- WLAN, 2, 3
- Zucotto, 20