Palm OS®
Programming Bible

Palm OS® Programming Bible

Lonnon R. Foster

IDG Books Worldwide, Inc.
An International Data Group Company

Foster City, CA ✦ Chicago, IL ✦ Indianapolis, IN ✦ New York, NY

Palm OS® Programming Bible

Published by
IDG Books Worldwide, Inc.
An International Data Group Company
919 E. Hillsdale Blvd., Suite 400
Foster City, CA 94404
www.idgbooks.com (IDG Books Worldwide Web site)

Copyright © 2000 IDG Books Worldwide, Inc. All rights reserved. No part of this book, including interior design, cover design, and icons, may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording, or otherwise) without the prior written permission of the publisher.

ISBN: 0-7645-4676-7

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

1B/QW/RR/QQ/FC

Distributed in the United States by IDG Books Worldwide, Inc.

Distributed by CDG Books Canada Inc. for Canada; by Transworld Publishers Limited in the United Kingdom; by IDG Norge Books for Norway; by IDG Sweden Books for Sweden; by IDG Books Australia Publishing Corporation Pty. Ltd. for Australia and New Zealand; by TransQuest Publishers Pte Ltd. for Singapore, Malaysia, Thailand, Indonesia, and Hong Kong; by Gotop Information Inc. for Taiwan; by ICG Muse, Inc. for Japan; by Intersoft for South Africa; by Eyrolles for France; by International Thomson Publishing for Germany, Austria, and Switzerland; by Distribuidora Cuspide for Argentina; by LR International for Brazil; by Galileo Libros for Chile; by Ediciones ZETA S.C.R. Ltda. for Peru; by WS Computer Publishing Corporation, Inc., for the Philippines; by Contemporanea de Ediciones for Venezuela; by Express Computer Distributors for the Caribbean and West Indies; by Micronesia Media Distributor, Inc. for Micronesia; by Chips Computadoras S.A. de C.V. for Mexico; by Editorial Norma de Panama S.A. for Panama; by American Bookshops for Finland.

For general information on IDG Books Worldwide's books in the U.S., please call our Consumer Customer Service department at 800-762-2974. For reseller information, including discounts and premium sales, please call our Reseller Customer Service department at 800-434-3422.

For information on where to purchase IDG Books Worldwide's books outside the U.S., please contact our International Sales department at 317-596-5530 or fax 317-572-4002.

For consumer information on foreign language translations, please contact our Customer Service department at 800-434-3422, fax 317-572-4002, or e-mail rights@idgbooks.com.

For information on licensing foreign or domestic rights, please phone +1-650-653-7098.

For sales inquiries and special prices for bulk quantities, please contact our Order Services department at 800-434-3422 or write to the address above.

For information on using IDG Books Worldwide's books in the classroom or for ordering examination copies, please contact our Educational Sales department at 800-434-2086 or fax 317-572-4005.

For press review copies, author interviews, or other publicity information, please contact our Public Relations department at 650-653-7000 or fax 650-653-7500.

For authorization to photocopy items for corporate, personal, or educational use, please contact Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, or fax 978-750-4470.

Library of Congress Cataloging-in-Publication Data

Foster, Lonnon R., 1972-
Palm OS programming Bible / Lonnon R. Foster.
p. cm.
ISBN 0-7645-4676-7 (alk. paper)
1. Palm OS. 2. PalmPilot (Computer)--Programming.
I. Title.
QA76.76.O63 F685 2000
005.26'8--dc21 00-044954

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND AUTHOR HAVE USED THEIR BEST EFFORTS IN PREPARING THIS BOOK. THE PUBLISHER AND AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS BOOK AND SPECIFICALLY DISCLAIM ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. THERE ARE NO WARRANTIES WHICH EXTEND BEYOND THE DESCRIPTIONS CONTAINED IN THIS PARAGRAPH. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES REPRESENTATIVES OR WRITTEN SALES MATERIALS. THE ACCURACY AND COMPLETENESS OF THE INFORMATION PROVIDED HEREIN AND THE OPINIONS STATED HEREIN ARE NOT GUARANTEED OR WARRANTED TO PRODUCE ANY PARTICULAR RESULTS, AND THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY INDIVIDUAL. NEITHER THE PUBLISHER NOR AUTHOR SHALL BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGES, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES.

Trademarks: All brand names and product names used in this book are trade names, service marks, trademarks, or registered trademarks of their respective owners. IDG Books Worldwide is not associated with any product or vendor mentioned in this book.

IDG BOOKS WORLDWIDE® is a registered trademark or trademark under exclusive license to IDG Books Worldwide, Inc. from International Data Group, Inc. in the United States and/or other countries.

ABOUT IDG BOOKS WORLDWIDE

Welcome to the world of IDG Books Worldwide.

IDG Books Worldwide, Inc., is a subsidiary of International Data Group, the world's largest publisher of computer-related information and the leading global provider of information services on information technology. IDG was founded more than 30 years ago by Patrick J. McGovern and now employs more than 9,000 people worldwide. IDG publishes more than 290 computer publications in over 75 countries. More than 90 million people read one or more IDG publications each month.

Launched in 1990, IDG Books Worldwide is today the #1 publisher of best-selling computer books in the United States. We are proud to have received eight awards from the Computer Press Association in recognition of editorial excellence and three from Computer Currents' First Annual Readers' Choice Awards. Our best-selling ...*For Dummies*® series has more than 50 million copies in print with translations in 31 languages. IDG Books Worldwide, through a joint venture with IDG's Hi-Tech Beijing, became the first U.S. publisher to publish a computer book in the People's Republic of China. In record time, IDG Books Worldwide has become the first choice for millions of readers around the world who want to learn how to better manage their businesses.

Our mission is simple: Every one of our books is designed to bring extra value and skill-building instructions to the reader. Our books are written by experts who understand and care about our readers. The knowledge base of our editorial staff comes from years of experience in publishing, education, and journalism — experience we use to produce books to carry us into the new millennium. In short, we care about books, so we attract the best people. We devote special attention to details such as audience, interior design, use of icons, and illustrations. And because we use an efficient process of authoring, editing, and desktop publishing our books electronically, we can spend more time ensuring superior content and less time on the technicalities of making books.

You can count on our commitment to deliver high-quality books at competitive prices on topics you want to read about. At IDG Books Worldwide, we continue in the IDG tradition of delivering quality for more than 30 years. You'll find no better book on a subject than one from IDG Books Worldwide.

IDG BOOKS WORLDWIDE

John Kilcullen
Chairman and CEO
IDG Books Worldwide, Inc.

Eighth Annual Computer Press Awards 1992

Ninth Annual Computer Press Awards 1993

Tenth Annual Computer Press Awards 1994

Eleventh Annual Computer Press Awards 1995

IDG is the world's leading IT media, research and exposition company. Founded in 1964, IDG had 1997 revenues of $2.05 billion and has more than 9,000 employees worldwide. IDG offers the widest range of media options that reach IT buyers in 75 countries representing 95% of worldwide IT spending. IDG's diverse product and services portfolio spans six key areas including print publishing, online publishing, expositions and conferences, market research, education and training, and global marketing services. More than 90 million people read one or more of IDG's 290 magazines and newspapers, including IDG's leading global brands — Computerworld, PC World, Network World, Macworld and the Channel World family of publications. IDG Books Worldwide is one of the fastest-growing computer book publishers in the world, with more than 700 titles in 36 languages. The "...For Dummies®" series alone has more than 50 million copies in print. IDG offers online users the largest network of technology-specific Web sites around the world through IDG.net (http://www.idg.net), which comprises more than 225 targeted Web sites in 55 countries worldwide. International Data Corporation (IDC) is the world's largest provider of information technology data, analysis and consulting, with research centers in over 41 countries and more than 400 research analysts worldwide. IDG World Expo is a leading producer of more than 168 globally branded conferences and expositions in 35 countries including E3 (Electronic Entertainment Expo), Macworld Expo, ComNet, Windows World Expo, ICE (Internet Commerce Expo), Agenda, DEMO, and Spotlight. IDG's training subsidiary, ExecuTrain, is the world's largest computer training company, with more than 230 locations worldwide and 785 training courses. IDG Marketing Services helps industry-leading IT companies build international brand recognition by developing global integrated marketing programs via IDG's print, online and exposition products worldwide. Further information about the company can be found at www.idg.com. 1/26/00

Credits

Acquisitions Editors
John Osborn
Greg Croy

Project Editor
Eric Newman

Technical Editor
JB Parrett

Copy Editors
Mildred Sanchez
S. B. Kleinman

Permissions Editor
Jessica Montgomery

Media Dev. Manager
Laura Carpenter

Media Dev. Supervisor
Rich Graves

Senior Permissions Editor
Carmen Krikorian

Media Dev. Coordinator
Marisa Pearman

Media Development Specialists
Megan Decraene
Brock Bigard

Project Coordinators
Joe Shines
Danette Nurse

Graphics and Production Specialists
Robert Bihlmayer
Jude Levinson
Michael Lewis
Victor Pérez-Varela
Ramses Ramirez

Quality Control Technician
Dina F Quan

Illustrators
Rashell Smith
Karl Brandt
Gabriele McCann

Proofreading and Indexing
York Production Services

Cover Image
Evan Deerfield

About the Author

Lonnon R. Foster is a freelance programmer who has spent the past seven years creating desktop applications, database front ends, Web sites, communications software, technical documentation, and handheld applications. He has been developing Palm OS applications almost as long as the platform has existed, starting with his first Pilot 5000. Lonnon fills his sparse free time with tactical tabletop gaming, recreational Perl coding, and reading everything he can get his hands on.

For Elisabeth, who believed from the very start that I could do it

Foreword

In 1992, Palm Computing (now Palm, Inc.) was founded. The rest is history. Of course, there's a lot more to it. Many good decisions were made, and many bad designs were thrown out. Palm started as a software company intending to influence existing handheld manufacturers with its easy-to-use software and synchronization technologies. The company soon realized two things. The first was that the hardware manufacturers didn't seem to believe in Palm's philosophy. The second was that it was difficult to make a viable business just writing software for this small market. Palm realized that its first take at this company was not the right one and decided to become the master of its destiny. The name was Pilot. Palm changed the focus of its business virtually overnight. This is a lot like in development, where you find that the first take is rarely the best one. I have often gone back through my code and discovered some wacky designs. Only then do I discover the best architectural design, giving me fewer bugs and the best feature set. In Palm, this is known as the sweet spot and it is a zone that few developers enter and fewer leave successfully. However, Palm not only entered this zone, it now owns it. This accounts for most of Palm's success. There are more powerful devices out there (PocketPC), and there are more connected ones (Cybiko). Yet all of those devices combined still do not add up to the amount of devices that Palm has shipped. Why? Because Palm has found the sweet spot, the spot where functionality and ease of use conflict, and difficult decisions are made to remove functionality (something that even Microsoft hasn't realized). Other companies have discovered this zone and understand what is necessary in the handheld market. That's the reason why these very wise companies have licensed the Palm OS. Companies like Sony, Handspring, and Symbol have all realized the importance of Palm's philosophy, where ease of use and power are not necessarily mutually exclusive, and where the end-user experience is always top notch.

At this writing, there are more than 100,000 Palm OS developers. This development community is as diverse as the world of computing: from high school kids writing in Basic to skilled university researchers writing in C, from enterprise developers writing in Java to commercial developers writing in C++. From Iceland to Argentina, these developers have realized not only that Palm is the best-selling handheld in the world but also that the Palm OS is an open operating system, intuitive to program and very clearly documented. I don't expect to see developers evangelize the virtues of *Palm OS® Programming Bible*, but they should. Lonnon Foster has proven that he understands not only the fundamentals of Palm OS programming

but the sweet spot of writing as well. This book covers everything from building forms and menus to programming sounds and color. The examples are complete yet amazingly simple. Not only will you learn to program the Palm OS, you will understand the philosophy that has made Palm successful, and in doing so, I hope that you will be successful too.

Phillip B. Shoemaker
Director, Development Tools
Palm, Inc.

Preface

The convenience, power, and ease of use of Palm OS handheld devices make them attractive to a wide variety of users. Handheld devices running the Palm OS have found their way into the shirt pockets of doctors, lawyers, sales personnel, business professionals, and other segments of society not normally given to using small electronic gadgetry. With more than 100,000 registered developers, the Palm OS has also proven to be popular with software authors, which is where this book comes in.

Palm OS® Programming Bible will show you how to create applications for Palm's popular line of handheld organizers, as well as third-party devices that also run the Palm OS. In addition, this book covers creation of Web clipping applications for the Palm VII/VIIx (and other wireless-equipped Palm OS handhelds) to allow wireless connection to the Internet. You will also find material on writing conduit programs to synchronize data between a Palm OS handheld and a desktop computer. Whether you are a developer for a large organization that is integrating Palm OS handhelds into its sales force or a hobbyist who wants to get the most from your organizer, you will find this book to be a useful guide to creating software for the Palm OS platform.

The primary focus of this book is Palm OS development in the C language, using CodeWarrior for Palm Computing Platform or the GNU PRC-Tools as a development environment. Other tools exist for developing Palm OS applications (and an overview of other such tools is available in the appendixes), but these two environments are popular with the largest developer audience, and they offer the most complete access to the many features of the Palm OS and the handhelds that run it.

Who Should Read This Book

This book was written with the experienced C programmer in mind. If you know nothing at all about Palm OS programming, this book will get you started with the fundamentals, teaching you how the Palm OS works, showing you the tools available for Palm OS development, and providing you with tips to make your own applications work seamlessly within Palm's programming guidelines.

Even if you already have delved into the world of creating Palm OS applications, you will find this book a useful resource, because it covers almost every aspect of Palm OS development in depth. The Palm OS is very large, and this book can serve as a guide to exploring those parts of the operating system that you have not yet dealt with.

If you wish to create Web clipping applications for the Palm VII/VIIx, you will need to know the basics of HTML and Web page creation to make the Palm Query Applications (PQAs) that reside on the handheld and provide a client-side connection to the Internet. To create the server side of a Web clipping application, you will need to be familiar with some sort of system for creating dynamic Web content, such as Perl CGI or Active Server Pages.

Conduit programming requires knowledge of C++, as well as a working knowledge of how to create desktop applications for either Windows or the Mac OS.

How This Book Is Organized

This book is organized into seven parts, plus four appendixes.

Part I: Getting Started

This first part of the book discusses the philosophy behind the Palm OS and introduces fundamental concepts behind the inner workings of the operating system.

Part II: Creating Palm OS Applications

The chapters in Part II cover the mechanics of making a Palm OS application. This section begins with a tour of the tools for Palm OS programming, then gets you off the ground with a simple "Hello, world" application and finally presents tools and techniques for every programmer's favorite part of writing an application: debugging.

Part III: Programming the Palm OS

The third part of this book focuses on actually writing the code to make a Palm OS application work. Starting with chapters on creating the resources that form the structure of an application, this part continues by showing how to actually make the program do something, from interacting with the user to manipulating text.

Part IV: Storing Information on the Handheld

Part IV shows how to store and retrieve application data. It starts with the big picture, showing how to interact with databases, then moves in for a closer look at the records that make up a database.

Part V: Communicating Outside the Handheld

The chapters in Part V cover the myriad methods a Palm OS handheld can use to communicate with the outside world, including infrared beaming, serial communication, and wireless Web clipping.

Part VI: Synchronizing Data with the Desktop

Part VI introduces the concepts behind the HotSync Manager, which allows a Palm OS handheld to synchronize its applications with desktop data sources. The section continues by showing how to write a conduit to customize the interaction between a Palm OS database and desktop applications.

Part VII: Advanced Programming Topics

In Part VII you will find various topics that do not come up as often as do the others in Palm OS programming, including managing color, creating large applications, and creating user interface elements dynamically while an application is running.

Appendixes

The final section of the book is devoted to four appendixes:

- **Appendix A, "Palm OS API Quick Reference,"** is a quick guide to the most common functions, data structures, and constants used in the Palm OS, including prototypes for Palm OS functions.
- **Appendix B, "Finding Resources for Palm OS Development,"** is a list of helpful resources for Palm OS developers.
- **Appendix C, "Developing in Other Environments,"** is a survey of alternative tools for Palm OS development.
- **Appendix D, "What's on the CD-ROM?"** describes the contents of the CD-ROM that accompanies this book, which features sample code and applications from the book, as well as all the tools a developer needs to get started with Palm OS development.

In addition, I've included a glossary at the end of the book.

How to Approach This Book

Readers who are completely new to Palm OS development will get the most benefit from this book by reading Parts I and II first to get a good handle on how the Palm OS works and how to use CodeWarrior or the PRC-Tools. Then look at Part III to learn what to do with those tools to make an actual application, and follow up with Part IV to learn how to save and retrieve an application's data. The other parts of the book may be read in any order; pick a topic of interest, and start reading.

For readers who have already done some Palm OS development, Part I probably will be material you already know. Part II can be useful if you use either CodeWarrior or the PRC-Tools, and you want to see how the other set of tools works in comparison with what you are using, and in particular, Chapter 5, "Debugging Your Program," contains useful tips for any Palm OS developer. Parts III and IV will serve as useful references to parts of the operating system that you may or may not already be familiar with, and later chapters introduce other parts of the Palm OS that are not strictly required by most applications.

Developers interested in creating Web clipping applications can go straight to Chapter 16, "Creating Web Clipping Applications." Most Web clipping development requires only a working knowledge of HTML, and if you run across any Palm OS–specific concepts you are not familiar with, a quick look through Part I should serve to resolve any confusion.

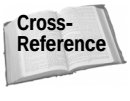
Anyone interested in creating conduits should first be familiar with the conceptual information in Part I. After you understand the concepts behind the Palm OS, turn to Part VI to learn how to hook the Palm OS up to a desktop computer.

Conventions Used in This Book

Each chapter in this book begins with a heads-up of the topics covered in the chapter and ends with a summary of what you should have learned by reading the chapter.

Throughout this book, you will find icons in the margins that highlight special or important information. Keep an eye out for the following icons:

A Caution icon indicates a procedure that could potentially cause difficulty or even data loss; pay careful attention to Caution icons to avoid common and not-so-common programming pitfalls.



Cross-Reference icons point to additional information about a topic, which you can find in other sections of the book.

A Note icon highlights interesting or supplementary information and often contains extra bits of technical information about a subject.

The On the CD-ROM icon is a pointer to information, tools, or programs available on the CD-ROM that accompanies this book.

Tip icons draw attention to handy suggestions, helpful hints, and useful pieces of advice.

In addition to the icons listed previously, the following typographical conventions are used throughout the book:

- Code examples appear in a `fixed width font`.
- Other code elements, such as data structures and variable names, appear in `fixed width`.
- File names and World Wide Web addresses (URLs) also appear in `fixed width`.
- Function and macro names are in **bold**.
- The first occurrence of an important term in a chapter is highlighted with *italic* text. *Italic* is also used for placeholders — for example, `ICON <icon file name>`, where `<icon file name>` represents the name of a bitmap file.
- A menu command is indicated in hierarchical order, with each menu command separated by an arrow. For example, File ➪ Open means to click the File command on the menu bar, and then select Open.
- Keyboard shortcuts are indicated with the following syntax: Ctrl+C.

What Is a Sidebar?

Topics in sidebars provide additional information. Sidebar text contains discussion that is related to the main text of a chapter, but not vital to understanding the main text.

Acknowledgments

Few books of this size and scope are ever the work of a single individual, and this one is no exception. I owe a debt of gratitude to many people for their help and encouragement in writing this book.

First and foremost, thanks to my acquisitions editors, John Osborn and Greg Croy, as well as my agent, Neil Salkind, for giving me the opportunity to write this book. Thanks also go to Erica Sadun, who had the whole idea in the first place.

I want to extend special thanks to Eric Newman, whose hard work as development editor was an incalculable asset in creating this book. Not only did he help wrestle the text of the book into a more focused and organized whole, he kept up the faith even in the face of slipping deadlines and the author's trip to England in the middle of the writing. A Palm enthusiast himself, Eric also kept me abreast of happenings in the Palm OS world that I would otherwise have missed during the busiest months of writing.

In addition, my thanks go out to the book's technical reviewer, JB Parrett, whose expertise and passion for good user interface improved the quality of the book immeasurably. I would also like to thank Mildred Sanchez and S. B. Kleinman for their copyediting as well as apologize for any gross abuses of the English language they were forced to endure.

A big thank you to the production team at IDG Books, including Gabriele McCann, Linda Marousek, Danette Nurse, Ronald Terry, and Mary Jo Weis, whose efforts behind the scenes made it possible for a random assortment of Word documents and bitmap images to transform magically into the printed copy you now hold. More thanks go to Jessica Montgomery, Lenora Chin Sell, and Carmen Krikorian, the media production folks who secured legal permissions for third-party CD-ROM content, and a very special thank you to Joe Kiempisty for his able assistance and patience in getting my own source code onto the CD.

I owe a lot to Lisa Rathjens and Ryan Robertson of Palm. Lisa's loan of time and software helped produce much better CodeWarrior support in this book, and Ryan's explanation of how tables work and answers to random technical questions ensured that some of the more confusing aspects of the Palm OS were made much clearer. Thanks also go to Christine Ackerman and Neil Shepherd of Oracle, Ivan Phillips of Pendragon Software, Ray Combs of PUMATECH, Dan Simon of Qualcomm, and Chris Ciervo of Symbol.

Special thanks go to Ken Martin, Gene Thompson, and Steve Feldon, whose commentary as “beta testers” was very useful in making sure that I have not ignored first-time Palm OS programmers in this volume. Steve also deserves my gratitude for introducing me to handheld computing, first with his old Newton (which he wouldn’t let me touch for fear of messing up the handwriting recognition), then again with his Pilot 5000 (which he did let me touch, and convinced me that I really needed to get one of my own).

An especially warm thank you goes out to the free software community, both for producing the free Palm OS development tools that allowed me to get into Palm development in the first place and for providing source code of working Palm OS programs, which allowed me to learn the ropes of the Palm OS. In particular, I thank Mitch Blevins, author of DiddleBug and other fine free software, for fabulous source code to work from and general camaraderie between developers, and John Marshall, maintainer of the PRC-Tools, for his able assistance in getting the PRC-Tools up and running under GNU/Linux.

I also thank Garbage, Goodness, Guano Apes, and other bands whose names do not begin with “G” (like Jethro Tull, Depeche Mode, and They Might Be Giants), for their inspiring tunes, which were of great help during those really long chapters. In addition, my thanks go out to Nullsoft, makers of the Winamp MP3 player, whose fine program allowed me to queue up hundreds of songs by the aforementioned bands and blast them at obnoxious volumes.

Finally, I would like to offer my eternal thanks to Elisabeth (my wife), Constance Maytum, John Hedtke, Alan Zander, both of my cats, and all my friends who put up with alternating blank stares and manic technical babbling from me over the course of this massive project.

Contents at a Glance

Foreword ix
Preface xi
Acknowledgments xviii

Part I: Getting Started 1
Chapter 1: Understanding the Palm Computing Platform 3
Chapter 2: Understanding the Palm OS 15

Part: II: Creating Palm OS Applications 41
Chapter 3: Introducing the Development Environments 43
Chapter 4: Writing Your First Palm OS Application 67
Chapter 5: Debugging Your Program 97

Part III: Programming the Palm OS 131
Chapter 6: Creating and Understanding Resources 133
Chapter 7: Building Forms 165
Chapter 8: Building Menus 193
Chapter 9: Programming User Interface Elements 213
Chapter 10: Programming System Elements 265
Chapter 11: Programming Tables 309

Part IV: Storing Information on the Handheld 371
Chapter 12: Storing and Retrieving Data 373
Chapter 13: Manipulating Records 403

Part V: Communicating Outside the Handheld 469
Chapter 14: Beaming Data by Infrared 471
Chapter 15: Using the Serial Port 507
Chapter 16: Creating Web Clipping Applications 535

Part VI: Synchronizing Data with the Desktop. 571
Chapter 17: Introducing Conduit Mechanics 573
Chapter 18: Building Conduits 593

Part VII: Advanced Programming Topics 657
Chapter 19: Programming in Color 659
Chapter 20: Odds and Ends 673

Appendix A: Palm OS API Quick Reference 707
Appendix B: Finding Resources for Palm OS Development 813
Appendix C: Developing in Other Environments 821
Appendix D: What's on the CD-ROM? 829

Glossary 837
Index 861
End-User License Agreement 894
GNU General Public License 897
CD-ROM Installation Instructions 904

Contents

Foreword ix

Preface. xi

Acknowledgments xviii

Part I: Getting Started 1

Chapter 1: Understanding the Palm Computing Platform 3

The Palm OS Philosophy 3
Comparing Desktop and Handheld Application Design 4
Expectation of Performance 5
Limited Input Methods 5
Small Screen Size 6
Battery and Processor Power 6
Limited Memory 7
RAM as Permanent Storage. 7
Connecting to the Desktop 8
Comparing Hardware Versions 9
Looking to the Future. 13

Chapter 2: Understanding the Palm OS 15

Understanding a Palm OS Handheld's Power Usage 15
Running a Palm OS Application 16
Responding to Launch Codes. 17
Handling Events 17
Managing Memory 18
Dynamic RAM. 19
Storage RAM 21
Using Resources. 23
Designing the User Interface. 24
Forms 25
Alerts 26
Menus 27
Tables 27
Lists 28
Pop-up Triggers. 28
Buttons 29

Repeating Buttons 29
Selector Triggers 30
Push Buttons 30
Check Boxes 31
Labels 31
Form Bitmaps 31
Fields 32
Graffiti Shift Indicator 32
Scroll Bars 33
Gadgets 34
Communicating with Other Devices 35
Serial 35
TCP/IP 35
Wireless 36
IrDA 36
Beaming 36
Comparing Palm OS Versions 37
Changes in Version 2.0 37
Changes in Version 3.0 38
Changes in Version 3.1 38
Changes in Version 3.2 38
Changes in Version 3.3 39
Changes in Version 3.5 39

Part II: Creating Palm OS Applications 41

Chapter 3: Introducing the Development Environments 43

Using CodeWarrior for Palm OS 43
Familiarizing Yourself with the IDE 45
Changing Target Settings 52
Compiling and Linking in CodeWarrior 56
Using the GNU PRC-Tools 57
Compiling and Linking with the PRC-Tools 59
Automating Builds with Make 61

Chapter 4: Writing Your First Palm OS Application 67

Looking at the Hello World User Interface 67
Walking Through the Hello World Code 68
Including Header Files 69
Entering the Application 71
Starting the Application 73
Closing the Application 73
Handling Events 73
Setting Up Forms 76
Responding to Form Events 77

Handling Menu Events 81
Displaying Alerts and Using the Text Field 83
Using Memory in the Palm OS 85
Putting It All Together 90

Chapter 5: Debugging Your Program 97

Using the Palm OS Emulator 97
Controlling POSE 100
Running POSE for the First Time 102
Installing a ROM Image 103
Installing Applications 106
Saving and Restoring Configurations 106
Adjusting POSE Settings 107
Handling Gremlins 112
Emulating a HotSync Operation 115
Taking Screen Shots 117
Handling Errors in POSE 118
Debugging at the Source Level 118
Debugging with CodeWarrior 119
Debugging with GDB 122
Resetting a Palm OS Handheld 125
Using Developer Graffiti Shortcuts 125
Using the Palm OS Error Manager 127

Part III: Programming the Palm OS 131

Chapter 6: Creating and Understanding Resources 133

Following Palm OS User Interface Guidelines 133
Making Fast Applications 134
Highlighting Frequently Used Functions 135
Designing for Ease of Use 136
Maintaining Palm OS Style 137
Creating Resources with Constructor 142
Exploring the Project Window 143
Creating Catalog Resources 156
Creating Resources with PilRC 157
Creating Application Resources 158
Previewing the Interface in PilrcUI 163
Assigning Constants to Resources 163

Chapter 7: Building Forms 165

Building Forms with Constructor 165
Setting Common Object Properties 167
Setting Individual Object Properties 169

Building Forms with PilRC 180
Creating a Form Resource 181
Adding Objects to a Form 182

Chapter 8: Building Menus 193

Building Menus with Constructor 193
Sharing Menus between Menu Bars 196
Building Menus with Rez 197
Integrating Rez Menus with Your Project 200
Building Menus with PilRC 202
Introducing Librarian, a Sample Application 204
Displaying Multiple Records in List View 204
Displaying an Individual Book in Record View 205
Editing a Record in Edit View 206
Examining Librarian's Menus 208

Chapter 9: Programming User Interface Elements 213

Programming Alerts 213
Programming Forms 216
Switching to a New Form 216
Displaying a Complex Modal Dialog Box 217
Displaying a Simple Modal Dialog Box 218
Programming Objects on Forms 221
Handling Form Object Events 222
Retrieving an Object Pointer 225
Hiding and Showing Form Objects 226
Programming Check Boxes and Push Buttons 227
Handling Control Groups 228
Programming Selector Triggers 229
Programming Fields 233
Setting a Handle for a Text Field 233
Modifying a Text Field 234
Retrieving Text from a Field 236
UInt16 length = FldGetTextLength(field); Setting Field Focus 236
Setting Field Attributes 237
Programming Gadgets 238
Programming Lists and Pop-up Lists 243
Retrieving List Data 243
Manipulating Lists 244
Programming Dynamic Lists 245
Handling Pop-up Lists 248
Programming Menus 249
Using MenuEraseStatus 250
Removing Menu Items 250
Drawing Graphics and Text 252
Understanding Windows 252

Drawing Lines . . . 256
Drawing Rectangles . . . 256
Drawing Text . . . 260
Drawing Bitmaps . . . 261

Chapter 10: Programming System Elements . . . 265

Checking for Supported Features . . . 265
Determining Operating System Version . . . 266
Checking Individual Features . . . 268
Manipulating Text . . . 270
Using Font Functions . . . 270
Using String Functions . . . 274
Using Character Macros . . . 276
Handling Pen Events . . . 279
Handling Key Events . . . 281
Setting Alarms . . . 284
Setting an Alarm . . . 285
Responding to Alarms . . . 286
Responding to Other Launch Codes . . . 289
Playing Sounds . . . 290
Looking Up Phone Numbers . . . 292
Launching Applications . . . 293
Calling the System Application Launcher . . . 294
Launching Applications Directly . . . 294
Sending Launch Codes Globally . . . 297
Creating Your Own Launch Codes . . . 297
Generating Random Numbers . . . 298
Managing Power . . . 299
Reacting to Low Battery Conditions . . . 300
Identifying the Device . . . 301
Manipulating Time Values . . . 302
Retrieving and Setting Time Values . . . 303
Converting Time Values . . . 303
Altering Time Values . . . 305
Using the Clipboard . . . 305

Chapter 11: Programming Tables . . . 309

Creating a Simple Table . . . 310
Understanding How Tables Work . . . 311
Initializing a Table . . . 314
Handling Table Events . . . 328
Hiding Rows and Columns . . . 328
Creating More Complex Tables . . . 331
Connecting a Table to Data . . . 332
Scrolling Tables . . . 350
Handling Table Text Fields . . . 365

Part IV: Storing Information on the Handheld 371

Chapter 12: Storing and Retrieving Data 373

Understanding the Data Manager 373
Resource Databases 378
Working with Databases 379
Creating Databases 379
Opening Databases 381
Closing Databases 382
Finding Databases 383
Deleting Databases 385
Retrieving and Modifying Database Information 386
Creating an Application Info Block 391
Storing Application Preferences 394
Reading and Setting System Preferences 397
Using Feature Memory 399

Chapter 13: Manipulating Records 403

Working with Records 403
Looking at Records in the Librarian Sample Application 404
Comparing Records 409
Finding Records 416
Creating Records 418
Deleting Records 422
Reading Records 424
Modifying Records 424
Sorting Records 435
Retrieving and Modifying Record Information 436
Categorizing Records 438
Implementing Private Records 448
Resizing Records 449
Working with Resources 450
Finding Resources 452
Creating Resources 453
Deleting Resources 454
Reading Resources 455
Retrieving and Modifying Resource Information 457
Resizing Resources 458
Implementing the Global Find Facility 458
Handling sysAppLaunchCmdSaveData 459
Handling sysAppLaunchCmdFind 460
Handling sysAppLaunchCmdGoto 464

Part V: Communicating Outside the Handheld 469

Chapter 14: Beaming Data by Infrared 471

Using the Exchange Manager 471
Registering a Data Type 475
Sending Data. 477
Customizing the Beam Acceptance Dialog Box 488
Receiving Data. 492
Displaying Beamed Records 499
Debugging Beaming. 499
Beaming Applications and Databases 500
Understanding the IR Library 503

Chapter 15: Using the Serial Port 507

Understanding Palm OS Serial Communications 507
Using the Serial Manager 510
Using the New Serial Manager. 511
Using the Old Serial Manager 529

Chapter 16: Creating Web Clipping Applications 535

Understanding Web Clipping. 535
Understanding Web Clipping Security 537
Designing PQAs and Web Clippings 537
Building Palm Query Applications 539
Organizing HTML Files 540
Defining Header Tags 541
Formatting Text 542
Linking to Other Pages and Applications 544
Constructing Query Forms 549
Adding Images. 554
Using the Query Application Builder 556
Looking at a Sample PQA 558
Building Web Clippings 561
Defining Header Tags 562
Creating Clipping Pages for Desktop Browsers 562
Linking Outside the Web Clipping 563
Adding Images. 563
Looking at a Sample Web Clipping 563
Testing Web Clipping Applications 567

Part VI: Synchronizing Data with the Desktop 571

Chapter 17: Introducing Conduit Mechanics. 573

Understanding Conduits 574
Stepping Through the HotSync Process 576
Designing Conduits 578
Choosing a Development Path 579
Installing Conduits. 580
Installing Conduits Manually 581
Creating Automatic Conduit Installations 585
Logging Actions in the HotSync Log. 588

Chapter 18: Building Conduits 593

Using the Conduit Wizard 593
Selecting a Conduit Type. 595
Choosing a Handheld Application 595
Selecting a Data Transfer Type 597
Selecting Conduit Features 598
Confirming Class and File Names 599
Implementing Conduit Entry Points. 601
Implementing GetConduitInfo 602
Implementing GetConduitName. 605
Implementing GetConduitVersion. 606
Implementing OpenConduit 606
Implementing Configuration Entry Points 610
Using the Palm MFC Base Classes 619
Following MFC Conduit Flow of Control 621
Implementing a Monitor Class. 622
Implementing a Table Class 623
Implementing a Schema Class. 626
Implementing a Record Class 627
Implementing a Link Converter Class 631
Using the Generic Conduit Base Classes 635
Following Generic Conduit Flow of Control 636
Describing the Desktop Record Format 637
Implementing Storage and Retrieval 639
Converting Data to and from CPalmRecord 643
Syncing the Application Info Block 645
Using the Sync Manager API 646
Registering and Unregistering a Conduit. 646
Opening and Closing Handheld Databases 646
Iterating Over Database Records 650
Reading and Writing Records 653
Deleting Records 654
Maintaining a Connection 655

Part VII: Advanced Programming Topics 657

Chapter 19: Programming in Color 659

Determining and Setting Color Depth 660
Retrieving Color Depth 662
Setting Color Depth 663
Using Color Tables 664
Translating RGB to Index Values 667
Using Color Bitmaps 667
Coloring the User Interface 669

Chapter 20: Odds and Ends 673

Creating Large Applications 673
Breaking the 32KB Barrier 674
Segmenting Applications 676
Adding Custom Fonts to Applications 684
Creating a Custom Font 685
Creating User Interface Dynamically 688
Localizing Applications 692
Using the Text and International Managers 692
Using the File Streaming API 699
Opening File Streams 700
Closing File Streams 702
Retrieving File Stream Errors 703
Deleting File Streams 703
Setting Position in a File Stream 704
Reading and Writing File Stream Data 704

Appendix A: Palm OS API Quick Reference 707

Appendix B: Finding Resources for Palm OS Development 813

Appendix C: Developing in Other Environments 821

Appendix D: What's on the CD-ROM? 829

Glossary. 837
Index 861
End-User License Agreement 894
GNU General Public License. 897
CD-ROM Installation Instructions. 904

CHAPTER 1

Understanding the Palm Computing Platform

In This Chapter

How and why the Palm Computing platform works

Designing applications for handheld devices

Connecting to the desktop

Comparing hardware versions

The future of Palm OS

Since the release of the Pilot 1000 in 1996, devices running Palm OS have dominated the handheld computing market. Right from the start, Palm Computing was able to combine just the right mix of features to make a Personal Digital Assistant (PDA) that is easy to integrate into almost any user's lifestyle. Programming an application that takes advantage of the strengths of the Palm Computing platform requires an understanding of not only how the platform works, but also why it was designed the way it was.

This chapter explains some of the thinking that has made the Palm Computing platform so successful. It also provides an overview of the different versions of Palm OS available and the hardware platforms on which they run.

The Palm OS Philosophy

Devices running the Palm OS are not intended to be portable versions of desktop computers. Instead, the handheld is a satellite device, designed as an extension to a desktop system. The handheld provides a window to desktop data, allowing that data to be viewed anywhere. Though it is indeed possible to perform many complex tasks with Palm OS handhelds, their form and function are optimized for viewing data and entering small amounts of data.

In order to meet the goal of conveniently presenting a user's data while away from the desktop, the handheld device must adhere to certain criteria:

- **Small size.** It needs to be small enough to be carried anywhere. Most of the devices currently available for the Palm Computing platform easily fit in a shirt pocket. The Palm V, smallest member of the Palm family, measures 4.5 × 3.1 × 0.4 inches, weighing a measly 4 ounces. Even the largest, Qualcomm's pdQ, measures only 1.4 × 6.2 × 2.6 inches, with a weight of 8.2 ounces, and it includes a fully functional cell phone in that small package.
- **Ergonomic interface.** Using the device must be simple and quick enough to not interrupt whatever the user is currently doing. Handheld users need to comfortably and rapidly operate the device during meetings, in airports, at business lunches, and in other situations where there is no convenient place to set the device down. Useful information should be available instantly and with a minimum of user interaction. The four main applications that ship with Palm OS handhelds (Date Book, Address Book, To Do List, and Memo Pad) can display useful information without any stylus input from the user; the user can scroll through the applications' data by using the hardware buttons.
- **Desktop integration.** The handheld must synchronize easily and reliably with the desktop computer. Synchronizing with the desktop not only backs up important data, but it also allows the user to input large amounts of data on a desktop machine with a mouse and keyboard, which is much better suited to mass data entry than the limited interface of the handheld. Palm OS handhelds include a cradle to sync the handheld to the desktop with a single button press, and Palm Computing's HotSync technology quickly transfers data between the handheld and desktop.

Palm Computing hit upon a perfect combination of these factors with its first device, and it has resisted the temptation to cram marginally useful features into new Palm devices. Although they have fewer features than many other handhelds, such as Windows CE and the older Newton devices, Palm OS handhelds are more focused on providing features that will be genuinely useful. Intelligent selection of features has made these devices into handy tools instead of merely expensive toys.

Comparing Desktop and Handheld Application Design

There are significant differences between a desktop computer and a handheld device — enough differences that designing a handheld application must be approached differently from designing a desktop application. Several elements must be kept in mind when designing a Palm OS application:

- Expectation of performance
- Limited input methods

- ✦ Small screen size
- ✦ Battery and processor power
- ✦ Limited memory
- ✦ RAM as permanent data storage

Expectation of Performance

Desktop application users usually won't mind waiting a few seconds for a program to load because they plan to be using the application for an extended period of time. A user seated at a desk probably isn't going anywhere anytime soon.

Compare this with a handheld user on the go. A person using a Palm OS handheld will need to look up a piece of data (such as a phone number) quickly, or spend a few seconds jotting down a note, while in the middle of performing some other task. Someone who is talking to clients on the phone or trying to catch a bus doesn't have the time to watch a spinning wait cursor while an application loads.

Speed and efficiency are key to a successful Palm OS application. Writing fast code is only a small part of the equation; the user interface must be simple and quick to use. The application should allow for rapid navigation, selection, and execution of commands. Functions that the user will use most often should require less interaction than those that will be used infrequently.

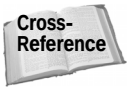
Limited Input Methods

A desktop system is ideal for entering large quantities of data. A keyboard and a fast processor allow desktop users to easily input lots of text into the computer in a short period of time.

A Palm OS handheld does not have a keyboard. Though third-party add-on keyboards exist, such as the Newton, GoType!, and Palm Portable keyboards, most users of a standard Palm OS handheld must enter text with a stylus and either Graffiti or an on-screen keyboard. Graffiti, a software system that converts a special type of shorthand into text, is faster and more accurate than previous attempts at handwriting recognition, notably those used by the Apple Newton. Instead of using the limited processor power and memory available on a handheld device to make sense of your own handwriting, Graffiti relies on a much more powerful system to perform its magic: the human brain. It is much simpler for a person to learn to write Graffiti's simple set of glyphs than it is for a piece of software to interpret the idiosyncrasies of most people's handwriting. (A friend of mine used to own a Newton, and after spending months tuning it to recognize his writing, he wouldn't let anyone else near the device for fear that they would "untrain" the recognition software.) Although Graffiti is faster than many forms of handwriting recognition, at a top speed of around thirty words per minute, it is still too slow for entering anything longer than a short note.

HotSync technology provides an easy way to get large amounts of data from the desktop system to the handheld. The Palm Computing platform is designed around the idea that users will perform mass data entry on the desktop machine, which is optimized for that kind of work, and then "sync" that data to the handheld. This kind of symbiosis between desktop computer and handheld plays to the strengths of both devices.

However, don't let this discourage you from writing applications that use a Palm OS handheld as a data collection tool. With intelligent interface design, you can perform data entry quickly and efficiently on such a device.



For more details about designing efficient data entry, see Chapter 2, "Understanding the Palm OS."

Small Screen Size

Current desktop machines have large monitors, generally running at a minimum resolution of 640 × 480 pixels. With this kind of screen real estate to play with, displaying large amounts of information and a complex user interface in the same space is easy.

By contrast, Palm OS handhelds have a screen 6 centimeters on a side, with a resolution of 160 × 160 pixels. This screen size is necessary to keep the device within the shirt-pocket size range that has contributed to the popularity of such devices.

Designing applications to use such a small screen is a challenge. Displaying the right information is more important than fitting as much information on the screen as possible. You must strike a balance between showing enough information and keeping the interface uncluttered and simple to use.

Requiring users to scroll through many screens of data to find the information they want will make your application frustrating to use. Find logical groupings of data and offer the user a way to filter different views of that data. The To Do List application is a good example of data filtering; its preferences allow the user to quickly choose what subset of the list should be displayed. Implementing the standard Palm OS user-defined categories can also help users zero in on exactly the data they want to view.

Battery and Processor Power

Unlike desktop machines, which are plugged into wall outlets and sport powerful, fast processors, Palm OS handhelds must rely on batteries for power, which limits them to slower processors. The small processor on such a device is not well suited to intense computation.

If your application has both handheld and desktop components, consider doing all your intensive number crunching in the desktop portion. A great example of

relegating processor-intensive tasks to the desktop machine is Doc, the de facto standard for large text documents on the Palm OS. Several converter applications exist for the desktop machine, which perform the computationally expensive conversion and compression of a large text document to Doc format. The newly formatted document can then be transferred to the handheld during the next HotSync session. All the Doc viewer application on the handheld need concern itself with is displaying the document; all the hard stuff has been handled by the faster desktop computer.

Limited Memory

As memory prices continue to drop, desktop applications can afford to be less choosy about how they deal with memory. When your application has 64MB or more to play with, it can load huge data structures into RAM and leave them there the entire time the program is running.

Palm OS handhelds have very limited memory space for running applications. On Palm OS 3 and later, there is less than 36KB of memory available for dynamic allocation, application global variables, and static variables. Earlier versions of Palm OS have considerably less room, so writing applications that are compatible with older Palm OS handhelds can be somewhat challenging. Keep this in mind when writing your application; things like deeply recursive routines, large numbers of global variables, and huge dynamically allocated data structures are not Palm OS–friendly.

RAM as Permanent Storage

Hard drives provide desktop computers with abundant permanent storage for vast amounts of data. Palm OS handhelds are considerably more limited in storage space because they must store applications and data in RAM.

Available memory on a Palm OS handheld ranges between 128KB on the Pilot 1000 and 8MB on the Palm IIIxe or Visor Deluxe. This kind of limited storage dictates that handheld applications must be as small as possible. Avoid adding features to your application that will be used infrequently; if a feature will be used by fewer than 20 percent of the users, leave it out.

For example, features that globally modify an application's data, but will see only infrequent use, are prime candidates for inclusion in a companion program on the desktop. A command that removes duplicate entries in a database would be perfect for the desktop; it's not likely to be used very often on the handheld, and removing it from the handheld application makes the program smaller.

Your application should also pack its data tightly before writing it to memory. Not only will this reduce the amount of RAM required to store your application's data, but it will also decrease the amount of time taken by HotSync when synchronizing that data with the desktop computer.

Connecting to the Desktop

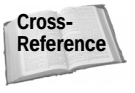
Sharing data with the desktop is a key ingredient in the popularity of Palm OS handhelds. The connection between desktop and handheld allows each device to borrow the strengths of the other. A desktop computer is great for large-scale data entry and crunching numbers, but you can't carry one in your pocket when visiting clients. A handheld device is perfect for taking quick notes and reminding you of appointments, but it's terrible for analyzing financial reports or writing a book. Together, the devices become greater than the sum of their parts.

The software component that forms the vital link between the Palm OS device and the desktop computer is called a *conduit*. HotSync calls code in a conduit, which resides on the desktop computer, during synchronization with your handheld application, and this code controls exactly what data HotSync transfers between the two devices. There are several different scenarios in which a conduit plays a vital role; here are just a few examples:

- Two applications, one on the handheld and one on the desktop, use the conduit to keep records in their databases in sync with each other. This is how the conduit for the Date Book and the three other main Palm OS applications works. In this scenario, the conduit is responsible for looking at the records in both databases and determining what records are different between them, as well as which direction those data must be transferred.
- The conduit keeps data in a handheld application synchronized with data in a centralized corporate database, either stored on the machine running HotSync, or another machine on a corporate network. In this case, the conduit might also sift the data and transfer only a customized subset to the handheld based on user preferences. Customization like this keeps the size of the data manageable and reduces the time required for HotSync to run.
- When syncing, the conduit compares content on the handheld with the contents of a Web page or Usenet newsgroup. If the information on the Web or newsgroup is newer than what the handheld application has stored, the conduit downloads the new data, processes it into a form the handheld application can read, and transfers it to the handheld. The conduit may also instruct the handheld application to cull out-of-date pages or articles. Since Internet connections are prone to delays, this sort of conduit should probably only look at information previously cached by a desktop application. A HotSync operation should be as short as possible because having the serial port open drains a Palm OS handheld's batteries rapidly.

If your application does not require the level of detailed synchronization logic that a conduit can provide, you may be able to use the default *backup conduit*. Instead of comparing the handheld application's database record by record with data on the

desktop, the backup conduit simply makes a copy of the entire database and transfers it to the desktop computer. This works perfectly well for small application databases but can slow down the HotSync process if your application stores a lot of data.



Chapter 17, "Introducing Conduit Mechanics," provides an introduction to developing Palm OS conduits. Further details on writing conduits follow in Chapter 18, "Building Conduits."

Comparing Hardware Versions

Palm OS handhelds have evolved slowly, adding just a few new features at a time. This incremental change is a boon to application developers, because it means that new versions of hardware and operating system software require only small changes, if any, to existing applications instead of requiring that they be rewritten from the ground up.

Even though Palm Computing has wisely refrained from making wild, earth-shattering changes to the platform, there are some significant differences between versions of the hardware that you should take into account when designing your application. Table 1-1 highlights the features of different Palm OS devices.

Table 1-1
Palm OS Handheld Features

Pilot family		
Feature	***Pilot 1000***	***Pilot 5000***
Palm OS version	1.0	1.0
Processor	Motorola MC68328 "DragonBall"	Motorola MC68328 "DragonBall"
Memory	128KB	512KB
Flash ROM	No	No
Backlight	No	No
TCP/IP	No	No
Infrared	No	No
Enhanced LCD screen	No	No
Battery	2 AAA alkaline batteries	2 AAA alkaline batteries
Hardware expansion	Replaceable memory card	Replaceable memory card

Continued

Table 1-1 *(continued)*

PalmPilot family

Feature	***PalmPilot Personal***	***PalmPilot Professional***
Palm OS version	2.0	2.0
Processor	Motorola MC68328 “DragonBall”	Motorola MC68328 “DragonBall”
Memory	512KB	1MB
Flash ROM	No	No
Backlight	No	No
TCP/IP	No	Yes
Infrared	No	No
Enhanced LCD screen	No	No
Battery	2 AAA alkaline batteries	2 AAA alkaline batteries
Hardware expansion	Replaceable memory card	Replaceable memory card

Palm III family

Feature	***Palm III***	***Palm IIIe***	***Palm IIIx***	***Palm IIIxe***	***Palm IIIc***
Palm OS version	3.0	3.1	3.1	3.5	3.5
Processor	Motorola MC68328 “DragonBall”	Motorola MC68EZ328 “DragonBall EZ”	Motorola MC68EZ328 “DragonBall EZ”	Motorola MC68EZ328 “DragonBall EZ”	Motorola MC68EZ328 “DragonBall EZ”
Memory	2MB	2MB	4MB	8MB	8MB
Flash ROM	Yes	No	Yes	Yes	No
Backlight	Yes	Yes	Yes	Yes	Yes; color screen acts as its own backlight
TCP/IP	Yes	Yes	Yes	Yes	Yes
Infrared	Yes	Yes	Yes	Yes	Yes
Enhanced	No	Yes	Yes	Yes	Color Active Matrix TFT LCD screen

Feature	*Palm III*	*Palm IIIe*	*Palm IIIx*	*Palm IIIxe*	*Palm IIIc*
Battery	2 AAA alkaline batteries	2 AAA alkaline batteries	2 AAA alkaline batteries	2 AAA alkaline batteries	Recharge-able lithium ion battery
Hardware expansion	Replaceable memory card	Not possible	Open connector slot	Open connector slot	Open connector slot

Palm V family and Palm VII

Feature	*Palm V*	*Palm Vx*	*Palm VII*	*Palm VIIx*
Palm OS version	3.1	3.3	3.2	3.5
Processor	Motorola MC68EZ328 "DragonBall EZ"	Motorola MC68EZ328 "DragonBall EZ"	Motorola MC68EZ328 "DragonBall EZ"	Motorola MC68EZ328 "DragonBall EZ"
Memory	2MB	8MB	2MB	8MB
Flash ROM	Yes	Yes	Yes	Yes
Backlight	Yes	Yes	Yes	Yes
TCP/IP	Yes	Yes	Yes, plus wire-less connectivity	Yes, plus less connectivity
Infrared	Yes	Yes	Yes	Yes
Enhanced LCD screen	Yes	Yes	Yes	Yes
Battery	Rechargeable lithium ion battery	Rechargeable lithium ion battery	2 AAA alkaline batteries	2 AAA alkaline batteries
Hardware expansion	Not possible	Not possible	Not possible	Not possible

Palm m100

Feature	*Palm m100*
Palm OS version	3.5.1
Processor	Motorola MC68EZ328 "DragonBall EZ"
Memory	2MB
Flash ROM	No
Backlight	Yes
TCP/IP	Yes

Continued

Table 1-1 *(continued)*

Feature	***Palm m100***
Infrared	Yes
Enhanced LCD screen	Yes; smaller (0.29 dot pitch)
Battery	2 AAA alkaline batteries
Hardware expansion	Not possible

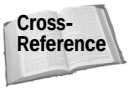
Handspring Visor and TRGPro

Feature	***Visor***	***Visor Deluxe***	***TRGPro***
Palm OS version	3.3	3.3	3.3
Processor	Motorola MC68EZ328 "DragonBall EZ"	Motorola MC68EZ328 "DragonBall EZ"	Motorola MC68EZ328 "DragonBall EZ"
Memory	2MB	8MB	8MB
Flash ROM	No	No	Yes
Backlight	Yes	Yes	Yes
TCP/IP	Yes	Yes	Yes
Infrared	Yes	Yes	Yes
Enhanced LCD screen	Yes	Yes	Yes
Battery	2 AAA alkaline batteries	2 AAA alkaline batteries	2 AAA alkaline batteries
Hardware expansion	Springboard module slot	Springboard module slot	CompactFlash slot

IBM's WorkPad series are essentially clones of PalmPilot Professional, Palm III, and Palm V, and they have the same features. Both the Symbol SPT line and the Qualcomm pdQ are based on the Palm III and share all of its features, but include extra hardware for specialized purposes; the SPT series includes barcode scanning equipment, and on the SPT 1740, a radio for connection to a Spectrum 24 short-range wireless network. The pdQ has an integral cell phone.

Fortunately for handheld software developers, these hardware differences are either insignificant from a programming perspective, or the Palm OS application programming interfaces (APIs) handle them gracefully. Your application can query

the Palm OS to determine what features are available before attempting to use a piece of hardware that doesn't exist on a particular handheld.



For more information on determining what features are available to your application, see Chapter 10, "Programming System Elements."

Physically, Palm OS handhelds have changed very little. Palm Computing has applied sleeker industrial design to more recent models, and the cases of the Symbol SPT series and the Qualcomm pdQ are shaped differently to accommodate a barcode reader and a cell phone, respectively. The Palm VII also has a slightly larger case, earning it the affectionate nickname "FrankenPalm" from some users, because of the enlarged "forehead" required to hold the wireless radio components. Likewise, the Palm IIIc is a bit larger than other Palm III series handhelds because of the extra space required for a color screen and rechargeable lithium ion batteries. All current Palm OS handhelds have the same familiar layout of hardware buttons and silk-screened Graffiti input area.

Looking to the Future

The official Palm OS documentation stresses the importance of developers' not making assumptions about the hardware that underlies their applications. This is important because the hardware may change, and if your code ignores the Palm OS APIs and directly accesses the hardware, your application is very likely to break on future devices.

Palm Computing has separate groups working on hardware and software. The group developing the Palm OS plans to add features to the operating system that the hardware group won't necessarily incorporate in handhelds made by Palm Computing. Instead, these APIs will be used by third-party partners making their own hardware that runs the Palm OS. Already, manufacturers such as Handspring and TRG have released their own handhelds running the Palm OS, and other companies, such as Sony, have announced their intentions to release new hardware that uses the Palm OS. There is an equally good chance that some of these devices may be very different from the current crop of Palm handhelds, incorporating new hardware features such as larger screens.

Use the Palm OS APIs instead of making direct calls to hardware. The Palm OS APIs are very complete, and if you stick with using the provided functions, your application will continue to run smoothly on new devices.

Summary

This chapter has explained the philosophy behind the Palm Computing platform and introduced you to the unique mindset required to write effective handheld applications. You should now know the following:

- ✦ The Palm Computing platform's success depends on a small device with ergonomic interface and seamless desktop integration.
- ✦ Handheld application development is very different from desktop application development and requires that you work within a number of constraints.
- ✦ Connecting your handheld device to the desktop computer takes place through a conduit.
- ✦ Though a number of Palm OS handhelds are on the market, they are very similar in form and function.
- ✦ You should call Palm OS functions instead of directly accessing the hardware in your application to ensure that it will continue to work on future hardware.

✦ ✦ ✦

CHAPTER 2

Understanding the Palm OS

In This Chapter

Understanding Palm OS power usage

Starting, running, and stopping Palm OS applications

Managing memory

Using resources and user interface elements

Communicating with other devices

Comparing Palm OS versions

The previous chapter introduced you to the philosophy behind the Palm Computing platform and the mindset required to write applications for it. Even with the limitations imposed upon a Palm application by the hardware and the very mobile nature of handheld usage, the Palm OS provides a wealth of features for the developer. The Palm OS handles everything from user interaction to database management to serial communications. This chapter provides an overview of the structure of the Palm OS and how it affects application design.

Understanding a Palm OS Handheld's Power Usage

Because of its small size, a Palm OS handheld must deal with power in a much different way than a desktop computer does. The Palm OS is designed to minimize power usage, making even ecologically conscious modern power-saving desktop systems look like energy hogs by comparison. Desktop machines with power-saving features do so to save money on the electric bill, but Palm OS devices have to be energy efficient for a different reason: battery life.

Most Palm devices run on a pair of AAA alkaline batteries, and even devices in the Palm V family, with their rechargeable lithium ion batteries, have little power to spare. The Palm OS manages to stretch the small amount of power available to it for weeks of normal operation, which is really amazing when you consider that the device is never really turned off.

A Palm OS handheld constantly supplies power to important subsystems. The on/off button on the device only toggles the device between a low-power mode and a more active mode. Power must not be completely turned off on the Palm, because the memory, real-time clock, and interrupt generation circuitry require some small amount of power constantly for proper operation. This is particularly important in the case of memory because the device stores applications and permanent data in RAM, which loses data if the handheld loses power.

The Palm OS supports three modes of operation:

- **Sleep mode.** This is the mode a Palm handheld user identifies with the device being turned "off." Everything on the device that does not require power is shut down, including the display, the digitizer, and the main system clock. Only essential systems such as interrupt-generation circuitry and the real-time clock are active, along with a trickle of power to keep the RAM from losing its data. When the device is in sleep mode, only certain interrupts, such as input from the serial port or a hardware button press, will "wake up" the device. After a user-customizable period of time (from one to three minutes), the device will drop automatically into sleep mode.
- **Doze mode.** Most of the time the device appears to be "on," it is in doze mode. The main clock, digitizer, and LCD screen are turned on, and the processor clock is running but not executing instructions. When there is no user input to process, the system enters doze mode. Any hardware interrupt (such as text input) that the processor receives will bring the processor out of doze, which is much faster than coming out of sleep mode, because the device does not need to power up any of its peripheral hardware.
- **Running mode.** In this mode, the processor is actively executing instructions. User input in doze mode will put the device into running mode, as will an interrupt while in doze mode or sleep mode, such as the system alarm going off or the user pressing a hardware button. The device remains in running mode long enough to process the user input, usually less than a second, and then it immediately drops to doze mode. Most applications will cause the system to enter running mode only 5 percent of the time.

Running a Palm OS Application

The Palm OS has a pre-emptive multitasking kernel. However, the User Interface Application Shell (UIAS), the part of the OS responsible for managing applications that display a user interface, runs only one application at a time. Normally, the only task running is the UIAS, which calls application code as a subroutine. The UIAS doesn't gain control again until the currently running application quits, at which point the UIAS immediately calls the next application as another subroutine.

Applications may not be multithreaded, because they must run within the single thread of the UIAS.

Certain calls in an application will cause the OS to launch a new task. For instance, the HotSync application starts another task to handle serial communication with the desktop computer. The serial communication task has a lower priority than the main user interface task.

This creation of a new task allows for more optimized communication, both with the desktop and the user. If the user taps on the screen to cancel the sync, the higher-priority user interface task processes the tap immediately, allowing for quick response to user input, even when the serial task is using most of the processor's time to talk with the desktop. However, because there usually isn't any user interaction during a HotSync session, the serial task gets all the processor time it needs for rapid communication.

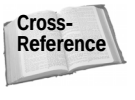
Only the system software can launch a new task. Application code does not have direct access to the Palm OS multitasking APIs.

Responding to Launch Codes

When the system launches an application, it calls a function named **PilotMain** (similar to the **main** function in a C program) and passes it a launch code. The launch code may tell the application to start and display its user interface, in which case it will start up its event loop and process the event queue. This kind of startup is called a normal launch.

Alternatively, the launch code may tell the application to perform some small task, without displaying its user interface, and then exit. The Palm OS global find function works this way, sending a launch code to each application on the device requesting that it search its own databases for a particular string. Launch codes also exist for many other purposes, such as opening the application to a specific record or notifying the application that a HotSync has just been completed.

When an application receives any launch code other than a normal launch, control does not pass to the event loop, but rather to another function in the application that does its job outside the event loop.



Launch codes are covered in more detail in Chapter 4, "Writing Your First Palm OS Application."

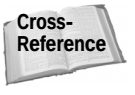
Handling Events

A Palm OS application is event-driven, receiving events from the OS and either handling them or passing them back to be handled by the OS itself. An event structure describes the type of event that has taken place (for example, a stylus tap on an on-screen button), as well as information related to that event, such as the screen coordinates of a stylus tap. During a normal launch, execution passes to the application's *event loop*, which retrieves events from the event queue and dispatches them according to the type of event.

The event loop passes most events back to the OS, because the system already has facilities for dealing with common tasks such as displaying menus or determining what button on the screen was tapped. Those events that are not handled by the OS go to the application's own event handler, which either handles the events if they are interesting to the application, or passes them back to the event loop.

A typical Palm OS application will remain in the event loop until it receives an event telling it to close the application, at which point the event loop will pass control to another function that performs cleanup operations and prepares the program to be shut down.

The standard event loop is an important ingredient in power management in a Palm OS application. A normal event loop calls OS functions to process the event queue, and these functions know enough about managing power to put the device into doze mode if no events currently need processing. Using a standard event loop also ensures that if the application is left on for a few minutes, the operating system's auto-off feature will put the device into sleep mode.



The event loop is covered in more detail in Chapter 4, "Writing Your First Palm OS Application."

Managing Memory

Because the Palm OS was designed to run on inexpensive, low-power handheld devices, it is very good at dealing with tight memory conditions. The Palm OS does not handle all the burden of dealing with such a limited memory space, though. Memory constraints on a Palm OS device require that you pay careful attention to how you use memory in your application. Therefore, understanding the memory architecture of the Palm OS is very important to writing a successful application.

Both the ROM and RAM of a Palm OS device reside on a memory module called a *card*. In the first Palm devices, this was an actual physical card, which a user could easily replace to upgrade the amount of memory available, exchange the OS and applications in the ROM for newer versions, or both. However, a "card" is only a logical abstraction used by the Palm OS to describe a memory area used to contain ROM and RAM; a device may have any number of logical cards or no cards at all. As of Palm OS 3.5, there is only one card available (card 0), but future Palm OS handhelds may actually have more than one memory card.

The Palm OS is built around a 32-bit memory architecture, with data types 8, 16, and 32 bits long. Memory addresses are 32 bits long, giving the OS a total of 4GB of address space in which to store data and code. The operating system reserves 256MB of address space for each card. Future versions of the Palm OS have a lot of room to expand, because current devices use only a fraction of the available address space.

The memory architecture described here is representative only of the implementation in Palm OS version 3.5 and earlier, and is subject to change in future versions. Relying on these implementation-specific details may cause your application to crash on future versions of the OS. Always use the Palm OS memory APIs to manipulate memory.

RAM in the Palm OS is divided into two separate areas: *dynamic RAM* and *storage RAM*. Figure 2-1 graphically depicts these areas of memory and what they contain.

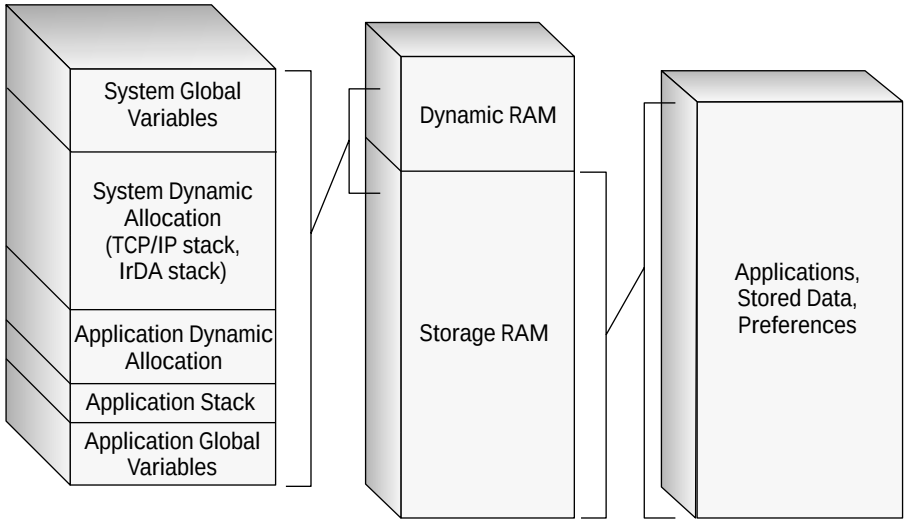


Figure 2-1: RAM in the Palm OS is divided into two areas: dynamic RAM and storage RAM.

Dynamic RAM is used for many of the same purposes as RAM on a desktop computer; it provides a space for temporary storage of global variables and other data that does not require persistence between executions of an application. Storage RAM is used in much the same way the file system on a desktop machine's hard drive is; it provides permanent storage for applications and data. Both dynamic and storage RAM are further detailed in the following section.

Dynamic RAM

The entire dynamic area of the device's RAM is used to implement the *dynamic heap*. A *heap* is a contiguous area of memory that manages and contains smaller units of memory. These smaller units are called *chunks*. A chunk is a contiguous area of memory between 1 byte and slightly less than 64KB in size. All data in the Palm OS environment are stored in chunks.

Current implementations of the Palm OS restrict chunks to less than 64KB in size, but this restriction may not exist in future versions of the OS. Again, always use the Palm OS memory APIs to manipulate memory.

The dynamic heap provides memory for several purposes:

- ✦ Application and system global variables
- ✦ Dynamic allocations by the system, such as the TCP/IP and IrDA stacks
- ✦ Stack space for the running application
- ✦ Temporary memory allocations
- ✦ Dynamic allocations by applications

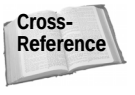
Table 2-1 shows how much space is allocated to the dynamic heap in different versions of the Palm OS, and it provides a breakdown of what that memory is used for. Notice that even in later versions, the dynamic heap is still a very small amount of memory, most of which is used by the operating system itself. Very little memory is left for application use.

Table 2-1
The Dynamic Heap in Various Versions of Palm OS

Memory Usage	*OS 3.x (more than 1MB total RAM; TCP/IP and IrDA)*	*OS 2.0 (1MB total RAM; TCP/IP only)*	*OS 2.0/1.0 (512MB total RAM; no TCP/IP or IrDA*
Total dynamic memory	96KB	64KB	32KB
System globals (UI globals, screen buffer, database references, and so forth.)	about 2.5KB	about 2.5KB	about 2.5KB
TCP/IP stack	32KB	32KB	0KB
System dynamic allocation (IrDA, “Find” window, temporary allocations)	variable amount	about 15KB	about 15KB
Application stack (call stack and local variable space)	4KB (default)	2.5KB	2.5KB
Remaining space (dynamic allocations, application global variables, static variables)	36KB	12KB	12KB

All of the RAM in the dynamic heap is dedicated to dynamic use. Even if some areas of the dynamic heap are currently not in use (for instance, no TCP/IP communication is currently taking place), that memory is still available only for the dynamic allocations outlined in Table 2-1.

Applications allocate, manipulate, and free allocated memory in the dynamic heap using the Palm OS memory manger. The memory manager functions allow safe use of the dynamic memory on the device, regardless of how the running version of the OS structures that memory internally.



See Chapter 4, "Writing Your First Palm OS Application," for more details about the memory manager APIs.

Storage RAM

Any memory on the device that is not dedicated to the dynamic heap is divided into a number of *storage heaps*. The size and number of storage heaps are dependent on the version of the OS and the total amount of RAM available on the device. In versions 1.0 and 2.0 of the Palm OS, storage RAM is divided into several 64KB storage heaps. Version 3.*x* treats all the storage RAM available as one big storage heap.

Storage Heaps and Memory Fragmentation

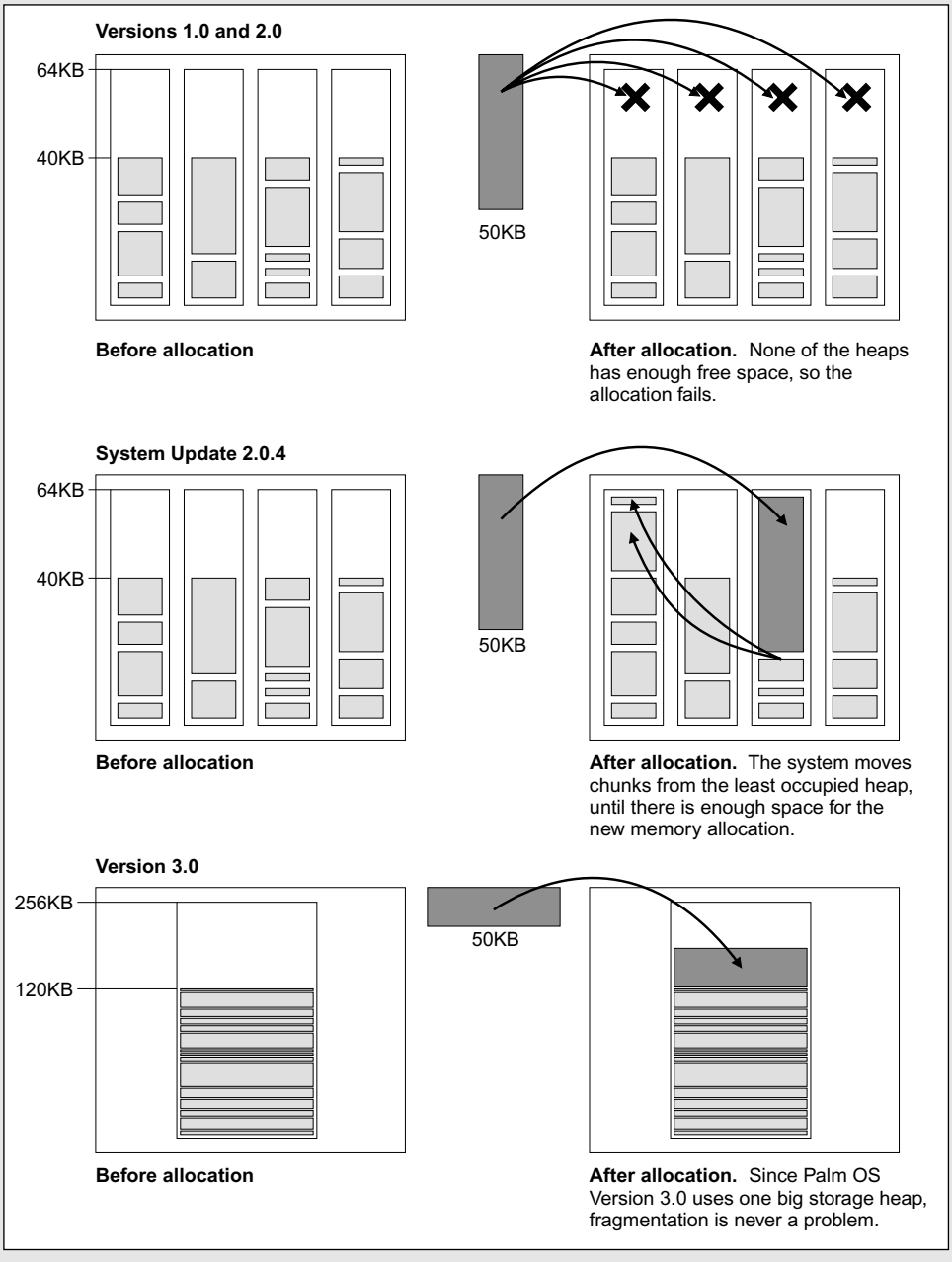
The version 3.*x* use of a single large heap is a big improvement over earlier versions of the Palm OS because it prevents fragmentation of storage memory. Fragmentation occurs as storage heaps fill with data. Even if there is enough total free memory for a new record, there may not be enough contiguous space in any given heap to contain that record.

For example, assume there are four storage heaps, each 64KB in size, with 40KB of memory filled in each heap. There is a total of 96KB of free memory, but if an application tries to allocate 50KB, it won't be able to, because there is, at most, only 24KB available in any given heap. The following figure illustrates this situation and shows how different versions of Palm OS try to deal with this problem. Notice that later versions of the Palm OS deal much better with fragmentation than earlier versions.

Version 1.0 uses an ineffective storage allocation strategy that attempts to keep all the heaps equally full, which in fact causes every new allocation to be more difficult than the last. Version 2.0 improves this a little by allocating memory from the heap with the most free space. System Update 2.0.4 further improves this scheme, moving chunks of memory from the least filled heap to other heaps until there is enough space for the new allocation. Palm OS version 3.0 finally did away with fragmentation problems by putting all storage memory in one big heap.

Fragmentation on earlier Palm devices is another good reason to make your application as small as possible. Not only is there less total RAM available on earlier devices, but memory fragmentation can make what seems like a reasonably sized application impossible to install.

Continued



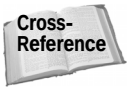
Memory fragmentation in different versions of the Palm OS

Memory chunks in a storage heap are called *records*. Each record is part of a *database* implemented by the Palm OS data manager. A database is simply a list of memory chunks and some database header information. Most of the time, records in a particular database share some kind of association, such as each record representing an appointment in the Date Book.

The Palm OS data manager provides functions for creating, opening, closing, and deleting databases, as well as functions to manipulate records within those databases. A database in the Palm OS serves much the same function as a file in a desktop computer. Depending on the contents of a database's records, a given database may represent an application, a shared library, or simply stored application data.

Because memory is such a limited commodity on a Palm OS handheld, applications do not copy data from a storage heap to the dynamic heap to modify it the way desktop computers copy data from the hard drive to memory. The data and memory managers in the Palm OS lock individual chunks of memory and edit them in place. RAM is used for permanent storage, and even the best programmers can introduce errors into their code that write to the wrong memory address. Because of this, the Palm OS will not allow an application to change the contents of any chunk of storage memory without using the memory and data manager APIs. It is still possible to change the contents of dynamic memory, though, so be sure to use caution when writing to the dynamic heap.

Records in a database may be scattered across multiple storage heaps and interspersed with records from other databases. They may also be located in ROM as part of the applications that ship with the OS. The only restriction on the location of individual records is that all the records in a given database must reside on the same memory card.



Part IV, "Storing Information on the Palm OS Handheld," provides details about using the Palm OS data manager to manipulate databases and records.

Using Resources

A Palm OS application is composed of *resources*, which are blocks that represent data, executable code, user interface elements, and other pieces of the application. Resources may be relocated in memory, so each is identified with a four-byte resource name (such as `tBTN` for a command button) and a two-byte ID number.

Three types of resources are as follows:

- System resources
- Catalog resources
- Project resources

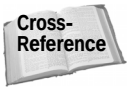
System resources include the application code itself, data structures for initializing the application's global variables, and startup information required by the OS for launching the application. These resources are usually created for you by the development environment from the source code you have written.

Catalog resources include the various user interface elements, from labels to buttons to scroll bars. You must create these resources yourself, supplying identifiers for them so your code will be able to use them during execution.

Project resources include things that will be referenced throughout your application, such as forms, alert dialogs, and menus. You must also create these resources yourself. Some project resources, such as forms, serve as containers for catalog resources and other project resources.

A Palm OS application is really a resource database. It contains all the code, user interface, and other resources necessary to make the application run properly. On the desktop, resource database files end in the extension `.PRC`, so resource databases are often referred to as *PRC files*.

Resources also allow for easier localization of an application. Because all the user interface elements and strings of an application may be kept in separate resources from the application code, translating an application to another language is a simple matter of rebuilding the resources in the new language. Using this modularity ensures that the code running the application need not be changed or recompiled to localize the application.



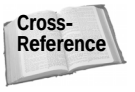
Chapter 6, "Creating and Understanding Resources," covers creating resources in detail.

Designing the User Interface

The Palm OS provides a variety of resources that represent user interface elements. The visible portion of a Palm OS application is where user interaction happens, so it is important to know what tools are available and how they work. More than any other part of an application, the user interface separates a good Palm OS application from one that is frustrating to use.

Every user interface element in the Palm OS is a resource that you build and then compile into your application. Different development environments provide different ways of generating interface resources, but your code will deal with them the same way no matter where they came from.

This section introduces the user interface elements available in the Palm OS, describes their function, and gives examples of each.



Complete details on programming user interface elements are available in Chapter 9.

Forms

A form provides a container, both visual and programmatic, for user interface elements. Forms contain other user interface elements. A given form usually represents a single screen in an application or a modal dialog box. Figure 2-2 shows different forms from the built-in applications. Notice that different sizes of forms are possible.

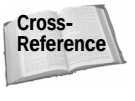
Figure 2-2: Palm OS forms come in different sizes and may contain a variety of user interface elements.

Every application must consist of at least one form, and most contain more than one to display different views and dialogs. Most forms will occupy the entire screen area, except for dialogs, which may occupy less height than a full-screen form but still occupy the entire width of the screen.

Optionally, forms may have the following features:

- A title bar
- An associated menu bar
- A tips icon (only in modal forms)

The tips icon appears as a small circled "i" in the upper-right corner of a form with a title bar (see the third form pictured in Figure 2-2). If the user taps the tips icon, another dialog opens, displaying helpful information about the dialog that contained the icon.



Adding tips to a dialog is covered in Chapter 7, "Building Forms."

Alerts

Alerts provide a facility for displaying simple modal dialogs to the user. An alert dialog is a special kind of form with a title bar, a text message, one or more buttons, and an icon. Alerts may also have a tips icon just like forms.

An alert can be one of four types:

- **Information.** An information dialog displays an "i" icon, which is similar to the tips icon, but larger. It is used to give the user simple information, or to inform the user that the requested action cannot or should not be performed. Such an action should not generate an error or result in data loss. Information alerts can also serve as simple application "about" boxes.
- **Confirmation.** A dialog of the confirmation type displays a "?" icon. It asks the user for input or confirmation and provides a number of buttons from which the user can choose.
- **Warning.** A warning dialog displays an "!" icon. This type of dialog should be used to ask confirmation when the user requests a potentially dangerous action. The difference between a warning and a confirmation dialog is whether the action is reversible or not. Use a confirmation dialog if the action can be reversed or if data deleted as a result of the action can be backed up to the desktop. Use a warning dialog if permanent data loss may result from the action.
- **Error.** An error dialog displays a circular stop sign that contains a white "X." Use this type of alert to inform the user that the last action caused an error or could not be completed.

If sounds are enabled on the device, different types of alerts will also produce different sounds when displayed. Figure 2-3 shows examples of all four types of alerts from the built-in applications.

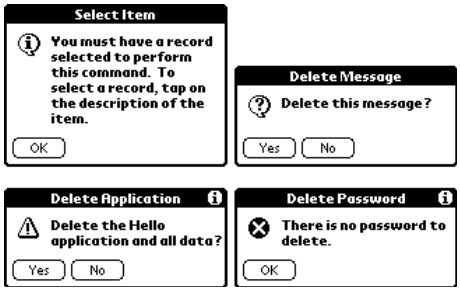


Figure 2-3: Alert dialogs come in four flavors (clockwise from upper left): information, confirmation, error, and warning.

Menus

Menus provide access to commands without occupying precious screen real estate. Each menu contains one or more *menu items*, each of which may have a *command shortcut* assigned to it. A command shortcut is a single character that allows Graffiti access to a menu item's command. If the user enters the Graffiti command stroke, pictured in Figure 2-4, followed by a menu shortcut character, the corresponding menu item is activated. The dot in the figure represents where the stroke should begin.

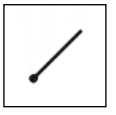
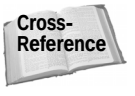


Figure 2-4: The Graffiti command stroke provides quick access to menu commands.

To visually group menu items, use a *separator bar*. In the menu resource, a separator bar is simply another menu item with special properties. There is a separator bar pictured in Figure 2-5, between the "Select All" and "Keyboard" menu items.

Menus themselves are contained in a *menu bar*. There can be only one menu bar assigned to any given form. Figure 2-5 shows a single menu bar, one of its menus, and that menu's menu items.



Creating menu resources is covered in Chapter 8, and programming them is detailed in Chapter 9.

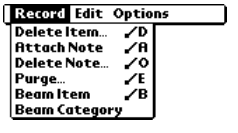


Figure 2-5: A menu bar contains one or more menus, each of which contains menu items.

Tables

Tables are a way to display data in columns. A table may organize a number of other user interface elements within its rows and columns. Objects contained in a row or column of a table often contain the same kind of objects. For example, in a two-column table, the first column might contain labels and the second column text fields.

A table may be scrolled vertically to display more rows of data than will fit on the screen at once. Tables cannot be scrolled horizontally, though. Figure 2-6 shows tables from the built-in To Do List and Address Book applications. Notice the variety of different things that a table's cells may contain. This kind of flexibility makes tables one of the more difficult user interface elements to implement correctly. It also makes them one of the most useful.

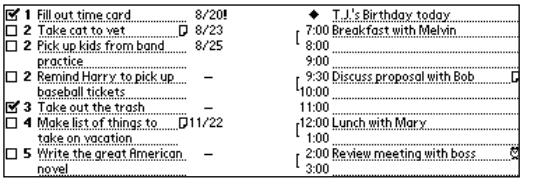


Figure 2-6: Tables are highly customizable, and they may contain many different user interface elements.

Lists

A list is ideal for displaying multiple rows of data in a single column. Like a table, it may scroll vertically to display more items than will fit on the screen at the same time. The Palm OS draws scroll indicators (small arrows) in the corners of a list to indicate that the list may be scrolled up or down to display more items. Unlike a table, a list is not well suited to displaying dynamic data. Use a list to offer static choices to the user; use a table to allow the user to directly edit displayed rows.

List resources may be displayed in two different ways. If you include a list directly in a form and set it to be visible, the system will draw the list with square corners and display it as a static user interface element. Alternatively, you can associate a nonvisible list with a pop-up trigger to create a pop-up list. Drawn on the screen with rounded corners, a pop-up list saves screen real estate by staying hidden until the user actually needs to select an item from it. Instead of occupying screen space with numerous list items, only a single item, displayed in the associated pop-up trigger, needs to show up on the screen. Both types of lists are shown in Figure 2-7.

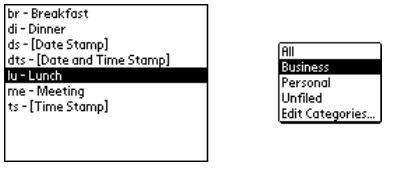


Figure 2-7: Lists may be static user interface elements (left), or they may be pop-up lists associated with a pop-up trigger (right).

Pop-up Triggers

A pop-up trigger consists of a downward-pointing arrow to the left of a text label, which can change its width to accommodate changes in the text. Pop-up triggers allow the user to choose an item from an associated list without occupying precious screen real estate by displaying the entire list. Only the currently selected list item is displayed in the pop-up trigger's label.

When the user taps the arrow or the text label in a pop-up trigger, the trigger's associated list is displayed. If the user taps a new item from the list, the list disappears and the pop-up trigger's caption changes to the newly selected item. If the user taps outside the list while it is displayed, the list disappears and the pop-up trigger's text remains the same.

The most common place where pop-up triggers appear in the built-in applications is in the upper right corner of a form for the selection of a category. Pop-up triggers make very efficient use of screen space. Figure 2-8 shows a pop-up list next to the list that appears when the pop-up trigger is tapped.

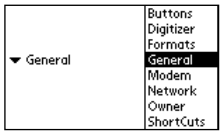


Figure 2-8: A pop-up trigger (left) and its associated list (right)

Buttons

Buttons are used to launch commands or switch to other screens in an application with a single tap of the stylus. A button usually has a rounded frame and contains a text caption, but rectangular and frameless buttons are also possible. Buttons highlight when tapped until the user lifts up the stylus or drags the stylus outside the button's boundaries. Figure 2-9 shows some sample buttons from the built-in applications.

Use buttons for the most frequently used functions in an application. Requiring only a single tap to activate a command, buttons are the quickest user interface element with which the user can interact. Buttons are perfect for creating new records, calling up details on a particular record, and changing between major forms in an application.

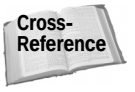
Figure 2-9: Buttons allow for quick access to commonly used commands.

Repeating Buttons

Unlike a button, which sends only one event when tapped, a repeating button continues to put events on the event queue while the user holds the stylus down on it. Repeating buttons are commonly used for scrolling other user interface elements, such as tables.

Although a repeat button may look exactly like a normal button, they are usually defined without borders. The Palm OS has a few symbol fonts that contain arrow characters suitable for use as captions in repeating buttons. Most of the built-in applications use a pair of repeating buttons with arrows in them as scroll controls. Figure 2-10 shows the pair of repeating buttons used for scrolling the To Do List.

Figure 2-10: Repeating buttons serve well to scroll other user interface elements.



Chapter 7 contains more details about setting up repeating button resources to mimic the arrow buttons in the built-in applications.

Selector Triggers

A selector trigger displays a value inside a rectangular box with a dotted-line border. When the user taps the box, a dialog appears that allows the user to change the data displayed in the box. Selector triggers grow or shrink to match the width of the values they display.

The most common use of a selector trigger is to allow selection of a time or date. There are functions in the Palm OS for displaying standard time and date picker dialogs, and these work perfectly with selector triggers. If the data you display in a selector trigger is not a time or a date, or you wish to show a different dialog from those supplied by the OS, you must supply the dialog that appears when the user taps on a selector trigger.

Figure 2-11 shows selector triggers from the Event Details dialog in the built-in Date Book application. Notice that the caption of a selector trigger may be any string of text you choose.

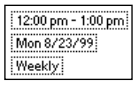


Figure 2-11: Selector triggers display a value that may be edited by the user's tapping on the control.

Push Buttons

Push buttons perform the same function as radio buttons in other graphical interfaces. A push button always occurs in a group of two or more push buttons. Only one button in the group may be selected at a time, and that button is highlighted. Use a group of push buttons when you need to present the user with only a small number of options for a particular value. If you need the user to pick from a large number of values, or if those values may change from time to time, use a list.

Figure 2-12 shows examples of push buttons from the built-in applications.

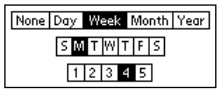


Figure 2-12: Push buttons allow selection of a single item from a group of choices.

Check Boxes

Use a check box to indicate a setting that may be switched either on or off. A check box consists of a square and a text caption. If the setting indicated by the check box is off, the square is empty; if the setting is on, the square contains a check. Tapping either the box or the text caption will toggle the value of a check box. The text caption of a check box always appears to the right of the square. If you want a check box to be labeled on the left, leave the check box's text caption empty and place a label resource to the left of the check box (see the "Labels" section in this chapter).

Like push buttons, check boxes may also be arranged into groups so that only one check box in the group may be checked at a time. Push buttons are better for indicating exclusive choices, though, because they provide a better visual cue that they are part of a group. Check boxes are better for situations where more than one setting may be turned on at a time.

Two check boxes, one checked and the other empty, are shown in Figure 2-13.

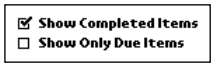


Figure 2-13: Check boxes allow the user to toggle a setting on or off.

Labels

A label is simply a bit of non-editable text that appears on a form or in a table. Use labels to provide descriptions of other user interface elements. For example, placing a label containing the text "Date:" to the left of a selector trigger tells the user that tapping the selector trigger will change the date listed in the selector. Labels also work to provide instructions or information in dialog boxes.

Figure 2-14 shows labels from a few different built-in applications.

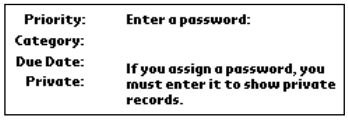
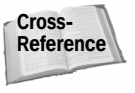


Figure 2-14: Labels describe user interface elements or provide information to the user.

Form Bitmaps

Every form may have one or more form bitmaps associated with it. Form bitmaps are typically used to display icons, such as those used by alert dialogs. A form bitmap also works well as a logo for an about box.

Form bitmaps simply attach a predefined bitmap resource to a form and specify where on the form the bitmap should appear. Figure 2-15 shows an example of a form bitmap.



If a bitmap must be able to change locations within a form, you must use the WinDrawBitmap function. Chapter 9 explains how.

Figure 2-15: A form bitmap

Fields

Fields allow in-place editing of text via Graffiti input or the on-screen keyboard. A text field is also useful for displaying non-editable text that may change as the program runs; labels may be used for this purpose, but they are somewhat more limited in what they can do than a text field.

Fields may be a single line or multiline. Single-line fields may be either left- or right-justified, and they do not accept Tab or Return characters. Multiline fields may be set to change height dynamically, so when text is added or removed from the field, its height expands or contracts to accommodate the text. Scroll bars are often used in conjunction with multiline fields to allow them to contain many pages of text.

The Palm OS keeps track of the current *insertion point*, a blinking cursor that indicates which field in a form is currently active, as well as where newly entered text will appear. Usually, you won't need to worry about the location of the insertion point, because the OS handles all the nitty-gritty implementation details.

Figure 2-16 shows both single-line and multiline text fields.

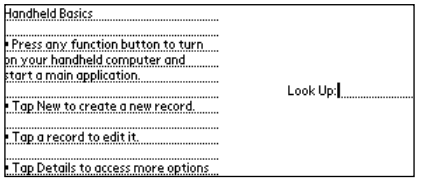


Figure 2-16: Fields are used for text entry and to display changeable strings. Pictured here are a multiline field (left) and a single-line field (right).

Graffiti Shift Indicator

Every form with editable text fields should also contain a Graffiti shift indicator, preferably in the lower right corner of the form. The state indicator shows the current shift state of the Graffiti text entry system: punctuation, symbol, uppercase shift, or uppercase lock. This provides an important visual cue for the user that aids in accurate data entry. Forgetting to add one to a form with fields will make your application frustrating for the user.

If your application is designed to run on Palm OS version 1.0, be sure to leave extra horizontal space for the Graffiti shift indicator. Instead of the underlined arrow used by current versions of the OS to indicate uppercase lock, version 1.0 actually displays the letters "Caps" in the shift indicator.

Fortunately, a Graffiti shift indicator is an easy user interface element to implement; simply include one as a resource on a form, and the Palm OS Graffiti manager will update it automatically as necessary. Figure 2-17 shows a Graffiti shift indicator in its four states. Notice the slight difference in size between the version 1.0 uppercase lock symbol and the symbol used by later versions of the Palm OS.

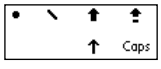


Figure 2-17: The four Graffiti shift states, pictured from left to right, are punctuation, symbol, uppercase shift, and uppercase lock.

Scroll Bars

The scroll bar element allows for vertical scrolling of tables, lists, or multiline fields. The arrow buttons at the top and bottom of a scroll bar can scroll a single line at a time. A solid bar in the middle of the scroll bar, called the *scroll car*, provides a visual indicator of what percentage of the total data contained in the attached field, list, or table is currently displayed on the screen. Users may tap the shaded area above or below the scroll car to move through the data a page at a time, or they may drag the scroll car to navigate directly to a specific location in the data.

Scroll bars are available only in Palm OS version 2.0 and later. If your application must run on version 1.0, use repeating buttons and the hardware scroll buttons instead of a scroll bar.

Implementing a scroll bar requires a certain amount of effort on your part. You must provide two-way communication between the scroll bar and the attached list, table, or field in the following manner:

- When the data in the element attached to the scroll bar changes, your code must alert the scroll bar to the change so it can properly position and size the scroll car.
- When the user taps the scroll bar or its arrows, or drags the scroll car, your code needs to update the list, field, or table to display the appropriate portion of its data. Your application may update the data display in two ways:
  - **Dynamic updating.** As users hold the stylus down on the scroll bar, the data display changes. This method of updating the data provides users with instant feedback about their current location in the data, but it can be slow if the data display is a complex table with many different types of data to draw.

- **Static updating.** The data display changes only after users release the stylus from the scroll bar. This method requires less processing and may be more appropriate for complex tables. It can be frustrating to users, though, because there is no indication of where they are in the data until they let up on the stylus, at which point they must use the scroll bar again if the field, list, or table is not displaying the correct data.

Figure 2-18 shows a scroll bar.

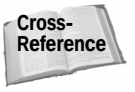
Figure 2-18: A scroll bar

Gadgets

If none of the other user interface elements in the Palm OS will work, you can make a custom user interface element using a gadget. A gadget contains information about its screen location, whether it is currently usable or not, and a pointer to a piece of data. You must implement everything else, from drawing the gadget to responding to stylus taps.

Because you have to do the bulk of the work to implement a gadget anyway, you may be thinking that you might as well code your own custom interface object from scratch. The gadget does offer some advantages over rolling your own object, though:

- ✦ Gadgets keep track of their rectangular bounds on the screen, making it easy to detect whether a particular tap on the screen was on the gadget or not. This also makes any drawing code for your gadget more portable, because it can draw relative to the gadget's bounds instead of requiring hard-coded screen coordinates. You can then use your gadget code in a different application, or even in a different location on the same form, and you will not need to rewrite a lot of your code.
- ✦ A gadget maintains a pointer to whatever data you wish to associate with the gadget.
- ✦ The Palm OS Emulator (POSE) has a testing feature called Gremlins that can randomly poke at your application and uncover obscure bugs that you might otherwise miss. Gremlins occasionally tap on random areas of the screen that don't contain any controls, but they are particularly attracted to standard user interface elements. Coding a custom element as a gadget ensures that Gremlins will give your custom interface a good workout.



More information about Gremlins is available in Chapter 5.

Figure 2-19 shows a gadget from the built-in Date Book's month view. This complex gadget draws a calendar view and indicates appointments with symbols. The user can pick a particular day by tapping it; this will display that particular day in a different screen.

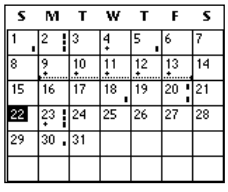


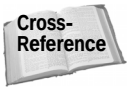
Figure 2-19: The month view gadget from the built-in Date Book application can display a lot of information at a glance.

Communicating with Other Devices

A key part of the Palm Computing platform's success is its ability to communicate with other devices. Current versions of the Palm OS offer a number of different communications protocols.

Serial

Palm OS devices use the serial protocol to synchronize through a cradle with a desktop computer. With the right cable or third-party hardware, the Palm OS can also talk to just about anything, from modems to temperature probes to GPS receivers. The Palm OS serial communications architecture supports several layers of varying utility and complexity, including byte-level serial I/O and high-level error-correcting protocols.



Palm OS serial communications are covered in detail in Chapter 15, "Using the Serial Port."

TCP/IP

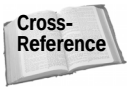
The standard protocol of the Internet, TCP/IP, allows a Palm OS device with the proper attached hardware to connect to any machine on the Internet and exchange data with it. Most of the functions in the Palm OS net library, which provides TCP/IP connectivity in Palm OS applications, are the spitting image of functions in the Berkeley Unix sockets API, which is the de facto standard for Internet applications. Applications written to use Berkeley sockets can be easily recompiled for the Palm OS with only a few changes.

Wireless

Introduced in Palm OS version 3.2, which is installed on the Palm VII, wireless communication on a Palm OS device takes place via the Palm.Net wireless network. Because wireless bandwidth is expensive, the Palm OS wireless system uses a strategy called *Web clipping* to minimize the amount of data that must be transferred between the Palm OS device's wireless radio and the network. A wireless Palm OS user runs a Web clipping application to request information, which is then displayed by Clipper, the browser application resident on the Palm OS device.

Applications cannot directly access the Internet through the Palm device's wireless modem. This is a deliberate limitation to minimize the amount of data sent across expensive wireless connections.

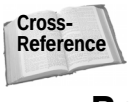
On the client side, PQAs are easy to make, because they are coded in Hypertext Markup Language (HTML) with only a few Palm OS–specific additions, and then compiled using the free Query Application Builder (QAB). The server end of a PQA may be constructed using existing Web technologies, because the Palm.Net service communicates with the server via standard Hypertext Transfer Protocol (HTTP).



More information about the Palm.Net service and creating PQAs may be found in Part VII.

IrDA

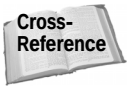
Starting with the Palm III and Palm OS version 3.0, Palm devices can communicate via the industry-standard Infrared Data Association (IrDA) protocol. This low-level communications protocol can be used to communicate via infrared (IR) with a growing variety of similarly equipped devices, including cell phones, pagers, and even desktop or laptop computers. Like the serial manager, the Palm OS infrared library offers low-level control of IR data transfer.



IrDA communication is covered in detail in Chapter 14.

Beaming

The Palm OS exchange manager provides facilities for beaming individual records and applications between two devices via infrared (IR). Although primarily used to beam information between two Palm OS handhelds, the exchange manager is a generic communications method that allows exchange of typed data objects between different devices. The exchange manager runs on top of the IrDA transfer implemented by the Palm OS infrared library.

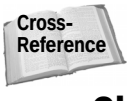


More information on beaming via the exchange manager is available in Chapter 13.

Comparing Palm OS Versions

Many of the changes between different models of Palm OS devices that you will need to keep in mind are changes to the Palm OS itself. Fortunately, just as Palm Computing and its partners have changed the hardware incrementally, the Palm OS has also evolved at an easy pace, making backwards compatibility much easier to implement. This section provides a brief overview of what has changed since version 1.0 of the Palm OS.

Because a number of new functions have been added to the Palm OS with each new version, the system provides facilities to easily determine what features are supported in the currently running environment. If your application uses functions from newer versions of the OS, it will run more smoothly if you check for the existence of those features before calling them. Checking for the version number of the operating system alone is not enough because future versions of the Palm OS will not necessarily implement all the features of earlier versions. Instead, the system can query whether specific *feature sets* are present in the version of the Palm OS on which your application is running.



For more detail about checking for the presence of feature sets, see Chapter 10.

Changes in Version 2.0

Features added to version 2.0 include:

- Scroll bars and associated functions for manipulating them
- New launch codes to support phone lookup and access to the system preferences panel
- TCP/IP support (only on devices with 1MB or more of memory)
- IEEE floating point math, including 32-bit floats and 64-bit doubles
- System-wide Graffiti reference dialog
- New string manipulation functions

Features changed from those in earlier versions include:

- Application preferences
- System keyboard dialog
- Edit categories dialog

Changes in Version 3.0

Features added to version 3.0 include:

- ✦ IR beaming
- ✦ A large bold font
- ✦ Dynamic user interface functions
- ✦ Custom fonts
- ✦ Progress dialog manager
- ✦ Unique device ID on hardware with flash ROM
- ✦ File streaming to support records larger than 64KB
- ✦ Support for Standard MIDI Files (SMF) and asynchronous sound playback

Features changed from those in earlier versions include:

- ✦ Further changes to the edit categories dialog
- ✦ Dynamic heap increased to 96KB in size
- ✦ Storage RAM configured as a single heap instead of multiple 64KB heaps
- ✦ Application launcher becomes an actual application rather than a system pop-up

Changes in Version 3.1

Features added to version 3.1 include:

- ✦ Contrast adjustment dialog (on devices in the Palm V family only)
- ✦ Support for the DragonBall EZ processor

Features changed from those in earlier versions include:

- ✦ Character encoding changed to match Microsoft Windows code page 1252
- ✦ Text fields may now have either dotted or solid underlines
- ✦ Character variables changed to be two bytes long

Changes in Version 3.2

Features added to version 3.2 include:

- ✦ Function to append data to the clipboard without erasing its current contents

- ✦ Alert dialog for runtime errors, to be used when a runtime error is not the application's fault (for instance, in networking applications)

Changes in Version 3.3

Features added to version 3.3 include:

- ✦ Support for the Euro currency symbol
- ✦ New Serial Manager introduced, adding more flexible serial connection capabilities, such as serial connections via infrared and support for the IrCOMM standard
- ✦ Login script enhancements for connecting to remote systems that use token-based authentication
- ✦ Faster HotSync operations, as well as HotSync operations via infrared

Changes in Version 3.5

Features added to version 3.5 include:

- ✦ Color screen and drawing support
- ✦ New data type definitions (for example, `UInt16` instead of `Word`)
- ✦ Command bar containing buttons for commonly used menu items
- ✦ Slider and repeating slider controls
- ✦ Graphical controls
- ✦ Overlay manager to allow easier localization of applications without requiring complete recompilation
- ✦ New security routines to allow changing hidden record status from within an application, instead of having to rely on the Security applet
- ✦ New table routines to implement masked records

Features changed from earlier versions include:

- ✦ Extended gadget support, including the ability to assign a callback function to a gadget to handle gadget events
- ✦ Text fields allow double taps to select words, or triple taps to select lines of text
- ✦ Menus may be displayed by tapping an application's title bar

Summary

This chapter gave you a whirlwind tour of the features of the Palm OS, and it explained a little about how many of them work. You should now know the following:

- The Palm OS power management scheme works in such a way that the device is never really "off," only resting.
- Palm OS applications respond to launch codes when they start, and if the code calls for a normal launch, they enter an event loop to process the system event queue.
- Memory in the Palm OS is divided into dynamic and storage areas, each with its own unique limitations.
- A Palm OS application is composed of resources, some of which are built by the development environment, and some of which must be supplied by the developer.
- User interface elements abound in the Palm OS, and if none of the provided elements will do the job in your application, you can always make your own using the gadget resource.
- The Palm OS provides numerous protocols for communicating with other devices.
- If your application uses features that were introduced in a recent version of the Palm OS, it can easily check its environment to see what features are available before calling a potentially unsupported function.

✦ ✦ ✦

CHAPTER 3

Introducing the Development Environments

In This Chapter

Introduction to Metrowerks CodeWarrior for Palm Computing platform

Introduction to the GNU PRC-Tools

Organizing projects in CodeWarrior

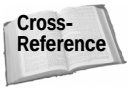
Compiling and linking with CodeWarrior

Compiling and linking with the GNU PRC-Tools

Using make to build applications

A great number of development environments exist for creating Palm OS applications. Of the many tools available, C and C++ are the most common languages used to develop applications for Palm Computing platform.

This chapter introduces two suites of tools for C/C++ Palm development. Metrowerks CodeWarrior for Palm Computing platform runs on both Mac OS and Windows systems. The GNU PRC-Tools are a free alternative, available on both the Windows and Unix platforms.



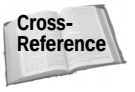
The myriad other Palm OS development systems are outlined in Appendix C, "Developing in Other Environments."

Using CodeWarrior for Palm OS

Metrowerks CodeWarrior for Palm Computing platform is an integrated development environment (IDE), containing all the tools you need to develop Palm OS applications in a single interface. CodeWarrior is the official development environment supported by Palm Computing; in fact, the Palm OS documentation and tutorial provided by Palm assume you are using CodeWarrior to make your applications.

The CodeWarrior package contains a number of tools:

- **Constructor for Palm OS**. Constructor is a resource editor with a graphical interface. You use Constructor to build the user interface elements of your application, which the other CodeWarrior tools then combine with your source code to create a finished program.



Complete details of using Constructor to create and edit Palm OS resources are available in Chapter 6, "Creating and Understanding Resources"; Chapter 7, "Building Forms"; and Chapter 8, "Building Menus."

- **CodeWarrior IDE**. The CodeWarrior IDE is the interface for all the CodeWarrior tools except for Constructor. From within the IDE, you can edit source code, compile and link applications, debug your program, and organize your project's source code and resource files. Many of the tools in CodeWarrior for Palm Computing platform are plugins that attach to the IDE. The IDE also contains CodeWarrior's source-level debugger, which can debug applications running on either the Palm OS Emulator, or on a real Palm OS handheld connected to the computer via a serial cable.
- **CodeWarrior Compiler for Palm OS**. CodeWarrior's compiler for Palm OS turns ANSI C/C++ code into object code for Motorola 68000–series processors.
- **CodeWarrior Linker for Palm OS**. The linker used by CodeWarrior to link compiled object code is actually the same linker used in other versions of CodeWarrior to create Mac OS programs.
- **CodeWarrior Assembler for Palm OS**. The Assembler for Palm OS creates executable code from Motorola 68000 assembly instructions. CodeWarrior's C/C++ compiler also supports inline assembly statements.
- **PalmRez**. The PalmRez plugin changes the linked object code generated by other parts of CodeWarrior into a `.prc` file suitable for installation and execution on a Palm device or in the Palm OS Emulator.
- **Palm OS Simulator**. Available only in the Mac version of the CodeWarrior for Palm Computing platform tools, the Palm OS Simulator allows you to execute and test a Palm OS application. Building an application to run under the Simulator is different from building a finished Palm OS executable program. Simulator applications contain a library, added at link time, that allows them to run as independent applications on a Mac OS computer. When the CodeWarrior tools were first released, the Palm OS Simulator was the only way to test Palm OS applications, short of downloading them to an actual device.
- **Palm OS Emulator**. Also known as POSE, the Palm OS Emulator imitates most of the hardware and software functions of an actual Palm OS handheld. Among other things, POSE accurately emulates the actual processors used in Palm OS devices. You can load real Palm OS applications into POSE, without needing to specially compile them as you would for the Palm OS Simulator. POSE also happens to be available for Windows, Mac OS, and Unix systems. Greg Hewgill, with help from other developers, originally wrote POSE as "Copilot." POSE receives a lot of development attention from both Palm Computing and other developers; check Palm's Web site (`www.palmos.com`) for newer versions than the one that ships with CodeWarrior.

CodeWarrior was originally designed as a Macintosh development tool. This is good news for many Macintosh developers. Chances are pretty good that if you have done any software development work on the Mac OS, you were using CodeWarrior, so you will have little trouble adapting to the few new things added to CodeWarrior for Palm Computing platform.

Unfortunately, very little in the CodeWarrior interface changed when Metrowerks ported the development tools to Windows. Developers used to Windows may find CodeWarrior's interface quirky and non-intuitive, if not downright irritating. The Windows version of CodeWarrior is still a very effective and dependable Palm OS development tool, but it may take some getting used to if you have a lot of experience with the Windows environment. The instructions in this section of the book are applicable to both the Mac OS and Windows versions of CodeWarrior for Palm Computing platform. Screen shots of CodeWarrior in this book only depict the Windows version, but the differences between the two interfaces are minor.

Familiarizing Yourself with the IDE

The CodeWarrior IDE, pictured in Figure 3-1, is a multiple document interface (MDI) that provides a container for the IDE's various menus, buttons, and windows. It is possible to work on more than one application's components at a time in the IDE, making it easy to borrow source code and resources from one program to use in another.

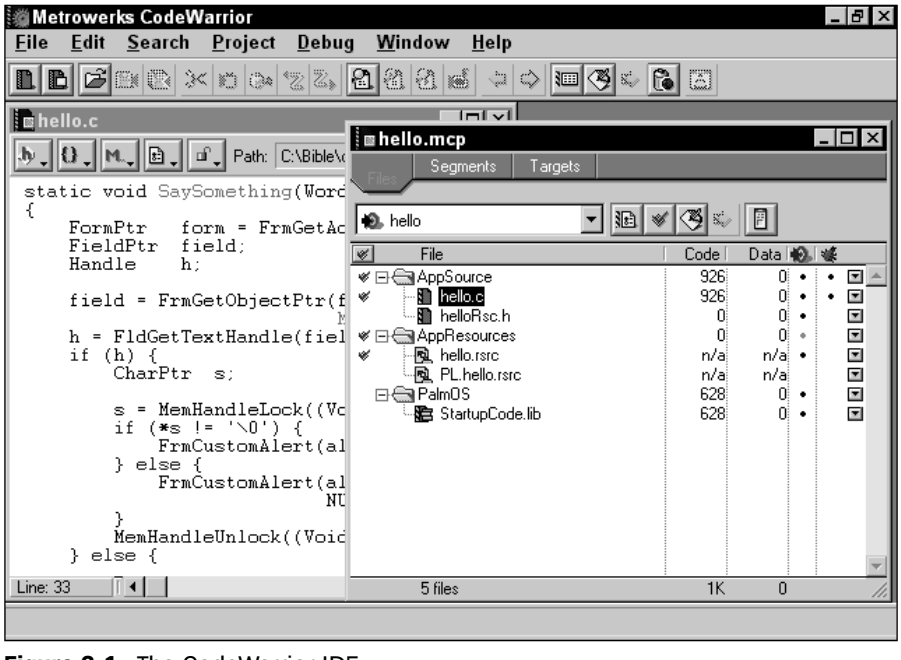


Figure 3-1: The CodeWarrior IDE

CodeWarrior organizes application development into *projects*. A project contains references to all the different source code and resource files that make up a particular application. You also use projects to save settings for building your application in different ways. Each different way to build an application is called a *target*. Having multiple targets within a single project is a useful way to generate both debug and release versions of an application, or to easily create many versions localized for various languages. Different targets may compile entirely different files or use completely different compiler and linker settings.

Opening a project

All the information about a particular CodeWarrior project resides in a *project file*, which has an `.mcp` extension in Windows. To open an existing project file, select File➪Open or press Ctrl+O. You may also use CodeWarrior's recently opened file list to open a project or source code file that you worked on in a previous session. Select File➪Open Recent to access the most recently used file list.

Creating a new project

When creating a new project, you can choose to either create an empty project, or you can use CodeWarrior's *project stationery*. Project stationery is a template for creating a particular kind of application, containing boilerplate code for common parts of the application. After creating a project from stationery, you then add your own code to the appropriate parts of the source files that CodeWarrior generates.

Start the project creation process by selecting File➪New Project or by pressing Ctrl+Shift+N. CodeWarrior presents you with the New Project dialog box, pictured in Figure 3-2.

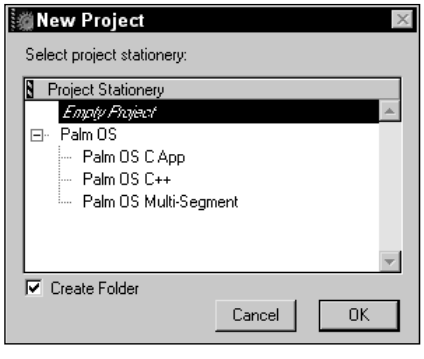


Figure 3-2: The New Project dialog box

To create an empty project, select "Empty Project" in the Project Stationery tree control, then press OK. Leave the Create Folder check box checked if you want CodeWarrior to create a folder to contain the new project. Find the location in which you wish to create the new project, give it a filename, and press Save.

To create a project from stationery, expand the "Palm OS" group in the Project Stationery list to display the different kinds of project stationery available. Select the stationery you wish to use, then press OK. CodeWarrior prompts you for the filename and location where you wish to create the new project. Once you have entered the appropriate information, press Save.

Which method of creating a project is better? At first glance, it looks like a stationery project should save you a lot of time. Projects created from stationery already contain a lot of the code necessary for a basic Palm OS application, which saves having to type a lot of code that you will use over and over again for different Palm OS applications. Stationery projects provide a rudimentary resource framework, with a single form and the beginnings of a menu system in place. Using stationery also configures project settings to compile a Palm OS application properly; the default settings in an empty project do not work. The stationery code also has the distinct advantage of working. Before you add your own code to it, a stationery project compiles without error, giving you a bug-free baseline on which to base your project.

Unfortunately, CodeWarrior's Palm OS stationery invariably creates a project called "Starter," whose file names are also based on the word "Starter" instead of the name you assigned to the new project. There are a few things in the automatically generated source code and resource files that are undesirable for many applications. For example, the stationery project implements a menu command to display an about box, but it uses the Palm OS **AbtShowAbout** function, which displays only the Palm Computing–specific about box shared by the built-in applications. The resources in the generic "Starter" application may not meet the needs of your program, either. As if that weren't enough, you still need to edit the target settings to properly set things like output file names and PalmRez options. By the time you finish renaming the files, changing the resources, ripping out unwanted code, and modifying target settings, you may not save any time over creating the whole project from scratch.

What the stationery does particularly well is to serve as a source of raw materials. Even when creating an empty project, I have found it practical to create a throwaway stationery project, cut and paste useful chunks of its source code into the "real" project I am working on, then delete the stationery project when I have stripped it of everything I need. This "strip-mining" approach still benefits from the correctness of the boilerplate code, while avoiding the annoyance of having to weed inappropriate bits of code from your project.

Exploring the project window

Once you have created or opened a project, CodeWarrior displays the project window, shown in Figure 3-3. From the project window, you can control which source files a particular project contains, how CodeWarrior compiles and links those files, and what build targets are available.

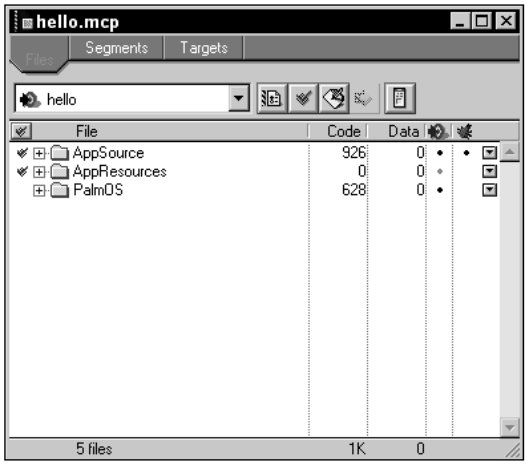


Figure 3-3: The project window

The drop-down in the upper left of the project window is for selecting the current build target. Changing the target displayed in the drop-down changes the rest of the window's display to reflect the settings for that particular target. Also, when the project window has the focus, the currently displayed target in this drop-down is what CodeWarrior will build when running a Make, Debug, or Bring Up To Date command from the Project menu.

The project window has three views: File, Segment, and Target. To display a particular view, select the appropriate tab at the top of the project window.

Managing files in the file view

The File view gives you control over what source code and resource files are part of a project. Figure 3-4 shows the File view for a simple application.

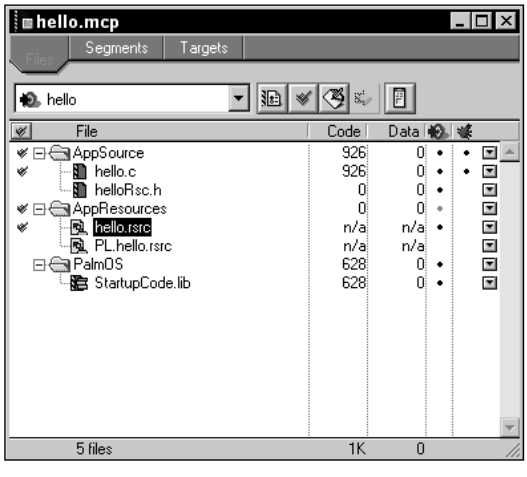


Figure 3-4: The project window's File view

The columns in the File view, from left to right, are described below:

- **Touch column.** The column with a check mark at the top indicates which files have been changed, or *touched*, since the last build, and therefore need to be compiled when building the project. Touched files have a check mark next to them. You can toggle whether a file is touched or not by clicking next to that file in the touch column. In Windows, holding down Alt while clicking in this column toggles the touch status of all the files in the project. Holding down the Option key performs the same action in the Mac OS.
- **File column.** This column lists all the files contained in the project. You can use *groups* to organize files. Selecting Project➪Create New Group... creates a new group, and selecting Project➪Add Files... presents you with a dialog to select new files to add to the project. You may change the order in which files are displayed, as well as which groups files occupy, by dragging them around the project window. Double-clicking a file, or pressing Enter if the file is currently selected, opens the file for editing. Groups may also contain subgroups if your project is complex enough to require that kind of organization.
- **Code column.** Code shows the size of the compiled object code associated with a particular source file or group of source files. A zero (“0”) in this column indicates code that CodeWarrior has not compiled yet. The total of the values does not necessarily add up to the total size of the compiled program. When linking, CodeWarrior may not use all the object code from a particular source, leaving out dead code that the rest of the project does not reference.
- **Data column.** The Data column shows the size of any non-executable data residing in the object code for a particular source file. If the source file is uncompiled, or if it contains no data section, this column displays a zero (“0”).
- **Target column.** Target has a bull’s-eye with an arrow pointing to it at the head of the column. CodeWarrior displays this column only if a project contains multiple targets, so many simple Palm OS applications will never need this column. A black dot in this column indicates that a particular file is part of the currently selected target.
- **Debug column.** Indicated by a small green bug, the debug column displays a dot next to any file that should contain debugging information when built. Clicking in this column toggles whether or not CodeWarrior includes debugging information in a file when building it.
- **Interface pop-up column.** The column full of small buttons with downward-pointing arrows is the interface pop-up column. Clicking one of these buttons displays a pop-up menu that performs different functions, depending on the type of item displayed in that row:
  - For file groups, the pop-up list contains a list of all the files in that group. Choosing a file from this list opens that file for editing.
  - For files, the pop-up list shows a list of header files included by that source file. Picking one of the files from the list opens it for editing. The interface pop-up menu also offers an option to touch or untouch that file.

Removing files from the project may be accomplished by selecting them, then either choosing the menu command Project➪Remove Selected Items or pressing Ctrl+Del.

You may also remove a file in Windows by right-clicking it, then selecting Remove Selected Items from the pop-up menu that appears. In the Mac OS, holding down Control while clicking a file opens the same pop-up menu.

Controlling link order with the segment view

The Segment view of the project window, pictured in Figure 3-5, controls the order in which CodeWarrior links your project's source files together. The linker follows the same order, from top to bottom, that is displayed in the Segment view. To change the link order, simply drag the files in the list until they are in the appropriate positions.

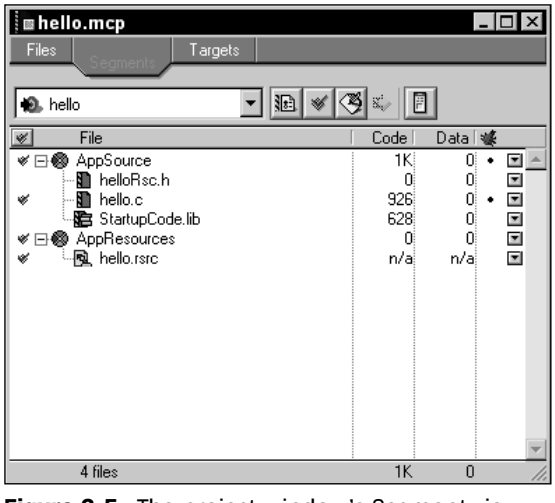
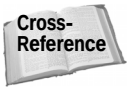


Figure 3-5: The project window's Segment view

Within the Segment view, you may group files into different *segments*. In a small application, segments are mostly just a way to organize the project into logical groups, and they function similarly to file groups in the project window's File view. For large applications composed of more than 64KB of compiled code, segments must be used to partition your source code into smaller chunks, resulting in a *multi-segment application*. Most Palm OS applications should be small enough to not require segmentation.



Full details on building multi-segment applications are available in Chapter 20, "Odds and Ends."

Creating different builds in the target view

The Target view of the project window, pictured in Figure 3-6, is where you define different build targets for the application. Any project must contain at least one target, and CodeWarrior generates a target with the same name as the project file when you create a new project.

Figure 3-6: The project window's Target view

To create a new target, select Project➪Create New Target. The New Target dialog, shown in Figure 3-7, appears.

Figure 3-7: The New Target dialog

The New Target dialog prompts you for a name to call the target you are creating. You can also choose to create an empty target by selecting the Empty target option,

or copy the settings from an existing target by choosing the Clone existing target option. Once you have named and determined the contents of the new target, press OK.

Since the default settings of an empty target do not work for compiling a Palm OS application, you can save yourself a lot of time by cloning an existing target whose settings already work. Before adding new targets to a project, change all the settings in your project's first target to appropriate values, then clone it when you add more targets to the project.

Once you have all the targets you need for your application, you may assign files to those targets from the File view of the project window. You may also assign targets for a particular file from the Project Inspector window, described later in this chapter.

Saving a project

CodeWarrior automatically saves changes to the project when you perform any of the following actions:

- ✦ Close the project.
- ✦ Change the Preferences or Target Settings of the project.
- ✦ Add files to or delete files from the project.
- ✦ Compile any file in the project.
- ✦ Edit any groups in the project.
- ✦ Remove any object code from the project.
- ✦ Quit the CodeWarrior IDE.

When saving changes to your project, CodeWarrior saves the names of your project's files and their locations, all the configuration options for the project, dependency information and touch state for all files, and the object code compiled from the project's source files. Since CodeWarrior saves all this information automatically, even when closing the IDE, you never have to manually save your project. Should you wish to copy your project, you can use the File➪Save A Copy As command to do so.

Changing Target Settings

The target settings dialog, pictured in Figure 3-8, is where you can change a wide variety of options that affect how the compiler and linkers assemble your project's code for a specific target. To access the target settings dialog, select Edit➪*target*

settings, where *target* represents the name of the current target selected in the project window. Double-clicking the target's name in the project window's Target view also opens the target settings dialog, and in Windows, you can also press Alt+F7.

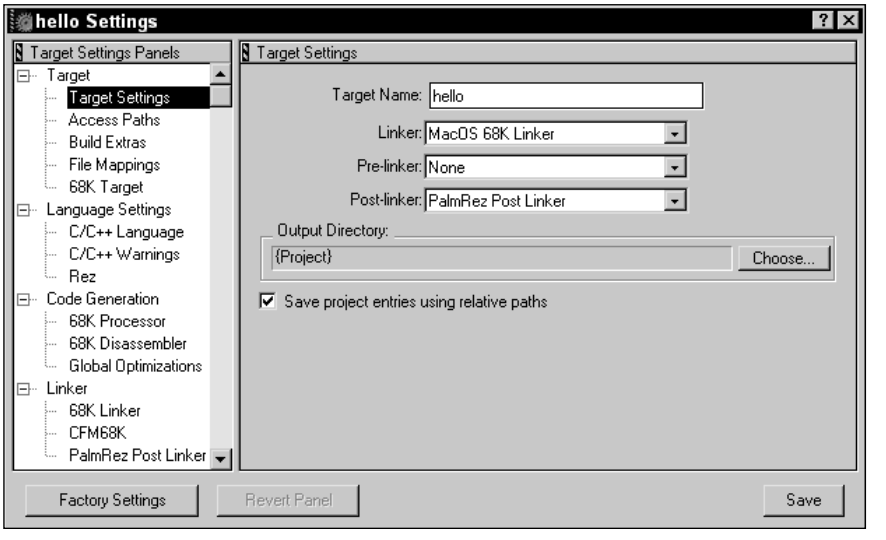


Figure 3-8: The target settings dialog

The left pane of the target settings dialog, labeled Target Settings Panels, shows a list of all the different settings panels, which appear in the right pane of the dialog. Select an item from the list to display its panel. There are a bewildering number of options in the target settings dialog, not all of which are directly applicable to Palm OS development. The CodeWarrior documentation does a good job of describing all the bells and whistles, so only selected panels and those settings that are critical for the compiling of a normal Palm OS application appear in the discussion below.

The Factory Settings button returns all the panels in the target settings dialog to their default state. After you make any changes to settings, the Revert Panel button becomes active. Clicking Revert Panel restores the current settings panel to the state it was in the last time you saved the settings. Click Save to save changes you have made to the target settings.

The default state of the panels in the target settings dialog box does not properly compile working Palm OS applications. The Factory Settings button is a useful feature for other versions of CodeWarrior that target different platforms, but you should never need to use it when developing for the Palm OS. If you did not use project stationery to create your project, or if you created an empty target, be sure to copy the settings from a stationery project to avoid compilation errors.

The first panel listed is Target Settings, underneath the Target category. This panel controls very general settings for the current target. You can rename the target in the Target Name text box. The drop-downs allow you to choose the linkers CodeWarrior should use to link the application. For Palm OS development, Linker should be set to "MacOS 68K Linker," Pre-linker should be "None," and Post-linker should be "PalmRez Post Linker." Checking the Save project entries using relative paths check box allows you to move a project to another location without disturbing the paths saved in the project file.

Farther down the Target category, and visible only if Linker in the Target Settings panel is set to "MacOS 68K Linker," is the 68K Target panel, shown in Figure 3-9. For Palm OS applications, the Project Type drop-down should read "PalmOS Application," and the File Name text box should contain something like "*project*.tmp", where *project* is the name of the project.

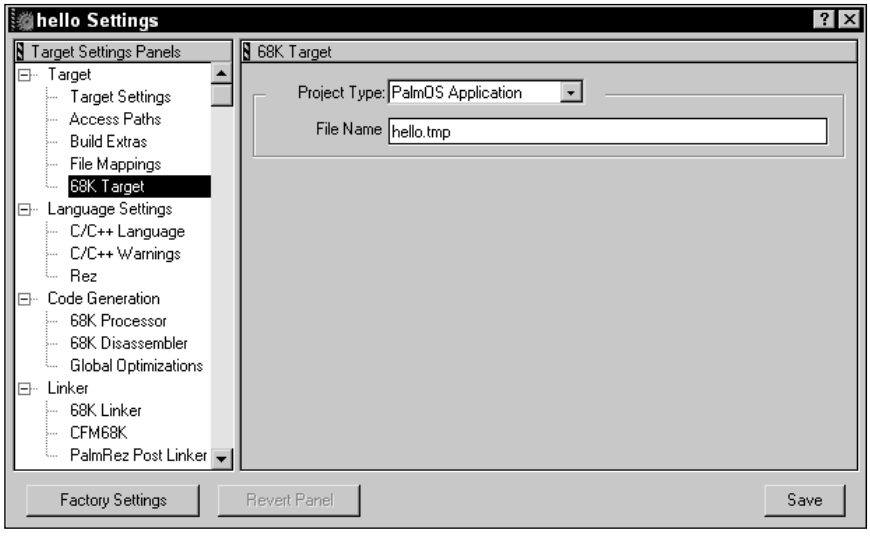


Figure 3-9: The 68K Target panel

The last panel with important Palm OS development settings is PalmRez Post Linker, under the Linker group. This panel, pictured in Figure 3-10, controls the settings for PalmRez, which is responsible for converting the Motorola 68000 code compiled by CodeWarrior into the `.prc` format understood by the Palm OS. The Mac Resource Files text box should contain the same file name you supplied for the File Name text box in the 68K Target panel. Enter the file name for the `.prc` file that will contain the

finished application in the Output File text box. Type should be set to “appl” for a standard Palm OS application. The Creator field should contain your application’s *Creator ID*, a four-byte code that uniquely identifies your application.

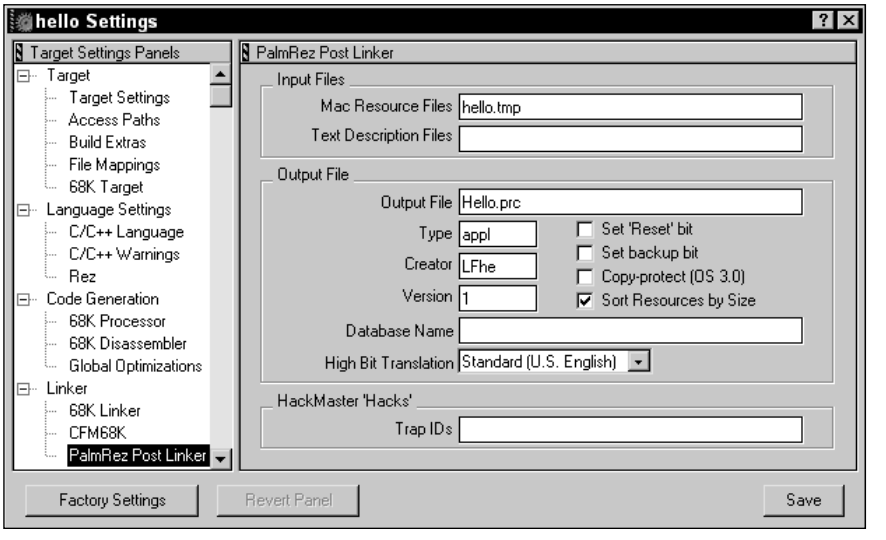


Figure 3-10: The PalmRez Post Linker panel

Every database on a Palm OS device, including each application, has a database name, which you can set in the Database Name field. Applications and databases must have unique names. Setting the application’s database name using the Database Name field is optional, though; if you leave this field blank, CodeWarrior will strip the `.prc` from the end of the file name in the Output File field and use that as the database name for the application. Just be sure to pick a file name that will be unique once it is on the handheld.

Checking the Set ‘Reset’ bit check box signals the Palm OS to reset the device after this application has been installed through a HotSync operation. This feature is needed only by applications that modify basic operating system behavior; most applications do not need the system to reset the device when they are installed. The Set backup bit check box controls whether or not the HotSync Manager should copy this `.prc` file to the user’s backup folder when synchronizing with the desktop. Copy-protect (OS 3.0), when checked, prevents Palm OS version 3.0 or later from beaming the application to another device via the infrared port. It has no effect on versions of the Palm OS that do not include IR beaming.

What Is a Creator ID?

All Palm OS applications and databases have a four-byte Creator ID to uniquely identify them to the operating system. To prevent your application from conflicting with others, you need to register a Creator ID with Palm Computing, which maintains a database of registered IDs. Creator ID registration is simple; just point your browser at `http://www.palmos.com/dev` and follow the Quick Index to Creator ID. From there, you may browse the list of registered Creator IDs and choose one that is not already in use.

Creator IDs are case-sensitive, composed of four ASCII characters in the range 33-127, decimal. Palm Computing has reserved Creator IDs composed entirely of lowercase letters for their own use, so your own Creator IDs must have at least one capital letter or symbol character.

Any application you release to the public, or even within a corporation, should have its own unique Creator ID. Applications with identical Creator IDs wipe each other out when installed to the same device, and data corruption is a definite possibility.

Compiling and Linking in CodeWarrior

CodeWarrior gives you the option to compile source files one at a time, a few at a time, or every source file all at once. Compiling produces only the object code for the appropriate source files, without linking them into a complete application. To compile one or more source files, select the desired files in the project window, then do one of the following:

- Select Project➪Compile.
- Press Ctrl+F7.
- In Windows, right-click the selected file or files and choose Compile from the pop-up menu. In the Mac OS, hold down Control while clicking to access the same menu.

If you have changed or added many files, you may wish to update the entire project at once. To do this, select Project➪Bring Up to Date. This command compiles all source code that has either not been compiled or has not been touched.

CodeWarrior sometimes does not recognize that you have made changes to a file. To force recompiling, touch the file first, then compile it.

To link all the object code in your project into a completed binary file, select Project➪Make, or press F7. Running the `Make` command first checks for newly added, modified, or touched files and compiles them. Then, `Make` runs the compiled object code through the linkers to produce a finished executable program.

Since the CodeWarrior IDE allows you to work on multiple projects containing multiple targets, determining which files CodeWarrior compiles and links can be confusing. When running a global command like `Bring Up to Date` or `Make`, CodeWarrior determines which target from which project to build based on the following rules:

- If a project window has the focus, CodeWarrior builds the currently selected target in that project window's target drop-down.
- If a different window has the focus, such as a source code editing window, CodeWarrior relies on its *default project* and *default target* settings.

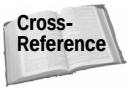
To set the default project, select Project ➪ Set Default Project, then select the appropriate project file. Likewise, to set the default target, select Project ➪ Set Default Target, then select the appropriate target.

Using the GNU PRC-Tools

When Metrowerks first released the development tools for Palm OS development, the only supported platform was the Mac OS. The open source development community, never willing to let a small obstacle like lack of development tools get between them and creating code for a new device, immediately began work on a free set of tools for Palm OS programming. The result is the GNU PRC-Tools, available for both Unix and Windows systems.

The GNU PRC-Tools package contains a modified version of the GNU C compiler (gcc), one of the most popular C/C++ compilers in the Unix world, and also used by many Windows developers. Often referred to as just GCC by Palm OS developers, the PRC-Tools have expanded to include a complete suite of code-editing and debugging utilities. The following are included in the GNU PRC-Tools:

- **M68K GNU C Compiler.** The heart of the PRC-Tools, this compiler transforms C/C++ source code into Motorola 68000 object code.
- **PilRC.** The PilRC tool is a resource compiler that transforms text descriptions of user interface and other resources into the correct binary format expected by the Palm OS.
- **PilrcUI.** The PilrcUI program provides a visual preview of the resources described in a PilRC source file.



Complete details of using PilRC and PilrcUI to create and edit Palm OS resources are available in Chapter 6, "Creating and Understanding Resources"; Chapter 7, "Building Forms"; and Chapter 8, "Building Menus."

- **build-prc.** This utility converts the Motorola 68000 code into the `.prc` format expected by the Palm OS, and also combines that code with PilRC-created resources to produce a finished executable program.

- **gdb.** The GNU debugger is a tool for debugging Palm OS applications at the source code level.
- **Copilot.** The first true Palm OS device emulator, Copilot accurately imitates a Palm OS device's hardware and operating system. You can install actual Palm OS programs in Copilot, and they behave almost exactly as they would on a real device. This, combined with its ability to connect to GDB for source-level debugging, makes Copilot an invaluable tool for Palm OS programming. Development of Copilot has passed from Greg Hewgill, its creator, to Palm Computing, which now releases it as POSE, the Palm OS Emulator. POSE is more up-to-date than Copilot and contains more useful features, so it is well worth the download time to pick up the latest version of POSE from Palm Computing's developer Web site.

Since its original release for Unix operating systems, developers in the open source community have also ported the GNU PRC-Tools to Windows, using the Cygnus Win32 version of gcc as a base. Palm Computing actually provides official support for the PRC-Tools, alongside the commercial CodeWarrior package.

Just to add a little confusion to the nomenclature surrounding Palm OS development, the term "GCC" can be read three different ways. In its original form, GCC stood for "GNU C Compiler." Open source developers, in response to adding Java and other language compilers to the GNU development tools, changed the GCC acronym to stand for "GNU Compiler Collection." The Palm development community, looking for a shorter name than "GNU PRC-Tools," often refers to the entire Palm OS development suite as GCC. (Originally, the package was called the "GNU Palm SDK," which you might also see in older Web sites or mailing list discussions about these tools.) Most of the time, if you are reading material in mailing lists, Web sites, and newsgroups devoted to Palm development, GCC specifically refers to the GNU PRC-Tools, not the generic GNU development tools. In this book, "GCC" is interchangeable with "GNU PRC-Tools." When talking about the original, generic GNU C compiler, I will use the lowercase "gcc" instead.

Both the Unix and Windows versions of the PRC-Tools are available on Palm Computing's developer Web site, at `http://www.palmos.com/dev/tech/tools/gcc`. The PRC-Tools are distributed as both GNU/Linux and Windows binaries, and also as source code.

Besides the PRC-Tools package, you will also need PilRC, a resource compiler that transforms text descriptions of user interface and other resources into the correct binary format expected by the Palm OS. PilRC also comes with PilrcUI, a program that provides a visual preview of the resources described in a PilRC source file. You may download PilRC from `http://www.ardiri.com/index.cfm?redir=palm&cat=pilrc`.

Both Windows and Unix versions of the GNU PRC-Tools are included on the CD-ROM accompanying this book.

Compiling and Linking with the PRC-Tools

Compiling and linking a Palm OS application requires the use of a number of different command line tools:

- **m68k-palmos-gcc.** This is the actual C/C++ compiler that turns source code into Motorola 68000 binary code.
- **m68k-palmos-obj-res.** This utility breaks the single binary file produced by m68k-palmos-gcc into separate code resources that may be included in a `.prc` file.
- **PilRC.** Discussed later in Chapter 6, "Creating and Understanding Resources," PilRC makes binary resources from a text file, usually with an `.rcp` extension.
- **build-prc.** The build-prc tool combines all the resources created by the other three tools into a `.prc` file, suitable for running on the Palm OS.

Compiling with m68k-palmos-gcc

Since the m68k-palmos-gcc compiler is just a modified version of the generic gcc compiler, it has many, many options, most of which are not particularly relevant to Palm OS development. Table 3-1 lists some common options used in compiling a Palm OS application. Note that all options are case-sensitive.

Table 3-1
m68k-palmos-gcc Compiler Options

Option	*What It Does*
`-c`	Compiles the sources without linking. This option produces raw object code from the source without linking it together into a larger executable. A common approach to compiling multiple source files is to run them through the compiler once with the `-c` option to produce an object code file for each source code file, then run all the object code files through the compiler again without the `-c` option to link them all together.
`-o` *`file`*	Places output in the file specified by *`file`*. Without the -o option, the compiler puts object code in files named with `.o` extensions, and it produces a file named `a.out` when linking multiple object files into an executable.
`-O`*`n`*	Performs optimization of code when compiling. *`n`* represents an integer value from 0 to 3, with 0 meaning no optimization and 3 meaning as much optimization as possible. Level 2 is adequate for most applications, resulting in a good balance between speed of execution and size of the application.

Continued

Table 3-1 *(continued)*

Option	What It Does
`-g`	Adds debugging information to the program for debugging with GDB. Without the -g option, GDB cannot provide source-level debugging of an application. This option adds a small amount of code to the compiled executable.
`-S`	Stops the compiler right after compiling but before assembling the code into a binary form. With this option set, the compiler produces an assembler code file. By default, this file has the same name as the input file, but with an `.s` suffix.

File extensions of input files passed to m68k-palmos-gcc determine how the compiler compiles those files. The compiler treats files with a `.c` extension as standard C source files, and files with a `.cc`, `.cxx`, `.cpp`, or `.C` extension as C++ source. A file with an `.h` extension is a header file, which m68k-palmos-gcc does not compile.

A typical command line for compiling a single C source file to object code looks like this:

```
m68k-palmos-gcc -O2 -c hello.c -o hello.o
```

To link multiple object files into a single executable, use a command line like the following:

```
m68k-palmos-gcc -O2 *.o hello
```

Breaking the code apart with m68k-palmos-obj-res

Once m68k-palmos-gcc has diligently assembled all your source files into a nice lump of Motorola 68000 code, you must run m68k-palmos-obj-res to separate the code into individual resources that the Palm OS understands. The m68k-palmos-obj-res post-processor generates files of the form *typeXXXX.yourfile*`.grc`, where *type* represents the kind of resource (“code” or “data”, for example), *XXXX* is the resource ID of the resource contained in the file, and *yourfile* is the name of the code file you passed to m68k-palmos-obj-res.

For example, the following command line breaks the `hello` executable code file into individual system resources:

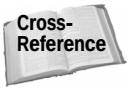
```
m68k-palmos-obj-res hello
```

The files resulting from running the previous command line are:

```
code0000.hello.grc
code0001.hello.grc
data0000.hello.grc
pref0000.hello.grc
rloc0000.hello.grc
```

Creating other resources with PilRC

The rest of an application's resources, primarily those that define its user interface, come from the PilRC tool. PilRC generates a file with a `.bin` extension for each resource it compiles.



See Chapter 6, "Creating and Understanding Resources," for details about using PilRC.

Assembling the application with build-prc

After you have created all the application's resources, the only thing you need to do to make a finished application is combine those resources into a `.prc` file with build-prc. The build-prc tool uses the following syntax:

```
build-prc <Destination.prc> <Database name> <Creator ID>
    <Resource 1> [Resource 2] [Resource 3] ...
```

The *Destination.prc* parameter specifies the file name of the completed `.prc` file. *Database name* is both the name of the application's database and the name that appears next to the application's icon in the launcher. *Creator ID* is the four-character unique identifier for your application. The build-prc tool treats everything else on its command line as the file name of a resource to include in the finished application.

The following example compiles all the `.grc` and `.bin` resources in a directory into a `.prc` file called `hello.prc`:

```
build-prc hello.prc "Hello" LFhe *.grc *.bin
```

Automating Builds with Make

In even a simple application, manually running the tools to create an application can easily become tedious. Fortunately, you can use the GNU *make* tool to automate the process. The Windows version of GCC installs make automatically; on Unix systems, make is a standard part of the GNU development tools. The make utility is powerful and complex. There are easily enough different options in make to warrant an entire book all by itself, so this book will cover only the basics.

For small projects on a Windows machine, it is also possible to automate builds by using a simple batch file. Batch files are not as flexible or powerful as the make utility, but they are quite sufficient for very simple applications. An example batch file is on this book's CD-ROM.

The make tool follows a series of directives you provide in a *makefile* to run the various tools necessary to produce a Palm OS application. Unless you have a particularly large and complex program, a single makefile will suffice for compiling most Palm OS programs. A makefile consists of *rules* and *commands*. Each rule tells make the name of a *target* and the *dependencies* required to create that target, and the

commands attached to that rule define the actual actions that make should take to create the target. For example, consider the following rule:

```
hello.o : hello.c
        m68k-palmos-gcc -c hello.c -o hello.o
```

This rule says that in order to create the file `hello.o`, the file `hello.c` must exist. If `hello.c` exists and it is newer than `hello.o`, make runs m68k-palmos-gcc with the specified options to create `hello.o`.

All commands in a makefile must begin with an actual tab character (ASCII 9, decimal). Some text editors replace tabs with spaces, which will cause make to fail. Be sure your editor inserts real tabs at the beginning of makefile commands.

When you run make, it looks for a file in the current directory called `makefile` or `Makefile` and begins to process it. The first rule in a makefile is the *default goal*, which make will attempt to process by default if you call make by itself, like this:

```
make
```

You can also call make with a specific target, which will process the appropriate rule in the makefile for that target. For example, the following command line tells make to update `hello.o` according to the rule in the preceding example:

```
make hello.o
```

If a dependency is also the target of another rule, make processes the other rule first to make sure the dependency is up-to-date. Expanding upon the previous example, take a look at the following:

```
hello : hello.o hi.o
        m68k-palmos-gcc hello.o hi.o -o hello

hello.o : hello.c
        m68k-palmos-gcc -c hello.c -o hello.o

hi.o : hi.c
        m68k-palmos-gcc -c hi.c -o hi.o

clean :
        rm hello hello.o hi.o
```

The first rule in this example links the object files `hello.o` and `hi.o` into the executable `hello`. If either of the `.o` files needs to be updated, make runs the commands associated with one or the other of the two last rules, depending on which object file needs to be updated.

The fourth rule in the example is a utility rule for cleaning the executable and its object code from the directory. Since the `clean` target is not a dependency of any other rule, make normally does not process its rule. To run the `clean` rule, invoke make with the following command line:

```
make clean
```

Make runs each command in a separate shell. When running a command, make always looks for the shell `/bin/sh`. If you install the Windows version of GCC using the default setup options, the GCC installer makes a copy of the bash shell called `sh.exe` and places it in `C:\bin`. This can cause a problem if you install GCC to another drive, such as `D:`, since the GCC installer isn't smart enough to change the drive to which it copies `sh.exe`. If you install GCC to a drive other than `C:`, be sure to make a `\bin` directory on that drive and copy `bash.exe` from the GCC `bin` directory to the newly created `\bin`, renaming `bash.exe` to `sh.exe`.

Using variables to simplify the makefile

In a large project containing many source files, having to type every single source and object file multiple times in the makefile becomes tedious and error-prone. For this reason, make allows you to define variables. The following example creates a variable named `OBJS` that contains all the object filenames in the project:

```
OBJS = hello.o hi.o
```

Now, with the variable `OBJS` defined, you may simplify the `hello` and `clean` rules from earlier examples like this:

```
hello : $(OBJS)
        m68k-palmos-gcc $(OBJS) -o hello
...
clean :
        rm hello ($OBJS)
```

Using pattern rules

You can simplify the makefile even further by using *pattern rules*. A pattern rule can use wildcard values to represent more than one file at a time. Consider the following:

```
%.o : %.c
        m68k-palmos-gcc -c $< -o $@
```

This takes the place of the `hello.o` and `hi.o` rules in previous examples. The percent sign (`%`) in this rule tells make that, in order to build a file with an `.o` extension, make should look for a dependency file with the same stem and a `.c` extension. The *automatic variables* `$<` and `$@` stand for the dependency file and target file, respectively.

On Windows, you may encounter problems running make because of the different processing that Windows performs on command-line arguments from the way most Unix shells handle a command line. Unix shells expand wildcards in a command-line argument before passing them to a program, like make, but Windows does not perform any expansion, or *globbing*, instead passing the wildcard characters to the program intact, which confuses make. If you have this problem in Windows, add the following environment variable to your `AUTOEXEC.BAT` file (in Windows 95 or 98) or your system environment settings (in Windows NT or 2000):

```
MAKE_MODE = UNIX
```

Putting it all together

Given the few simple rules explained above, it is possible to write much more complex makefiles than the examples earlier in this chapter. Listing 3-1 shows the makefile for the Hello World application from the next chapter. Notice that the makefile is built in such a way that with only a few simple changes to its variables, it can be reused to build an entirely different application.

A generic makefile, which you may use to compile any Palm OS application with only a few modifications, is on the CD-ROM accompanying this book.

Listing 3-1: The Hello World makefile

```
APP             = hello
ICONTEXT        = "Hello"
APPID           = LFhe
RCP             = $(APP).rcp
PRC             = $(APP).prc
SRC             = $(APP).c

CC              = m68k-palmos-gcc
PILRC           = pilrc
OBJRES          = m68k-palmos-obj-res
BUILDPRC        = build-prc

# Uncomment this if you want to build a debug version for GDB
# CFLAGS = -O0 -g
CFLAGS = -O2


all: $(PRC)

$(PRC): grc.stamp bin.stamp;
        $(BUILDPRC) $(PRC) $(ICONTEXT) $(APPID) *.grc *.bin
        ls -l *.prc
```

```
grc.stamp: $(APP)
        $(OBJRES) $(APP)
        touch $@

$(APP): $(SRC:.c=.o)
        $(CC) $(CFLAGS) $^ -o $@

bin.stamp: $(RCP)
        $(PILRC) $^ $(BINDIR)
        touch $@

%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@
#               touch $<
# Enable the previous line if you want to compile EVERY time.

depend dep:
        $(CC) -M $(SRC) > .dependencies

clean:
        rm -rf *.o $(APP) *.bin *.grc *.stamp *~

veryclean: clean
        rm -rf *.prc *.bak
```

Summary

In this chapter, you took a look at the two most popular C/C++ development environments for Palm OS programming. After reading this chapter, you should know the following:

- Metrowerks CodeWarrior is a commercial IDE for Palm OS development, officially supported by Palm Computing. CodeWarrior runs on Windows and the Mac OS.
- The GNU PRC-Tools, also known as GCC, is an open source, free alternative for Palm OS development that runs on both Unix and Windows systems.
- CodeWarrior stores information about an application and how it should be built in a *project*, which may contain multiple *targets* to direct the CodeWarrior tools to build the project in different ways.
- You control how CodeWarrior compiles and links an application by changing settings in the *project window* and *target settings dialog*.

- GCC uses a series of command line tools (m68k-palmos-gcc, m68k-palmos-obj-res, PilRC, and build-prc) to compile and link source code into a finished Palm OS executable file.
- The *make* tool is the easiest way to automate the steps required to produce a Palm OS application with the PRC-Tools.

✦ ✦ ✦

CHAPTER 4

Writing Your First Palm OS Application

In This Chapter

- Looking at the Hello World application
- Learning how the Palm OS starts your application
- Running through the main event loop
- Setting up callback event handlers
- Managing memory

The previous chapter showed you the tools for building applications for the Palm OS. Now it is time to look under the hood of a simple application and see what makes it work. In the long-standing tradition of computer language examples, this chapter will walk you through the code of a "Hello, World" program, introducing you to the general layout of a Palm OS application. Along the way, you will also learn about how the Palm OS starts your application, how the application responds to events, how to handle callback functions in the gcc compiler, and how to properly manage memory.

Looking at the Hello World User Interface

Before delving into the code in the Hello World application, a quick description of what the program looks like and what it does is in order. Because user interaction is such an integral part of the Palm OS, this example program does a little more than simply print an impersonal "Hello World" on the screen. Hello World has a text field for the user's name; Figure 4-1 shows the application's main form after a little text has been entered into the field.

Figure 4-1: The Hello World application

Buttons at the bottom of the form display alerts when tapped, customized with a string entered by the user in the name field. The program also has an about box to display a little information about the program; the about box may be accessed from the application's menus. Figure 4-2 shows the alerts that appear when the user taps the Say Hello or Say Goodbye buttons, or when the user selects the About Hello World menu item.

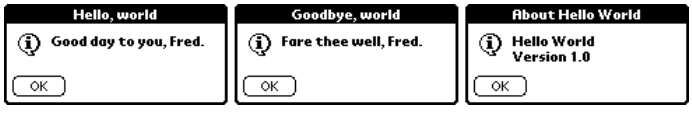


Figure 4-2: Alerts in the Hello World application

Hello World also implements the standard Palm Edit menu to provide text-editing commands for the form's name field. An Options menu offers an About Hello World item to display the application's about box. Figure 4-3 shows the menus in the Hello World application.

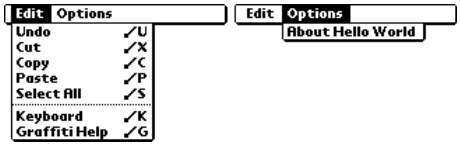
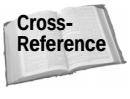


Figure 4-3: Menus in the Hello World application

The resources for the Hello World application's form, alerts, and menus are defined in separate files from the source code shown in this chapter. These files, and the processes used to generate them, are different between the CodeWarrior and GNU Palm SDK environments.



Creating resources is covered in detail in Chapter 6, "Creating and Understanding Resources."

Walking Through the Hello World Code

Now that you have some idea what the Hello World application is supposed to do, it is time to take a close look at how it works. The file `hello.c` contains the bulk of the code for the Hello World application. A complete listing of `hello.c` is at the end of this chapter in Listing 4-3. This section will deal with a small piece of the application code at a time.

The complete code for Hello World is also available on the CD-ROM attached to this book.

Including Header Files

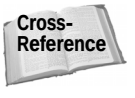
At the top of `hello.c` are the following `#include` directives:

```
#include <PalmOS.h>

#ifdef __GNUC__
#include "callback.h"
#endif

#include "helloRsc.h"
```

The `PalmOS.h` file contains further includes for most of the include files in the Palm OS. In the CodeWarrior environment, `PalmOS.h` includes a prebuilt header to assist in faster compilation. Because Hello World is a very simple application, the includes in `PalmOS.h` are more than sufficient for everything the application must accomplish.

Prior to Palm OS 3.5, the main include file for the Palm OS SDK was called `Pilot.h`. If you are compiling an application using headers that are older than the 3.5 SDK, you should use `Pilot.h` instead of `PalmOS.h`. See the sidebar titled "Moving Applications to Palm OS 3.5" later in this chapter for more details on the differences between 3.5 and earlier versions of the Palm OS headers.

The file `callback.h` contains macro definitions needed by older versions of the GNU PRC-Tools to properly compile callback functions. The `#ifdef` prevents the GCC-specific callback code from compiling under CodeWarrior.



More details on the macros in `callback.h` and their use are included later in this chapter, under "Using Callback Functions with GCC."

Hello World also includes the file `helloRsc.h`, which defines resource constants used throughout the application to identify menus, controls, alerts, and other resources. For example, the constant `MainNameField` identifies the text field in Hello World's main form.

In the CodeWarrior environment, the Constructor tool generates `helloRsc.h` automatically. Listing 4-1 shows an example of a resource constant file generated by Constructor. In the GNU PRC-Tools, you normally create the resource constant file yourself. Listing 4-2 shows how this handmade file looks.

Listing 4-1: **The helloRsc.h file, as generated by Constructor**

```
// Header generated by Constructor for Pilot 1.0.2
//
// Generated at 3:26:43 PM on Monday, 23 August, 1999
//
// Generated for file: D:\Bible\code\ch04\cw\hello.rsrc
//
// THIS IS AN AUTOMATICALLY GENERATED HEADER FILE FROM
// CONSTRUCTOR FOR PALMPILOT;
// - DO NOT EDIT - CHANGES MADE TO THIS FILE WILL BE LOST
//
// Pilot App Name:         "Hello"
//
// Pilot App Version:      "1.0"



// Resource: tFRM 1000
#define MainForm                                1000
#define MainHelloButton                         1002
#define MainGoodbyeButton                       1003
#define MainNameField                           1001
#define MainUnnamed1099Label                    1099


// Resource: Talt 1000
#define HelloAlert                              1000
#define HelloOK                                 0

// Resource: Talt 1100
#define GoodbyeAlert                            1100
#define GoodbyeOK                               0

// Resource: Talt 1200
#define AboutAlert                              1200
#define AboutOK                                 0

// Resource: MBAR 1000
#define MainMenuBar                             1000

// Resource: MENU 1000
#define MainEditMenu                            1000
#define MainEditUndo                            1000
#define MainEditCut                             1001
#define MainEditCopy                            1002
#define MainEditPaste                           1003
#define MainEditSelectAll                       1004
#define MainEditKeyboard                        1006
#define MainEditGraffitiHelp                    1007
```

```
// Resource: MENU 1100
#define MainOptionsMenu                         1100
#define MainOptionsAboutHelloWorld              1100
```

Listing 4-2: **The helloRsc.h file, made by hand for the GNU environment**

```
// Main form
#define MainForm              1000
#define MainNameField         1001
#define MainHelloButton       1002
#define MainGoodbyeButton     1003

// Menubar
#define MainMenuBar           1000

// Menu commands
#define MainEditUndo          1000
#define MainEditCut           1001
#define MainEditCopy          1002
#define MainEditPaste         1003
#define MainEditSelectAll     1004
#define MainEditKeyboard      1006
#define MainEditGraffitiHelp  1007

#define MainOptionsAboutHelloWorld   1100

// Alerts
#define HelloAlert            1000
#define GoodbyeAlert          1100
#define AboutAlert            1200
```

Entering the Application

The first code the Palm OS executes in your application is the **PilotMain** routine, which looks like this in Hello World:

```
UInt32 PilotMain(UInt16 launchCode, MemPtr cmdPBP,
                 UInt16 launchFlags)
{
    Err  err;


    switch (launchCode) {
        case sysAppLaunchCmdNormalLaunch:
            if ((err = StartApplication()) == 0) {
                EventLoop();
```

```
                StopApplication();
            }
            break;

        default:
            break;
    }

    return err;
}
```

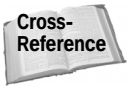
The first parameter to **PilotMain** is a launch code telling your application how to start itself. In a normal application launch, the Palm OS passes the constant `sysAppLaunchCmdNormalLaunch` to the **PilotMain** routine.

During a normal launch, the `cmdPBP` and `launchFlags` parameters are not used. They contain extra parameters and flags used when the operating system calls the application with a different launch code. For example, the `sysAppLaunchCmdFind` launch code, sent by the OS during a global find, passes a pointer to the text to search for in the `cmdPBP` parameter. Table 4-1 lists some common launch codes and what they indicate.

Table 4-1
Selected Palm OS Launch Codes

Launch Code	*Description*
`sysAppLaunchCmdAddRecord`	Adds a record to the application's database
`sysAppLaunchCmdDisplayAlarm`	Tells the application to display a specified alarm dialog or perform other lengthy alarm-related actions
`sysAppLaunchCmdFind`	Finds a text string somewhere in the application's stored data
`sysAppLaunchCmdGoto`	Goes to a specific record in the application's database
`sysAppLaunchCmdNormal`	Launches the application normally
`sysAppLaunchCmdSystemReset`	Allows the application to respond to a system reset

The Palm OS supports many other launch codes. Fortunately, your application does not need to respond to all of them. If a particular launch code is inappropriate for your application, simply leave that code out of your application's **PilotMain** routine.



A complete list of launch codes is available in Appendix A, "Palm OS API Reference."

PilotMain in the Hello World application deals only with the `sysAppLaunchCmdFind` launch code. When it receives this code, it passes execution to **StartApplication**.

Starting the Application

The **StartApplication** routine in Hello World is listed below:

```
static Err StartApplication(void)
{
    FrmGotoForm(MainForm);
    return 0;
}
```

StartApplication is where more complex programs would perform database initialization and retrieval of user preferences prior to running the rest of the application. Because Hello World does not save any data between the times the user runs it, its **StartApplication** routine is quite brief. In Hello World, this routine has only one responsibility: starting up the main form that contains the application's interface. The **FrmGotoForm** function tells the currently open form to close, and then puts `frmLoadEvent` and `frmOpenEvent` events into the event queue, signifying the specified form to load itself and open.

Closing the Application

Skipping ahead a bit, the **StopApplication** routine runs when the **EventLoop** routine has exited and the application is shutting down. Here is **StopApplication** from Hello World:

```
static void StopApplication(void)
{
}
```

Because Hello World does not need to perform any cleanup before closing down, its **StopApplication** routine is empty. Normally, this routine contains code to close the application's database, save user preferences, and perform other tasks prior to the program ending execution.

Handling Events

After **PilotMain** calls **StartApplication** to initialize the program, execution passes to the **EventLoop** routine:

```
static void EventLoop(void)
{
    EventType  event;
```

```
    UInt16     error;

    do {
        EvtGetEvent(&event, evtWaitForever);
        if (! SysHandleEvent(&event))
            if (! MenuHandleEvent(0, &event, &error))
                if (! ApplicationHandleEvent(&event))
                    FrmDispatchEvent(&event);
    } while (event.eType != appStopEvent);
}
```

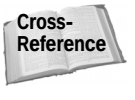
The event loop is responsible for processing events received by the application. Incoming events enter the *event queue*, which **EventLoop** processes one event at a time. **EventLoop** grabs events from the queue with the **EvtGetEvent** function, and then dispatches those events to the event handling routines. Each of the four event handlers gets an opportunity to process the event in turn, in this order:

1. **SysHandleEvent** handles system events.
2. **MenuHandleEvent** takes care of menu events.
3. **ApplicationHandleEvent** loads form resources and sets up form-specific event handlers.
4. **FrmDispatchEvent** passes the event to the application's own event handler, or lets the operating system perform default actions for that event.

Once the event has been handled, the event loop starts over again, retrieving the next event from the event queue and repeating the process. The **EventLoop** routine continues until it pulls an `appStopEvent` from the queue, at which point it stops and passes execution back to **PilotMain**.

The **EvtGetEvent** function warrants further discussion. Its first parameter merely provides an address at which the function should put the next event it retrieves. The second parameter to **EvtGetEvent** is a time-out value in *ticks*, or hundredths of a second. If no event enters the queue for before the time-out value elapses, **EvtGetEvent** returns the value `nilEvent` in the first parameter.

Most applications should pass the constant `evtWaitForever` (equal to -1) as the second parameter of **EvtGetEvent**, which puts the system into doze mode to conserve power until another event enters the queue. Time-out values greater than or equal to zero are primarily useful in applications that must animate screen images, such as games.



Animation is detailed in Chapter 10, "Programming System Elements."

Processing system events in SysHandleEvent

The first event handler that receives an event from the event loop is **SysHandle Event**, which gives the OS an opportunity to handle important system events. The system handles such things as Graffiti input, hardware button presses, and taps on the silk-screened buttons. **SysHandleEvent** also takes care of low battery notifications, the global find function, and other system-wide events that may interrupt whatever the application is currently doing.

Depending on the event, **SysHandleEvent** often puts more events back into the queue. For example, when the user enters strokes into the Graffiti area, the system interprets the resulting Graffiti events and places corresponding key events into the queue. The event loop eventually pulls these key events out of the queue and processes them.

If it handles the event completely, **SysHandleEvent** returns `true`. The event loop then calls **EvtGetEvent** to process the next event in the queue.

Handling menu events in MenuHandleEvent

If the system was not interested in handling an event, **MenuHandleEvent** gets the next crack at it. **MenuHandleEvent** cares about only two kinds of events:

- Any taps from the user that invoke a menu, in which case **MenuHandleEvent** displays the appropriate menu.
- Taps inside a menu that activate a menu item, which cause **MenuHandle Event** to erase the menu from the screen and put events corresponding to the selected command into the event queue.

Like **SysHandleEvent**, **MenuHandleEvent** also returns `true` if it completely processes an event.

Preparing forms in ApplicationHandleEvent

Events that make it this far into the event loop are of potential interest to the application itself. **ApplicationHandleEvent** is a function you must write yourself, and its only purpose is to handle the `frmLoadEvent`. In the **ApplicationHandleEvent** function, your program loads and activates form resources. This function is also where the application sets up a callback function to serve as an event handler for the current active form.

ApplicationHandleEvent and callback event handlers are covered in more detail later in this chapter.

Dealing with form events in FrmDispatchEvent

The **FrmDispatchEvent** function is like a miniature event loop within the more complicated **EventLoop**. **FrmDispatchEvent** first passes the event to the active form's event handler, which was set up previously in **ApplicationHandleEvent**. Because Hello World has only one form, **FrmDispatchEvent** will pass events to the **MainFormHandleEvent** callback function. If the form event handler fully processes the event, it returns `true`, causing **FrmDispatchEvent** to return execution to the event loop.

FrmDispatchEvent passes events not handled by the application to **FrmHandleEvent**, a function that lets the system perform default processing of the event. This processing usually involves standard user interface actions, such as highlighting a button tapped by the user. In any case, all events not previously handled in the event loop meet their final resting place in the **FrmHandleEvent** function, which does not return any value.

Setting Up Forms

Events not handled by **SysHandleEvent** or **MenuHandleEvent** make their way to **ApplicationHandleEvent**. Unlike the first two event handlers, **ApplicationHandleEvent** is not part of the Palm OS. You must provide this function yourself, because its contents will vary from program to program. The **ApplicationHandleEvent** function from Hello World is listed as follows:

```
static Boolean ApplicationHandleEvent(EventPtr event)
{
    FormPtr  form;
    UInt16    formID;
    Boolean   handled = false;

    if (event->eType == frmLoadEvent) {
        formID = event->data.frmLoad.formID;
        form = FrmInitForm(formID);
        FrmSetActiveForm(form);

        switch (formID) {
            case MainForm:
                FrmSetEventHandler(form, MainFormHandleEvent);
                break;
        }
        handled = true;
    }

    return handled;
}
```

ApplicationHandleEvent is responsible for two things:

- ✦ Initializing form resources
- ✦ Setting callback event handlers for forms

The **ApplicationHandleEvent** function accomplishes the first of these two goals by calling two system functions, **FrmInitForm** and **FrmSetActiveForm**. **FrmInitForm** loads the form resource into memory and initializes its data structure, returning a pointer to that form. Then **FrmSetActiveForm** takes the pointer to the newly initialized form and makes that form into the *active form*. Only one form may be active at a time in the Palm OS. The currently active form receives all input from the user, both pen and key events, and all drawing occurs within the active form.

Now that the form has been initialized and activated, it needs an event handler so it knows what to do with all the input it will soon be receiving. **ApplicationHandleEvent** sets the form's event handler with the **FrmSetEventHandler** function. Each form's event handler is a separate callback function that you provide. Once this event handler is set, the OS passes all events that are not handled by the system or the menu manager to this form event handler.

In Hello World, there is only one form, and its event handler is **MainFormHandle Event**. In more complex applications, **ApplicationHandleEvent** sets a different event handler for each form in the application.

Responding to Form Events

The **MainFormHandleEvent** function in Hello World processes input to the application's main form. **MainFormHandleEvent** looks like this:

```
static Boolean MainFormHandleEvent(EventPtr event)
{
    Boolean  handled = false;

#ifdef __GNUC__
    CALLBACK_PROLOGUE;
#endif

    switch (event->eType) {
        case frmOpenEvent:
        {
            FormType  *form = FrmGetActiveForm();

            FrmDrawForm(form);
            FrmSetFocus(form, FrmGetObjectIndex(form,
                MainNameField));
            handled = true;
        }
            break;
```

```
            case ctlSelectEvent:
                switch (event->data.ctlSelect.controlID) {
                    case MainHelloButton:
                        SaySomething(HelloAlert);
                        handled = true;
                        break;

                    case MainGoodbyeButton:
                        SaySomething(GoodbyeAlert);
                        handled = true;
                        break;

                    default:
                        break;
                }
                break;

            case menuEvent:
                handled =
                    MainMenuHandleEvent(event->data.menu.itemID);
                break;

            default:
                break;
        }

    #ifdef __GNUC__
        CALLBACK_EPILOGUE;
    #endif

        return handled;
    }
```

This function receives events from **FrmDispatchEvent** in the event loop. Hello World actually begins doing some real work in this function instead of relying on the system to take care of everything.

MainFormHandleEvent handles the `frmOpenEvent` by drawing the form on the screen with the **FrmDrawForm** function. Because there is a text field on the screen, it is also a good idea to put the focus in that field so the user may immediately begin entering text without first tapping in the field. The **FrmSetFocus** function performs this task.

If the user taps and releases either of the buttons on the main form, a `ctlSelect Event` enters the queue and eventually makes its way to **MainFormHandleEvent**. Depending on which button was tapped, **MainFormHandleEvent** displays one of two alerts using the **SaySomething** function, which is explained later in this chapter.

The **MainFormHandleEvent** function can also handle menu events. To keep the function small and easy to read, though, **MainFormHandleEvent** defers menu handling to **MainMenuHandleEvent**, another function provided by the application instead of the operating system.

You may be wondering what all the `#ifdef __GNUC__` stuff is for in this function. The macros in these conditional compilation statements are required for callback functions to properly compile using GCC, because the GNU compiler handles callbacks a little differently from CodeWarrior. For more information, see the sidebar "Using Callback Functions with GCC."

Using Callback Functions with GCC

The Palm OS and early versions of the GNU PRC-Tools (0.5.0 and earlier) make different assumptions about how some registers will be used during the life of an application. GCC sets the A4 register, through which it accesses an application's global variables, and the compiler expects that the contents of A4 will remain untouched. However, the contents of A4 are not sacred to the Palm OS, and if the system calls a GCC-compiled function as a callback, the A4 register may be altered, causing the application to crash when the callback tries to access global variables.

To work around this problem, Ian Goldberg, one of the contributors to the GNU PRC-Tools, wrote a pair of macros that set the A4 register when starting a callback function, and then restore the register when the callback exits. These macros reside in a convenient header file, `callback.h`, which you can `#include` in your own application:

```
#ifndef __CALLBACK_H__
#define __CALLBACK_H__

/* This is a workaround for a bug in the current version of gcc:

   gcc assumes that no one will touch %a4 after it is set up in
   crt0.o.  This isn't true if a function is called as a callback
   by something that wasn't compiled by gcc (like
   FrmCloseAllForms()).  It may also not be true if it is used as
   a callback by something in a different shared library.

   We really want a function attribute "callback" which will
   insert this prologue and epilogue automatically.

      - Ian */

register void *reg_a4 asm("%a4");
```

Continued

Continued

```
#define CALLBACK_PROLOGUE \
   void *save_a4 = reg_a4; \
   asm("move.l %%a5,%%a4; sub.l #edata,%%a4" : :);
#define CALLBACK_EPILOGUE reg_a4 = save_a4;

#endif
```

When writing a callback function, you must put the `CALLBACK_PROLOGUE` macro at the beginning of the function, just after its variable declaration. Just before the function returns, insert the `CALLBACK_EPILOGUE` macro into your code. The MainFormHandleEvent function from Hello World shows proper placement of the macros.

Be careful not to access global variables in your callback functions before the `CALLBACK_PROLOGUE` macro. A common mistake is to declare a variable and initialize it with the value of a global variable at the top of the code. The following example will cause an application to crash:

```
static int BadCallback ()
{
    int localVariable = gGlobalVariable;  // error
    CALLBACK_PROLOGUE
    ...
}
```

Instead, set the value of the variable after `CALLBACK_PROLOGUE` has been called:

```
static int GoodCallback()
{
    int localVariable;
    CALLBACK_PROLOGUE
    localVariable = gGlobalVariable;
    ...
}
```

Likewise, be sure your code runs `CALLBACK_EPILOGUE` before returning. The easiest way to ensure this is to make sure your function returns at the bottom. If you must have your function return somewhere in the middle, be sure to put another occurrence of `CALLBACK_EPILOGUE` into your code:

```
static Boolean ReturnFromMiddle(Boolean foo)
{
    Boolean  handled = false;
    CALLBACK_PROLOGUE

    if (foo) {
        handled = true;
```

```
            CALLBACK_EPILOGUE  // epilogue macro required here...
            return handled;
        }

        CALLBACK_EPILOGUE      // ...and also here
        return handled;
    }
```

At the time I'm writing this, PRC-Tools 2.0 has successfully solved the callback problem. The 2.0 version is based on the newer EGCS (Experimental GNU Compiler System), and it no longer suffers from the callback problem exhibited by version 0.5.0 or earlier of the PRC-Tools. If you are using the GNU PRC-Tools 2.0 or later, you can safely leave the `CALLBACK_PROLOGUE` and `CALLBACK_EPILOGUE` macros out of your code, and callback functions will still work properly.

Handling Menu Events

MainFormHandleEvent passes menu events to another function, **MainMenu HandleEvent**, which is shown in the following example:

```
static Boolean MainMenuHandleEvent(UInt16 menuID)
{
    Boolean    handled = false;
    FormType   *form;
    FieldType  *field;


    form = FrmGetActiveForm();
    field = FrmGetObjectPtr(form,
        FrmGetObjectIndex(form, MainNameField));

    switch (menuID) {
        case MainEditUndo:
            FldUndo(field);
            handled = true;
            break;
        case MainEditCut:
            FldCut(field);
            handled = true;
            break;
        case MainEditCopy:
            FldCopy(field);
            handled = true;
            break;
        case MainEditPaste:
            FldPaste(field);
```

```
            handled = true;
            break;
        case MainEditSelectAll:
            FldSetSelection(field, 0,
                            FldGetTextLength(field));
            handled = true;
            break;

        case MainEditKeyboard:
            SysKeyboardDialog(kbdDefault);
            handled = true;
            break;

        case MainEditGraffitiHelp:
            SysGraffitiReferenceDialog(referenceDefault);
            handled = true;
            break;

        case MainOptionsAboutHelloWorld:
            FrmAlert(AboutAlert);
            handled = true;
            break;

        default:
            break;
    }

    return handled;
}
```

The Edit menu commands handled by **MainMenuHandleEvent** are standard menu items, which you should implement in all your forms and dialogs that contain text fields. Because the Palm OS provides simple system functions to deal with the standard field actions, there is no excuse for not including them in your application. There are few things more frustrating than a text field that will not allow copying and pasting, and users may find your application difficult to use if Graffiti help, supplied by the **SysGraffitiReferenceDialog** function, is not available.

The SysKeyboardDialog and SysGraffitiReferenceDialog functions are indigenous to Palm OS version 2.0 and later. The system-wide Graffiti reference does not exist in version 1.0, and the version 1.0 function for calling up the keyboard dialog is SysKeyboardDialogV10, which has no parameters. Keep this in mind when writing applications that must run on Palm OS 1.0.

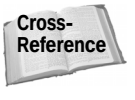
The only other menu option handled by **MainMenuHandleEvent** is the application's about box, which is an alert resource. The menu-handling function displays the about alert with the **FrmAlert** function.

Displaying Alerts and Using the Text Field

MainFormHandleEvent calls the **SaySomething** function when the user taps one of the main form's two buttons. **SaySomething** is listed in the following example:

```
static void SaySomething(UInt16 alertID)
{
    FormType   *form = FrmGetActiveForm();
    FieldType  *field;
    MemHandle  h;


    field = FrmGetObjectPtr(form, FrmGetObjectIndex(form,
                            MainNameField));
    if (FldGetTextLength(field) > 0) {
        FldCompactText(field);
        h = FldGetTextHandle(field);
        if (h) {
            Char  *s;

            s = MemHandleLock((void *)h);
            FrmCustomAlert(alertID, s, NULL, NULL);
            MemHandleUnlock((void *)h);
        }
    } else {
        // No text in field, so display a "whoever you are"
        // dialog.
        FrmCustomAlert(alertID, "whoever you are", NULL, NULL);
    }
}
```

SaySomething takes a single argument, the resource ID of an alert that it should display. Instead of popping up a static alert box with **FrmAlert**, as **MainMenuHandle Event** did for the application's about box, **SaySomething** customizes the alerts it displays by using the **FrmCustomAlert** function. **FrmCustomAlert** replaces up to three special characters in the alert resource it displays with strings supplied in three of its arguments. Hello World needs to fill in only one piece of custom text in each alert, so **SaySomething** passes `NULL` for the third and fourth arguments to **FrmCustomAlert**.



More information on displaying custom alerts is in Chapter 7, "Building Forms."

In order for **SaySomething** to grab the string entered in the form's text field, the function needs a pointer to that field. **SaySomething** accomplishes this with the **FrmGetObjectIndex** and **FrmGetObjectPtr** functions. **FrmGetObjectIndex** takes the resource ID of the text field and returns its *object index*. Every object on a form has a unique object index, starting at zero. **FrmGetObjectPtr** takes the object index obtained by **FrmGetObjectIndex** and returns a pointer to the field.

The construct `FrmGetObjectPtr(form, FrmGetObjectIndex(form, objectID))` is very common in Palm OS programming, and you will find yourself using it throughout your applications. If you wish to save some typing while writing your code, a function similar to the following may come in handy:

```
static VoidPtr GetObjectPtr (Word objectID)
{
    FormPtr  form;

    form = FrmGetActiveForm();
    return (FrmGetObjectPtr(form,
        FrmGetObjectIndex(form, objectID)));
}
```

Now that **SaySomething** has a pointer to the field, the function checks to see if the field contains any text by calling the system function **FldGetTextLength**. The **FldGetTextLength** function returns the number of bytes of text contained in a field. If there is no text in the field, **SaySomething** fills in an appropriate generic name, `"whoever you are"`, so the alert will have something to display to the user.

Because the text in a field may change size dynamically as the user edits it, the Palm OS uses a moveable chunk of memory to store the field's text. The system uses *handles* to keep track of a moveable chunk's location instead of pointers, which indicate specific, immobile chunks of memory. Allowing chunks of memory to be mobile ensures that the OS can relocate memory as it sees fit to make room for more memory allocations, a very useful feature indeed on a platform with so little dynamic memory available.

Before retrieving a handle to the text in the field, **SaySomething** calls **FldCompactText** to release any unused space in the memory chunk containing the field's text. As a user enters text in a field, the system increases the size of the field's text chunk several bytes at a time to avoid having to expand the field as each and every new character is entered. This may result in a few empty bytes, so **FldCompactText** can be used to shrink the text chunk to take up as little space as possible.

After the field's text chunk has been compacted, **SaySomething** retrieves a handle to the memory holding the field's text by calling **FldGetTextHandle**. Once the field's text handle is retrieved, **SaySomething** locks the handle in place with **MemHandleLock**. Without locking the chunk containing the field's text, the operating system could very well move the chunk before the application is done reading or modifying it, so it is important to lock handles before using their contents. **MemHandleLock** returns a pointer to the locked memory chunk, which **SaySomething** uses to retrieve the field's contents and pass them to **FrmCustomAlert**. Then **SaySomething** calls **MemHandle Unlock** to free the handle, allowing the operating system to move that chunk again.

If you only need to read data from a text field, you can also directly retrieve a pointer to the field's contents with the FldGetTextPtr function. However, you should only use FldGetTextPtr if you do not need to alter the contents of the field, since it treats the text field's memory chunk as a fixed pointer, rather than a moveable handle.

SaySomething also fills in an appropriate generic name, "whoever you are", when the user has not entered any text into the field. Notice that this occurs in two different places in the function. The code that detects an empty string for the contents of the field, `if (*s != '\0')`, is the obvious case; a string containing only a trailing null character must be empty. However, the OS will not allocate a handle for the text field until the user enters some text. If the form has just been opened (such as when the application first starts), no handle exists for the field yet, thus necessitating the use of a second "whoever you are" in the code to take care of this situation.

Using Memory in the Palm OS

One of the most common errors in programming is accidentally writing to the wrong memory address. In a desktop computer, this can cause some spectacular application crashes, but it usually won't affect permanently stored data, because that resides on a separate storage device from the system's main memory. Because Palm OS devices use the same RAM for storage that they use for dynamic memory, a more stringent system of memory management is necessary to prevent badly written applications from corrupting permanently stored data.

The memory manager in the Palm OS provides just such a facility. Allocation and manipulation of memory cannot be accomplished on a Palm OS device without the appropriate Palm OS memory APIs. You can put away **malloc** and **free**, because you won't need them for Palm OS programming.

Most memory manipulation functions in the Palm OS fall into two categories: *pointer* functions and *handle* functions. Palm OS uses the functions **MemPtrNew** and **MemPtrFree** instead of the C standard library calls **malloc** and **free** to allocate and deallocate pointers. Other than this syntactic difference, most pointer use should be about what you expect from a standard C application. However, pointers use only unmovable chunks of memory, which doesn't allow them to take advantage of the operating system's ability to efficiently manage the small amount of dynamic RAM available.

Handle functions allow you to manipulate chunks of memory that may be moved by the operating system. Whenever the operating system needs to allocate more memory, it will move handles around until there is enough free contiguous memory for the new allocation. This scheme allows for much more effective use of the limited memory on a Palm OS device.

You can allocate a new memory handle with the **MemHandleNew** function. This function takes an argument specifying the size of the memory chunk to allocate, in bytes. The following code allocates a new, 32-byte handle:

```
VoidHand newHandle = MemHandleNew(32);
```

Because the operating system may freely move the memory connected to a handle, you must first lock a handle with **MemHandleLock** before you can read data from

or write data to it. Once you have finished using the handle, unlock it with **MemHandleUnlock** so the operating system can once again move that handle's chunk around. You've already seen **MemHandleLock** and **MemHandleUnlock** in action in the **SaySomething** function of Hello World.

The Palm OS keeps track of how many times you have locked a particular chunk of memory with a *lock count*. Every time you call **MemHandleLock** on a particular handle, the operating system increments the lock count on that handle's memory chunk. Likewise, each call to **MemHandleUnlock** decrements the lock count by one. Only when the lock count reaches zero is the chunk actually unlocked and free to be moved by the OS again.

Because the operating system cannot move a locked chunk of memory, you must be sure to unlock a chunk as soon as possible after it has been locked. Otherwise, memory fragmentation may occur, possibly preventing further memory allocation because the Palm OS cannot locate a large enough area of contiguous memory.

Each chunk of memory in the Palm OS contains a `lock:owner` byte. The high nibble of this byte contains the lock count for that particular chunk of memory. A value of 15 in the lock count field indicates an unmovable chunk. Because of this, a chunk may be locked only 14 times; the fifteenth call to **MemHandleLock** will result in an error.

The other half of the `lock:owner` byte stores the owner ID of the application that owns this particular chunk of memory. When an application exits, the operating system automatically deallocates all chunks of memory with that application's owner ID. This garbage collection prevents a careless programmer from creating a memory leak, which would be a serious problem on a platform with, at most, 96KB of dynamic RAM.

Relying on the Palm OS to clean up your memory for you is sloppy programming. Be sure to explicitly free memory that you have allocated.

It may be more convenient to unlock a chunk of memory using the **MemPtrUnlock** function. Instead of having to pass the chunk's handle around between different routines in your application, **MemPtrUnlock** will unlock a chunk given a pointer to that chunk, rather than a handle.

When you are through using a chunk of memory, call **MemHandleFree** to dispose of a moveable chunk, or **MemPtrFree** to deallocate an unmovable chunk:

```
MemHandleFree(someHandle);
MemPtrFree(somePointer);
```

You can retrieve the size of a particular chunk with the **MemHandleSize** and **MemPtrSize** functions:

```
ULong sizeOfHandle = MemHandleSize(someHandle);
ULong sizeOfPointer = MemPtrSize(somePointer);
```

Resizing a moveable chunk is also possible, using the **MemHandleResize** function:

```
switch (MemHandleResize(someHandle, newSize)) {
    case memErrInvalidParam:
        // Invalid parameter passed
    case memErrNotEnoughSpace:
        // Not enough free space in the current heap
        // to grow the chunk
    case memErrChunkLocked:
        // The chunk passed to MemHandleResize is locked
    case 0:
        // Resizing was successful
}
```

MemHandleResize works only on an unlocked chunk of memory. It first looks for more memory immediately following the current chunk, so the chunk will not have to be moved. If there is not enough free space directly following the chunk, the operating system will move the chunk to a new location that does contain enough contiguous space. There is also a **MemPtrResize** function that will work on a locked chunk of memory, but only if there is enough free space available right after that chunk. Both **MemHandleResize** and **MemPtrResize** always succeed when shrinking the size of the chunk.

The Palm OS also has a few utility functions for manipulating the contents of memory. **MemMove** will move a certain number of bytes from one location in memory to another, handling overlapping ranges as appropriate:

```
MemMove(void *destination, void *source,
    ULong numberOfBytes);
```

MemSet will set a certain number of bytes at a particular memory location to a specific value:

```
MemSet(void *changeThis, ULong numberOfBytes, Byte value);
```

Finally, **MemCmp** can be used to compare the values of two different memory locations:

```
Int difference;
difference = MemCmp(void *a, void *b, ULong numberOfBytes);
if (difference > 0) {
    // a is greater than b
} else if (difference < 0) {
    // b is greater than a
} else {
    // The two blocks are the same
}
```

Moving Applications to Palm OS 3.5

With the release of Palm OS version 3.5, Palm Computing made significant changes to the header files in the Palm OS SDK. For starters, prior to the 3.5 headers, the basic file to include in a Palm OS application was `Pilot.h`; in 3.5, this file is called `PalmOS.h`.

Within the header files themselves, there are different type definitions, intended by Palm Computing to improve clarity and consistency. For example, the 3.5 headers use `UInt32` instead of `DWord` and `Int16` instead of `Int` to better indicate the size of particular data types, and whether or not those types are signed. Also, early versions of the headers made typedef declarations of the form `FooPtr`, where `FooPtr` is simply a pointer to a `FooType` structure. Palm Computing has retired this convention in favor of using the standard C convention of `FooType *` to refer to a pointer to a `FooType` structure.

If you are working with code originally written using headers earlier than those included with Palm OS SDK 3.5, you will need to search for the older-style data types and replace them with their new equivalents. There is also a file called `PalmCompatibility.h` in the 3.5 SDK, which maps the older-style data types to their new names. You can include `PalmCompatibility.h` in an older project to help it deal with compilation under the 3.5 headers, but you are probably better off in the long run to bite the bullet and search and replace the data types yourself. Your code will be easier to read and less likely to break in the future if you fix it properly instead of patching it with `PalmCompatibility.h`.

The `PalmCompatibility.h` file does make a good reference for what needs to be changed in your source files, though. The following section of `PalmCompatibility.h` points out the key differences in data types between the Palm OS 3.5 headers and earlier versions of the SDK:

```
typedef Int8    SByte;
typedef UInt8   Byte;

typedef Int16   SWord;
typedef UInt16  Word;

typedef Int32   SDWord;
typedef UInt32  DWord;


// Logical data types
typedef Int8    SChar;
typedef UInt8   UChar;

typedef Int16   Short;
typedef UInt16  UShort;

typedef Int16   Int;
typedef UInt16  UInt;
```

```
typedef Int32   Long;
typedef UInt32  ULong;


// Pointer Types
typedef MemPtr     VoidPtr;
typedef MemHandle  VoidHand;

typedef MemPtr     Ptr;
typedef MemHandle  Handle;


// Because "const BytePtr" means "const pointer to Byte" rather
// than "pointer to const Byte", all these XXXXPtr types are
// deprecated: you're better off just using "Byte *" and so on.
// (Even better, use "UInt8 *"!)

typedef SByte*   SBytePtr;
typedef Byte*    BytePtr;

typedef SWord*   SWordPtr;
typedef Word*    WordPtr;
typedef UInt16*  UInt16Ptr;

typedef SDWord*  SDWordPtr;
typedef DWord*   DWordPtr;

// Logical data types
typedef Boolean*  BooleanPtr;

typedef Char*    CharPtr;
typedef SChar*   SCharPtr;
typedef UChar*   UCharPtr;

typedef WChar*   WCharPtr;

typedef Short*   ShortPtr;
typedef UShort*  UShortPtr;

typedef Int*     IntPtr;
typedef UInt*    UIntPtr;

typedef Long*    LongPtr;
typedef ULong*   ULongPtr;
```

Putting It All Together

The complete code listing for Hello World's `hello.c` file follows in Listing 4-3.

Listing 4-3: **The hello.c file from Hello World**

```
#include <PalmOS.h>

#ifdef __GNUC__
#include "callback.h"
#endif

#include "helloRsc.h"


static Err StartApplication (void)
{
    FrmGotoForm(MainForm);
    return 0;
}


static void StopApplication (void)
{
}


static void SaySomething (UInt16 alertID)
{
    FormType   *form = FrmGetActiveForm();
    FieldType  *field;
    MemHandle  h;


    field = FrmGetObjectPtr(form, FrmGetObjectIndex(form,
                            MainNameField));
    if (FldGetTextLength(field) > 0) {
        FldCompactText(field);
        h = FldGetTextHandle(field);
        if (h) {
            Char  *s;

            s = MemHandleLock((void *)h);
            FrmCustomAlert(alertID, s, NULL, NULL);
            MemHandleUnlock((void *)h);
        }
    } else {
        // No text in field, so display a "whoever you are"
```

```
        // dialog.
        FrmCustomAlert(alertID, "whoever you are", NULL, NULL);
    }
}


static Boolean MainMenuHandleEvent (UInt16 menuID)
{
    Boolean    handled = false;
    FormType   *form;
    FieldType  *field;


    form = FrmGetActiveForm();
    field = FrmGetObjectPtr(form,
        FrmGetObjectIndex(form, MainNameField));

    switch (menuID) {
        case MainEditUndo:
            FldUndo(field);
            handled = true;
            break;
        case MainEditCut:
            FldCut(field);
            handled = true;
            break;
        case MainEditCopy:
            FldCopy(field);
            handled = true;
            break;
        case MainEditPaste:
            FldPaste(field);
            handled = true;
            break;
        case MainEditSelectAll:
            FldSetSelection(field, 0,
                            FldGetTextLength(field));
            handled = true;
            break;

        case MainEditKeyboard:
            SysKeyboardDialog(kbdDefault);
            handled = true;
            break;

        case MainEditGraffitiHelp:
            SysGraffitiReferenceDialog(referenceDefault);
            handled = true;
            break;
```

Continued

Listing 4-3 *(continued)*

```
        case MainOptionsAboutHelloWorld:
            FrmAlert(AboutAlert);
            handled = true;
            break;

        default:
            break;
    }

    return handled;
}


static Boolean MainFormHandleEvent (EventPtr event)
{
    Boolean  handled = false;

#ifdef __GNUC__
    CALLBACK_PROLOGUE;
#endif

    switch (event->eType) {
        case frmOpenEvent:
        {
            FormType  *form = FrmGetActiveForm();

            FrmDrawForm(form);
            FrmSetFocus(form, FrmGetObjectIndex(form,
                MainNameField));
            handled = true;
        }
            break;

        case ctlSelectEvent:
            switch (event->data.ctlSelect.controlID) {
                case MainHelloButton:
                    SaySomething(HelloAlert);
                    handled = true;
                    break;

                case MainGoodbyeButton:
                    SaySomething(GoodbyeAlert);
                    handled = true;
                    break;

                default:
                    break;
            }
            break;
```

```
        case menuEvent:
            handled =
                MainMenuHandleEvent(event->data.menu.itemID);
            break;

        default:
            break;
    }

#ifdef __GNUC__
    CALLBACK_EPILOGUE;
#endif

    return handled;
}


static Boolean ApplicationHandleEvent (EventPtr event)
{
    FormType  *form;
    UInt16    formID;
    Boolean   handled = false;


    if (event->eType == frmLoadEvent) {
        formID = event->data.frmLoad.formID;
        form = FrmInitForm(formID);
        FrmSetActiveForm(form);

        switch (formID) {
            case MainForm:
                FrmSetEventHandler(form, MainFormHandleEvent);
                break;

            default:
                break;
        }
        handled = true;
    }

    return handled;
}


static void EventLoop (void)
{
    EventType  event;
    UInt16     error;
```

Continued

Listing 4-3 *(continued)*

```
    do {
        EvtGetEvent(&event, evtWaitForever);
        if (! SysHandleEvent(&event))
            if (! MenuHandleEvent(0, &event, &error))
                if (! ApplicationHandleEvent(&event))
                    FrmDispatchEvent(&event);
    } while (event.eType != appStopEvent);
}


UInt32 PilotMain (UInt16 launchCode, MemPtr cmdPBP,
                  UInt16 launchFlags)
{
    Err  err;


    switch (launchCode) {
        case sysAppLaunchCmdNormalLaunch:
            if ((err = StartApplication()) == 0) {
                EventLoop();
                StopApplication();
            }
            break;

        default:
            break;
    }

    return err;
}
```

Summary

In this chapter, you were introduced to the inner workings of a simple Palm OS application. After reading this chapter, you should understand the following:

- ✦ All Palm OS applications begin execution in the function **PilotMain**, which handles launch codes and starts the application's event loop.
- ✦ The event loop of a Palm OS application is responsible for dispatching events from the event queue to the appropriate event handlers.
- ✦ Most events can be handled by the system, using the functions **SysHandle Event**, **MenuHandleEvent**, and **FrmHandleEvent**.

- The **ApplicationHandleEvent** function, which is not provided by the OS, initializes forms and sets event handlers for them. These event handlers are callback functions that you write.
- You manage memory in the Palm OS by calling the operating system's memory API functions. Much of memory management involves locking a handle to a chunk of memory, using that memory, and then unlocking the handle so the OS is able to move that memory as it sees fit.

✦ ✦ ✦

CHAPTER 5

Debugging Your Program

In This Chapter

Using POSE

Debugging at the source level

Resetting a Palm OS handheld

Using developer Graffiti shortcuts

Using the Palm OS error manager

Between contributions from Palm Computing and the Palm OS development community, there is a rich set of tools available for debugging Palm OS applications. Emulators, source-level debuggers, an assembly-level debugger, and some developers' aids built into the Palm OS itself all help to make the Palm OS a programmer-friendly environment for writing bug-free applications. This chapter takes a look at tools and Palm OS features designed to make your life easier when it comes time to hunt down and squash bugs in your applications.

Using the Palm OS Emulator

Debugging your Palm OS applications would be a frustrating and laborious process if you were forced to install the applications to an actual handheld for testing. The debugging process usually involves a repeated cycle of compilation, testing, and fixing code before all the bugs have been worked out of a program. Installing an application on an actual Palm OS device is rather slow, which bogs down the testing portion of the debugging cycle.

Fortunately, the Palm OS Emulator (POSE) provides a much faster alternative. Based on Copilot, an emulator written by Greg Hewgill, POSE emulates a Palm OS handheld at the hardware level, right down to the Motorola DragonBall or DragonBall EZ processor. POSE can do almost anything that an actual handheld is capable of doing, with only a few omissions because of differences in hardware between a desktop system and a handheld device. Because POSE runs on the same system you use to do your development work, installing and testing applications is fast and simple. POSE even looks just like a Palm OS handheld; Figure 5-1 shows POSE as it appears on the desktop.

Figure 5-1: The Palm OS Emulator

In addition to simple emulation, POSE also provides a wealth of useful debugging features:

- ✦ The ability to enter text using the desktop computer's keyboard instead of having to use Graffiti or the on-screen keyboard
- ✦ Support for source-level debugging in external debuggers, such as CodeWarrior and GDB, both of which are described later in this chapter
- ✦ Configurable memory size, up to 8MB
- ✦ Automated random testing facility (Gremlins)
- ✦ Profiling code to determine what routines it spends the most time in, which is very useful for optimization
- ✦ Extensive monitoring of memory actions, to ensure that your application does not try to access memory that it should leave alone, such as low memory or hardware registers
- ✦ Complete logging of all application activity, including handled events, called functions, and executed CPU operation codes
- ✦ Easy screen shot facility, great for showing off an application on Web sites (or in programming books)
- ✦ Redirection of network library calls to the host system's TCP/IP stack
- ✦ Redirection of serial I/O to the host system's own serial ports

If you have read any of Palm Computing's documentation, there is a good chance you have seen mention of the Palm OS Simulator. Do not confuse the Simulator with the Palm OS Emulator; they are not the same tool. The Simulator is available as part of the Mac OS version of CodeWarrior only. Simulator allows you to build a standalone Macintosh application that simulates your application. If you have POSE for Mac OS, you can probably ignore Simulator entirely; POSE is a newer and much more versatile debugging tool.

Palm Computing supports versions of POSE for both Windows and Mac OS; both versions, as well as their source code, are available as free downloads. There is also a Unix source code distribution, which still enjoys excellent response time from developers at Palm Computing and in the Palm OS development community, despite the Unix version's official status as unsupported software. With the source code at your disposal, you can also alter the Emulator to your own tastes. If you come up with a modification (or a bug fix) that might be useful to other developers, be sure to send the changes back to Palm Computing so they can roll them into the main POSE code base. Much of POSE has actually been developed in this way, from contributions by programmers outside of Palm Computing; this spirit of cooperation, combined with outstanding efforts on the part of Palm Computing, has made POSE an indispensable tool for Palm OS development.

In addition to the wonderful features mentioned previously, POSE is an economical way for you to test software on a variety of different Palm OS systems without having to drop a large amount of cash on several pieces of hardware. POSE can emulate the hardware side of most Palm OS devices on the market (including limited support for third-party hardware, such as the Visor, the TRGPro, and Symbol's SPT series), and Palm Computing provides ROM images for every release of the Palm OS since version 1.0.

Even though POSE is able to perform most of the testing required to make a reasonably bug-free Palm OS application, there are subtle differences between POSE and an actual hardware Palm OS device. As a final step in testing an application, you should install it on an actual Palm OS handheld to make sure it works in a real environment, rather than just the virtual environment provided by POSE. In particular, POSE does have the following limitations:

- Graffiti input is difficult using a mouse, trackball, or touch pad. Though POSE allows easy text input using the desktop computer's keyboard, if your application needs to deal with Graffiti input at a lower level, such as processing individual stylus strokes, it can be difficult to test such input using POSE.
- POSE has no way to simulate an infrared port. You can perform some infrared testing by setting POSE in infrared loopback mode using the "t" shortcut, or by switching the Emulator into serial IR mode with the "s" shortcut (see the "Using Developer Graffiti Shortcuts" section later in this chapter for details). However, in the final equation, the only real way to fully test IR beaming is with a pair of suitably equipped Palm OS handhelds.

- Execution speed in POSE is directly affected by the speed of the desktop system, and by whatever other processes you might be running on that system. This means that POSE may run slower or faster than an actual handheld, which makes code optimization very tricky.

POSE is under constant development, and the folks at Palm Computing release new versions very rapidly. Keep an eye on the Palm Computing Platform Development Zone Web site for information on the latest release of POSE, or better yet, subscribe to the Emulator Forum mailing list for an up-to-the-minute view of POSE development. See Appendix B, " Finding Resources for Palm OS Development," for more information about the Development Zone Web site and the Emulator Forum mailing list.

Controlling POSE

Interacting with applications running in POSE is very simple. Just use the mouse as if it were a stylus, and POSE will respond the same way a real Palm OS handheld would. POSE also recognizes Graffiti characters drawn in its silk-screened Graffiti area. Because it is rather difficult (and time-consuming) to use a mouse for Graffiti input, you may also type on the desktop computer's keyboard to enter text into POSE.

Clicking the hardware buttons on the POSE display is also identical to pressing hardware buttons on a real device. You can even press and hold the buttons to perform actions such as continuous scrolling. As an example, try clicking and holding down the power button for a few seconds to activate POSE's "backlight." In addition to using the mouse with the on-screen buttons, you can use keyboard shortcuts for these buttons, as outlined in Table 5-1. These keyboard shortcuts are identical to clicking the on-screen buttons in the Emulator display.

Table 5-1
POSE Hardware Button Keyboard Shortcuts

Button	*Keyboard Shortcut*
Power	Esc
Date Book	F1
Address Book	F2
To Do List	F3
Memo Pad	F4
Scroll Up	Page Up
Scroll Down	Page Down

In addition to these shortcuts, POSE also understands a number of Control key combinations that send special virtual character codes, which the Palm OS interprets to perform special actions, such as displaying a menu bar. Table 5-2 describes some of these shortcuts.

Table 5-2
POSE Control Key Shortcuts

Shortcut	*Description*
Ctrl+A	Sends a `menuChr` character, opening the menu bar on the current form
Ctrl+B	Sends a `lowBatterChr` character, which the system normally sends when battery power is low as a notification to the current application that it should respond accordingly to the low power condition
Ctrl+C	Sends a `commandChr` character, which is the special Graffiti command stroke that allows for rapid activation of menu commands
Ctrl+D	Sends a `confirmChr` character
Ctrl+E	Sends a `launchChr` character, which starts the system application launcher program
Ctrl+F	Sends a `keyboardChr` character, which displays the on-screen keyboard dialog
Ctrl+M	Sends a `linefeedChr` character (a simple linefeed)
Ctrl+N	Sends a `nextFieldChr` character, which moves entry to the next text field in applications that handle `nextFieldChr`
Ctrl+S	Sends an `autoOffChr` character, which the system sends when the auto-off timeout has been reached
Ctrl+T	Sends a `hardContrastChr` character, which launches the contrast adjustment screen on the Palm V and other devices that support software contrast adjustment
Ctrl+U	Sends a `backlightChr` character, which toggles the backlight on and off on devices that have a backlight

To control POSE-specific functions, the Palm OS Emulator has a menu. On Windows and Unix, the menu is a pop-up, which you can open by right-clicking anywhere in the POSE display, or by pressing the F10 key. On the Mac OS, the menu is presented as a menu bar that floats just above the main POSE display. Menu commands on the Mac OS menu bar are separated into four menus: File, Edit, Gremlins, and Profile. Figure 5-2 shows the pop-up menu in the Windows version of POSE.

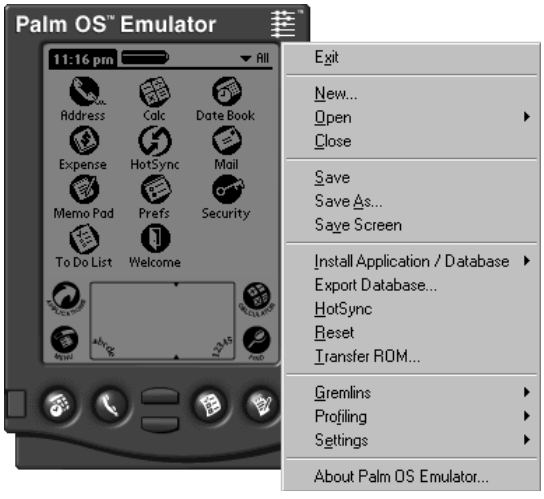


Figure 5-2: The POSE pop-up menu in Windows

Running POSE for the First Time

When you start POSE for the first time, or if the Caps Lock key is active, the Emulator presents you with the dialog shown in Figure 5-3, but only if you are running POSE on a Windows system. A Mac OS system displays the New Configuration dialog, instead; see Figure 5-4 for the Windows version of the New Configuration dialog, which is very similar to the Mac OS version. Unix POSE does not have an automatic startup sequence.

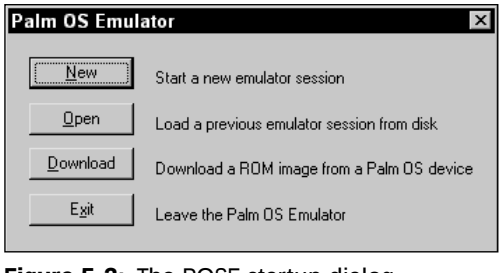


Figure 5-3: The POSE startup dialog

The New button fires up the New Configuration dialog, shown in Figure 5-4, which prompts you for all the parameters required to start a brand new Emulator session; the "Installing a ROM Image" section later in this chapter has more information about the New Configuration dialog. The Open button opens a file dialog, prompting you to select a saved emulator session to load; see the "Saving and Restoring Configurations" section of this chapter for more details. Clicking the Download

button begins the process of downloading a ROM image from a handheld connected to the desktop machine by cradle; see the "Downloading an Image from an Actual Handheld" section for more details. If you click Exit, POSE quits.

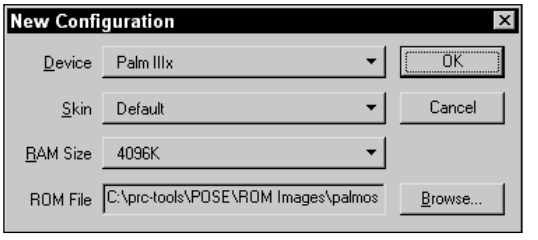
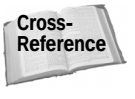


Figure 5-4: The New Configuration dialog in POSE

Installing a ROM Image

POSE emulates only the hardware of a Palm OS handheld. In order for the emulator to be truly useful, it needs a ROM image containing the Palm OS system software. Palm Computing has ROM images for all major releases of the Palm OS available for download from its Web site. ROM image files have a `.rom` extension.



You must be a member of the free Palm Solution Provider Program to be able to download ROM images. See Appendix B, "Finding Resources for Palm OS Development," for more details about joining the Palm Solution Provider Program.

For most versions of the Palm OS, Palm Computing provides both release and debug versions of each ROM image. Release ROM images are identical to the ROM contents of an actual Palm OS handheld. Debug ROM images contain extra debugging code to help you track down hard-to-find errors in your code, such as illegal attempts to access protected memory areas.

Be sure to test your application on debug ROM images if they are available for the platforms you intend to support. The release images run a bit faster on the Emulator (particularly for Palm OS 3.5), but release code is designed to fail gracefully and quietly if anything goes wrong, which will prevent you from seeing some kinds of errors, particularly memory leaks. It is well worth the effort to test an application on the debug ROM images, because they produce very useful debugging information.

If you click the New button in the POSE startup dialog, or if you choose the New item from the POSE menu, the New Configuration dialog appears, as shown in Figure 5-4.

From this dialog, you can choose the hardware and amount of memory you want to emulate, the appearance of POSE on the desktop, and a ROM image file. Clicking the Device button presents a drop-down menu, from which you should select the hardware device that you want to emulate.

The Skin button allows you to customize the appearance of POSE by applying a *skin*, or bitmap image, to the Emulator. Depending on what hardware you have selected, different skins are available. The "Default" skin is always an option and results in a generic PalmPilot-shaped case graphic with "Palm OS Emulator" written across it, as pictured in Figure 5-1. The "Standard - English" skin gives you a graphic that looks exactly like the type of hardware you have selected, and the "Standard - Japanese" draws the emulator as an equivalent IBM Workpad with the Japanese language silkscreen area. In POSE version 3.0a5 and later, you can even skin the emulator as a Symbol SPT 1700 or a Handspring Visor (in different colors!). Figure 5-5 shows off POSE as a Japanese IBM Workpad c3; the settings required to make POSE look like this are the Device selection of "Palm IIIx" and the "Standard - Japanese" skin.

Figure 5-5: By wearing different skins, POSE can look like almost any Palm OS handheld.

Set the size of the memory available to the Emulator with the RAM Size button's drop-down menu. You can set the size at various values between 128KB and 8MB. By setting the memory size, you can duplicate the memory available on actual Palm OS handhelds or see what happens when your application is running on a device with very little memory to spare.

Finally, click the Browse button next to ROM File to open a file dialog and browse for an appropriate `.rom` file to install. Make sure the ROM image you choose will run on the hardware you have selected. Otherwise, POSE will display an error dialog if you choose a ROM file that is incompatible with the hardware you have selected, such as trying to load a Palm IIIx ROM onto an emulated Palm VII device.

You can also install a ROM image by dragging the `.rom` file onto POSE with the mouse. Doing so opens the New Configuration dialog (as shown in Figure 5-4) with the appropriate ROM file preselected.

Instead of using a ROM image from Palm Computing, you can download the ROM from an actual Palm OS handheld. As of this writing, this is the only way to get ROM images for some third-party handhelds, such as the Handspring Visor.

To perform a ROM transfer from a handheld to the desktop, you must first inst all the `ROM Transfer.prc` file to the handheld as you would any other Palm OS application. This `.prc` file contains ROM Transfer, a utility program that handles the handheld side of the ROM transfer.

Once you have ROM Transfer installed, leave the handheld in its cradle. Make sure the HotSync Manager is not running, because the ROM image transfer cannot take place if the serial connection is busy. Then, follow these steps to transfer the ROM to the desktop computer:

1. On the handheld, start the ROM Transfer application.
2. If you are running the Windows version of POSE for the first time, click the Download button in the POSE startup dialog (as shown in Figure 5-3). On Windows, as well as on Unix, you can also select the Transfer ROM option from the pop-up menu. If you are running POSE on the Mac OS, select File➪ Transfer ROM from the POSE menu bar. POSE displays the Transfer ROM dialog, shown in Figure 5-6.

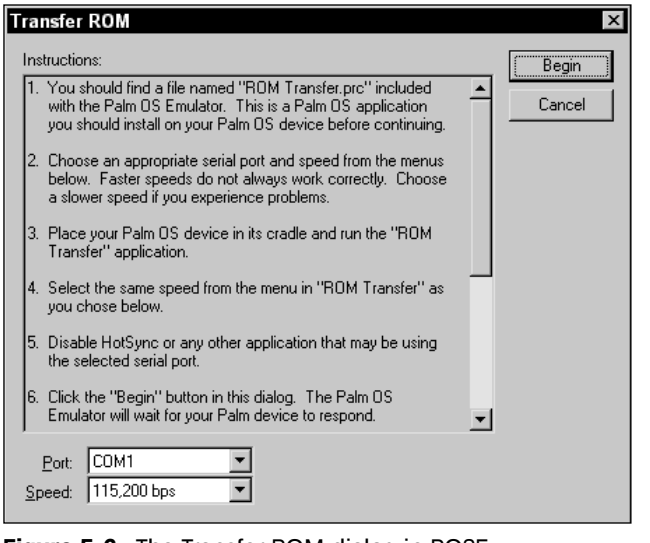


Figure 5-6: The Transfer ROM dialog in POSE

3. Select an appropriate serial port and connection speed in the Transfer ROM dialog.
4. On the handheld, select the same connection speed from the Transfer Speed pop-up list that you set on the desktop.

5. Click the Begin button in the Transfer ROM dialog on the desktop.
6. Tap the Begin Transfer button on the handheld.
7. Wait a few minutes while the transfer takes place.
8. Once the transfer is complete, POSE will present you with a file dialog, prompting you for a file name and location to save the transferred ROM image.

After following these steps, you should have a `.rom` file containing an image of your handheld's ROM. Install this ROM image the same way you would install a debug ROM downloaded from Palm Computing, as described earlier in this section.

Installing Applications

Because performing an actual HotSync operation to install files to the Emulator would be a waste of time, POSE allows you to install Palm OS applications and databases in a much more direct fashion. On Windows and Unix, select the Install Application/Database menu item; on the Mac OS, select File➪Install Application/Database. POSE presents you with a standard file dialog, from which you may select any `.prc`, `.pdb`, or `.pqa` files that you want to install.

POSE cannot install a new copy of a database that already exists in POSE if the old database is in use. If you are running an application in POSE and you want to overwrite it (or its database) with a new version, switch to another application in POSE before installing. The Calculator or the other four built-in applications are good choices, because you need to click only once to launch them.

Once you have installed the applications you want to test, you can open them just as you would on a real handheld by clicking on the application launcher button, and then selecting the appropriate icon.

The system application launcher does not update its icons in response to the installation of a new application in POSE, so a new application's icon will not immediately appear if the launcher is open when you install the new application. Switch to another application, and then re-open the launcher, and the new icon will appear.

You can also install applications and databases by dragging-and-dropping them onto POSE. To save even more time, you can select as many `.prc`, `.pdb`, and `.pqa` files as you want, and drag all of them at once for a single mass installation.

Saving and Restoring Configurations

If you have POSE set up just the way you want it, you can take a snapshot of POSE's current state and save it for later retrieval. Saving a POSE session keeps track of all aspects of the Emulator, including the selected hardware type and skin, all the currently installed applications and databases, and even the exact state of the emulated RAM.

To save a session in Windows or Unix POSE, select the Save or Save As menu options from the pop-up menu. In Mac OS POSE, select the File ➪ Save or File ➪ Save As menu commands. If the current session is loaded from a saved position, the Save command overwrites the currently open session file. Otherwise, Save does the same thing as Save As, which is to present you with a standard file dialog to prompt you for a file name and location to use for saving the current session. POSE saves Emulator session files with an extension of `.psf`.

To open a saved session in Windows or Unix, select Open from the pop-up menu; on the Mac OS, select File ➪ Open. POSE presents you with a file dialog, from which you can select the `.psf` file that contains the desired emulator session.

To save time when testing an application on multiple platforms, create a "clean" session for each ROM image you want to test. Right after creating a new configuration in the New Configuration dialog (as shown in Figure 5-4), and before installing any applications or data, save the session with a descriptive name, such as `Palm IIIx 3.5 debug.psf`. Then you can quickly retrieve saved sessions by using the Open menu command. You can also drag a `.psf` file onto POSE to open a particular saved session.

Adjusting POSE Settings

The Palm OS Emulator is incredibly configurable. Most of the configuration options for POSE are located under the Settings option in the pop-up menu on Windows or Unix, or under the Edit menu in the Mac OS version of POSE. Among other options, you can change the way POSE looks, how POSE communicates with the desktop, when POSE should break execution, and how strictly POSE should monitor different kinds of memory access.

Setting properties

The Settings ➪ Properties menu command on Windows and Unix (Edit ➪ Preferences on Mac OS) opens the Properties dialog, shown in Figure 5-7.

From the Properties dialog, you can control several different kinds of POSE behavior. The different parts of the dialog are described as follows:

- **Serial Port:** Use this drop-down list to select a serial port on the desktop computer. POSE redirects serial port calls from applications on the Emulator through the selected serial port on the desktop machine, which allows you to test serial communications by connecting a serial cable between the appropriate port and whatever device POSE should talk to. You can even connect a pair of serial ports on the desktop machine with a null modem cable to allow POSE to communicate with a terminal program on the same computer.

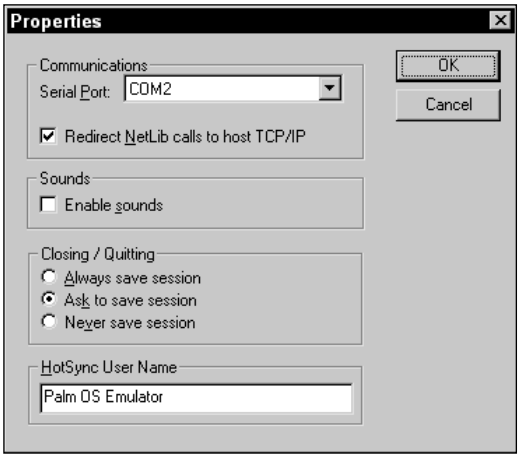


Figure 5-7: The Properties dialog in POSE

- **Redirect NetLib calls to host TCP/IP:** When checked, this option redirects network library calls in POSE to the TCP/IP stack on the host computer. This option is a fantastic way to test Internet-enabled Palm OS applications without having to wait for a connection through a handheld modem.
- **Enable sounds:** If this is checked, POSE attempts to provide audio feedback from the Palm OS through the desktop computer. This sound reproduction is not faithful to what you would hear on an actual Palm OS handheld; in fact, depending on your desktop system, POSE might be capable of only primitive beeps and clicks through the system's speaker. For applications where sound is an important part of the program, such as music or game programs, test the sound on a real handheld.
- **Closing/Quitting:** The radio buttons in this section of the dialog control how POSE saves the current session when it closes. If Always save session is selected, POSE automatically overwrites the current session file with the emulator's current state when the emulator exits. If Ask to save session is selected, POSE prompts you to save the session, and if Never save session is selected, POSE simply exits without saving.
- **HotSync User Name:** This text box gives you a place to specify the HotSync name to use when emulating a HotSync operation in POSE. Normally, every Palm OS handheld has a name saved in it to identify its user, which comes in handy when multiple organizers are synchronized with the same desktop machine. See the "Emulating a HotSync Operation" section later in this chapter for more details.

Setting debug options

Selecting Settings➪Debug Options on Windows or Unix, or Edit➪Debug Options on the Mac OS, presents you with the Debug Options dialog, shown in Figure 5-8.

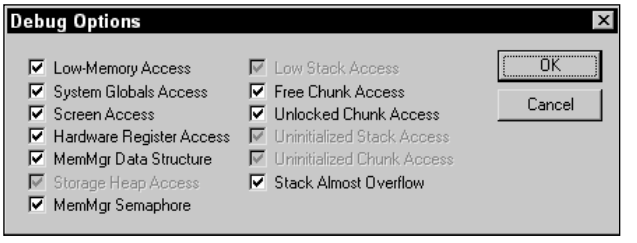


Figure 5-8: The Debug Options dialog in POSE

In this dialog, you can set how sensitive POSE is to certain actions a program might perform that are either illegal or counter to recommended Palm OS programming practice. By default, POSE leaves all these options enabled, and so should you for most applications. If you happen to be writing an application that cannot strictly conform to Palm Computing's programming guidelines, such as low-level system hacks that are intended to run only on a single, very specific hardware model, you can disable warnings that you do not want to see.

POSE is most useful with all of the debug options enabled. If you leave the Debug Options dialog alone, POSE can help you make sure your application not only runs properly on current Palm OS handhelds but also on future devices by following Palm Computing's recommendations.

Changing POSE's appearance

On Windows or Unix, the Settings ➪ Skins menu command opens the Skins dialog, shown in Figure 5-9. You can open this dialog in the Mac OS version of POSE from the Edit ➪ Skins menu option.

Figure 5-9: The Skins dialog in POSE

From the Skins dialog, you can set the skin that POSE uses to display the emulated handheld. There are also other appearance-related options at the bottom of the dialog:

- **Double scale:** When checked, this option doubles the size of POSE on the screen. The double scale mode is particularly useful for making fine adjustments to screen layout during the design stages of an application, because you can count individual pixels without having to squint at an Emulator the size of a real handheld. If you run your desktop machine at 1600×1200 or greater resolution, double scale might simply be a necessity if you want the POSE display to be legible at all.
- **White Background:** When checked, this option replaces the usual LCD green display background with basic white. This option makes for much cleaner screen shots, and you might simply prefer it over the green background, because it improves contrast significantly.

Setting breakpoints

In the Windows version of POSE, you can set breakpoints in your code, places where a program will suspend execution and pass control to a debugger. You can set breakpoints in the Breakpoints dialog (as shown in Figure 5-10), which is available from the Settings➪Breakpoints menu command. Breakpoints are not available in the Mac OS or Unix versions of POSE. When POSE breaks execution while connected to an external debugger, such as CodeWarrior or GDB, POSE sends notification to the debugger that a breakpoint has been hit, and the debugger handles the break. If POSE is not connected to a debugger, it displays an error message when it hits a breakpoint, which typically displays something like "TRAP $0 encountered."

Using POSE to set breakpoints requires working knowledge of assembly code and the Motorola 68k processor family architecture. POSE breakpoints are not for inexperienced developers or the faint of heart. For most debugging purposes, it is much easier to set breakpoints in CodeWarrior or GDB, which are described later in this chapter.

From the Breakpoints dialog, you can set up to six conditional code breakpoints and a single data breakpoint. To set a code breakpoint, select a breakpoint from the list and click Edit, or just double-click the desired breakpoint in the list. The Code Breakpoint dialog, shown in Figure 5-11, appears.

Specify the address of the code breakpoint, in either decimal or hexadecimal (0x) format, in the Address text box. Set the condition that must be true to generate a break by entering the condition in the Condition text box. If the program counter reaches the specified address, and the condition specified is true, POSE generates a break. Conditions must have the following format:

```
<register> <condition> <constant>
```

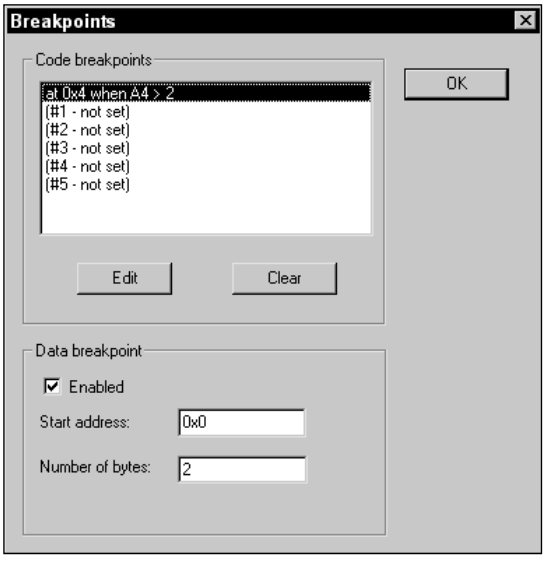


Figure 5-10: The Breakpoints dialog in POSE

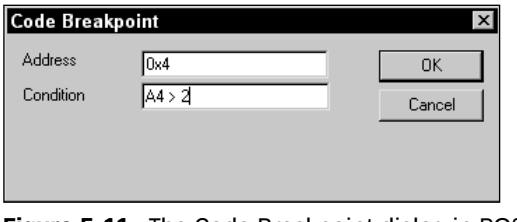


Figure 5-11: The Code Breakpoint dialog in POSE

For `register`, you must specify one of the following registers: `A0`, `A1`, `A2`, `A3`, `A4`, `A5`, `A6`, `A7`, `D0`, `D1`, `D2`, `D3`, `D4`, `D5`, `D6`, or `D7`. The `condition` value may be one of the following operators: `==`, `!=`, `<`, `>`, `<=`, or `>=`. The `constant` value may be any decimal or hexadecimal constant value.

To set a data breakpoint, check the Enabled check box in the Breakpoints dialog. The data breakpoint causes a break if the contents of a specific range of memory change while POSE is running. Set the beginning address of the range in either decimal or hexadecimal format in the Start address text box, and define the size of the range by entering the number of bytes in the Number of bytes text box.

Handling Gremlins

Structured functional testing is a good way to make sure that all the parts of an application are working the way they are supposed to be working. However, sometimes the best way to find obscure bugs in an application is by randomly banging on it until something breaks. Gremlins are a testing feature built into POSE that allow you to perform random tests on your application to shake out those bugs that functional testing might not find. More importantly, each Gremlin test is completely reproducible, so you can run the same Gremlin again and get the same error, making it much easier to see what went wrong.

Gremlins use several techniques to torture test an application. While they are running, Gremlins simulate taps on random areas of the screen, but areas that contain actual user interface elements, such as buttons and pop-up triggers, particularly attract them. Gremlins also enter Graffiti text into the application, both random strings of garbage and occasional quotes from Shakespeare.

An individual Gremlin has a seed value from 0 to 999, and each produces its own unique series of random stylus and key input events. You can restrict a Gremlin to stay within a particular application or group of applications; this is particularly useful when you want to concentrate on hammering your own code without wasting time in other programs.

Gremlins may also be run in *Gremlin hordes*. A Gremlin horde will run multiple Gremlins, giving your application a real workout. Setting up a Gremlin horde to run overnight is a good way to ensure that a program has been thoroughly tested.

To set up Gremlins select the Gremlins➪New menu command. POSE displays the New Gremlin Horde dialog, shown in Figure 5-12.

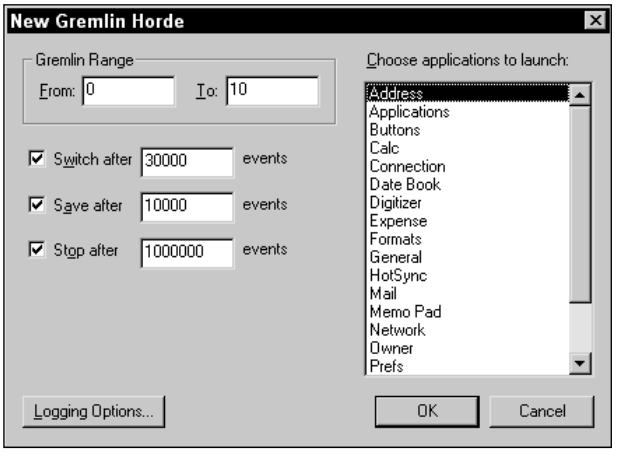


Figure 5-12: The New Gremlin Horde dialog in POSE

The parts of the New Gremlin Horde dialog are described as follows:

- **Gremlin Range:** Set the Gremlins that make up the beginning and end of a Gremlin horde in the From and To text boxes. Enter a number from `0` to `999` into each box, and POSE will run each Gremlin in this range. To run just a single Gremlin, enter the same value into both boxes.
- **Switch after:** If this is checked, POSE will switch to a new Gremlin in the horde when the currently running Gremlin generates the specified number of events; this number of events is called the *switching depth*. If this box is not checked, POSE will not switch to a new Gremlin in the horde until each Gremlin hits the number of events specified in the Stop after text box. Because a particular Gremlin always repeats the same series of events each time it runs, making Gremlins take turns at attacking your application can add more of a random element to your testing.
- **Save after:** If this is checked, POSE saves a snapshot of the Emulator as a `.psf` file after the specified number of events. You can open the `.psf` file in POSE later to examine the state of POSE at that particular moment in time, or to begin debugging again from a specific point. Make sure you have sufficient disk space if you set Save after to a low value, because this option can fill your hard drive very quickly with `.psf` files.
- **Stop after:** If this is checked, POSE stops each Gremlin when it generates the specified number of events. Without this box checked, a Gremlin will run indefinitely, or until it encounters an error.
- **Choose applications to launch:** Select an application or group of applications to restrict the Gremlin horde to while it is running. You can select multiple applications by holding down the Ctrl key (Control on a Macintosh) while clicking items in the list.
- **Logging Options:** Click this button to change exactly what events, actions, and errors Gremlins will write to a log file. The default options are sufficient for most debugging purposes, but you may also wish to check other options if your application performs special actions. For example, enabling the Serial Activity and Serial Data options may help you discover the cause of bugs in an application that uses the serial port.
- **OK:** Click this button to launch the Gremlin horde.
- **Cancel:** Click this button to exit the dialog without starting a new Gremlin horde.

When running a Gremlin horde, POSE steps through the following sequence of events:

1. POSE saves the current state of the Emulator to a `.psf` file.
2. POSE starts the first Gremlin, indicated by the value of the From text box.

3. The first Gremlin runs until it posts a number of events equal to the Switch after value, at which point POSE saves the Emulator's state and suspends that Gremlin. If the Gremlin encounters an error before it hits the switching depth value, POSE terminates the current Gremlin instead of suspending it.
4. POSE loads the original saved state of the Emulator.
5. The second Gremlin begins execution and runs until it hits the switching depth or encounters an error.
6. POSE runs each Gremlin in the horde until each one has been suspended or terminated.
7. Now POSE returns to the first suspended Gremlin in the horde and reloads its saved Emulator state. The Gremlin then runs from where it left off the last time. POSE skips over Gremlins that have been terminated because of errors, restarting only those that were suspended after reaching the switching depth.
8. The entire process repeats itself, with POSE suspending Gremlins as they reach the switching depth again, or terminating those that produce errors. Each Gremlin runs until it has finished. A Gremlin is finished when POSE has terminated it because of an error, or when the Gremlin reaches a total number of events equal to the Stop after value specified in the New Gremlin Horde dialog.

While POSE is running Gremlins, the Gremlin Control dialog, shown in Figure 5-13, appears. The Gremlin Control dialog allows you to control the execution of Gremlins more directly, and it displays the current event that is executing, the number of the current Gremlin, and the total elapsed time that the Gremlin horde has been running. Clicking the Stop button pauses execution of the Gremlin horde, and clicking Resume continues the horde where it left off. When a horde is stopped, you can also click the Step button to step through a few Gremlins at a time.

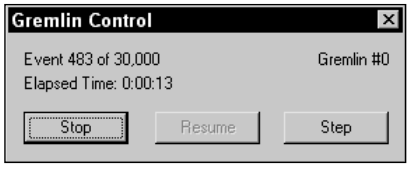


Figure 5-13: The Gremlin Control dialog in POSE

Try running Gremlins when you have a source-level debugger hooked up to POSE, such as CodeWarrior or GDB. If your application generates an error while a Gremlin is playing with it, the debugger will drop straight to the line of code responsible for the error.

Emulating a HotSync Operation

Any application that has a close relation with desktop data eventually needs to be tested to see how it behaves during a HotSync operation. Setting up POSE to properly communicate with the desktop during a HotSync operation is somewhat tricky, but it can be done. There are two ways to set up POSE for a HotSync operation:

- Set POSE to use one of the desktop system's serial ports and the HotSync Manager to use a different serial port, and then connect the two ports with a null modem cable. This setup is cumbersome and requires a machine with two free serial ports, which can be a problem if, as is common on many PCs, one of the serial ports is in use by a mouse or other peripheral. However, this setup is the only way to test HotSync operations in POSE with a ROM image containing Palm OS 3.0 or earlier.
- Set up POSE and the HotSync Manager for a Network HotSync operation. This is a much simpler way to test HotSync connections, because it does not require an extra cable connection at the back of the computer. However, it works only with the Windows version of the HotSync Manager. Also, you may only perform a Network HotSync operation with a Palm OS 3.1 or later ROM image, or with a 3.0 image and the NetSync.PRC application. NetSync.PRC adds Network HotSync capability to Palm OS 3.0, and the program is available for download from `http://www.palm.com/custsupp/downloads/netsync.html`. If you want to set up Network HotSync on a Palm OS 3.0 ROM image, begin by installing NetSync.PRC on the Emulator, and then reset POSE by right-clicking the Emulator and choosing Reset.

To set up POSE to use the serial port for HotSync operations, follow these steps:

1. In POSE, open the Properties dialog (shown in Figure 5-7). Set POSE to use one of the host machine's serial ports.
2. Make sure the HotSync Manager running on the desktop is set to use a different serial port from the one set up in POSE.
3. Connect the two serial ports with a null modem cable.

To set up POSE to use the Network HotSync configuration, follow these steps:

1. Click the HotSync icon in the system tray, and select Network from the pop-up menu that appears if Network does not already have a check mark next to it.
2. Open the Properties dialog in POSE by selecting the Settings ➪ Properties menu command from POSE's pop-up menu. Check the Redirect NetLib calls to host TCP/IP check box and click OK.
3. Open the HotSync application in POSE.
4. Select the Options ➪ Modem Sync Prefs menu command. In the Modem Sync Preferences dialog that appears (as shown in Figure 5-14), select the Network push button. Tap the OK button.

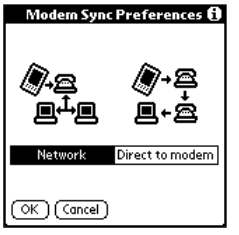


Figure 5-14: The HotSync application's Modem Sync Preferences dialog

5. Select the Options ➪ LANSync Prefs menu command. In the LANSync Preferences dialog that appears (as shown in Figure 5-15), select the LANSync push button. Tap the OK button.

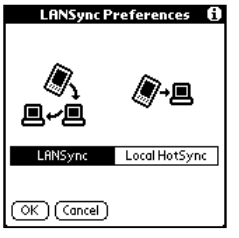


Figure 5-15: The HotSync application's LANSync Preferences dialog

6. Select the Options ➪ Primary PC Setup menu command. In the Primary PC Setup dialog that appears (as shown in Figure 5-16), enter the network address `127.0.0.1` into the Primary PC Address text field. Tap the OK button.

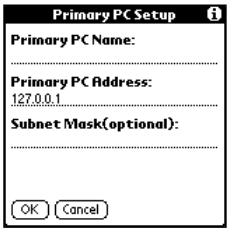


Figure 5-16: The HotSync application's Primary PC Setup dialog

7. Depending on which version of the Palm OS is running in POSE, you will need to do one of the following:

- On Palm OS 3.1, tap the Select Service selector trigger under the Modem Sync icon to open the Preferences dialog (as shown in Figure 5-17). Tap the Phone selector trigger, enter two zeroes (`00`) into the Phone # field, and then tap the OK button. Tap the Done button.

- On Palm OS 3.3 and later, tap the Modem push button, and then tap the Select Service selector trigger under the HotSync icon to open the Preferences dialog. Tap the Phone selector trigger, enter two zeroes (00) into the Phone # field, and then tap the OK button. Tap the Done button.

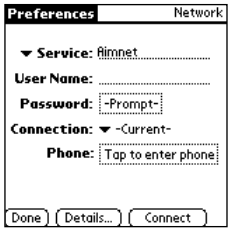


Figure 5-17: The HotSync application's service Preferences dialog

Note that you do not actually have to set up any information for a service profile in Step 7; you need only have a service selected. POSE should now be ready for a network HotSync operation.

Telling POSE to start a HotSync operation is easy. Either select the HotSync menu command in Windows or Unix (or the File➪HotSync command on the Mac OS), or open the HotSync application in POSE and tap the HotSync button in the center of the screen (or the Modem HotSync button, if you are using a Network HotSync setup on Windows).

The HotSync Manager is very processor intensive, which can cause very slow HotSync operations when both the HotSync Manager and POSE are fighting for resources on the same machine. To speed up HotSync operations in POSE, click the POSE window to bring it to the foreground after starting a HotSync operation.

POSE uses the value you set for HotSync User Name in the Properties or Preferences dialog (shown in Figure 5-7) as the user name to identify the emulated handheld. When you first synchronize POSE with the desktop, the HotSync Manager will prompt you for a user profile with which to synchronize.

Synchronizing POSE with your own personal data is a sure way to cause all sorts of trouble, including lost data or duplicate records. Your best bet is to create a brand new user profile on the desktop, exclusively for use with POSE.

Taking Screen Shots

Making screen shots of Palm OS applications in POSE is simplicity itself. Once you have the screen you want a picture of visible in the Emulator's window, select the Save Screen menu option in Windows or Unix (File➪Save Screen on the Mac OS). POSE presents you with a file dialog, where you can specify the file name and location to save the image file. POSE saves screen shots as 160×160 pixel `.bmp` files on Windows systems, or as SimpleText images on the Mac OS.

To make screen shots more legible, open the Skins dialog (shown in Figure 5-9) and check the White Background option before capturing the screen.

Handling Errors in POSE

When POSE encounters an error during execution of an application, it displays an error dialog box, shown in Figure 5-18. The text of the error message may be different from what is pictured here, of course, as the text depends on what the actual error is.

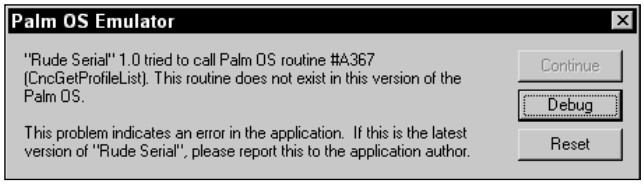


Figure 5-18: The Palm OS Emulator error dialog

Depending on the type of error POSE has encountered, the Continue or Debug buttons may be grayed out and thus unusable. The buttons have the following effects when clicked:

- **Continue:** POSE tries to continue executing application code, if possible.
- **Debug:** POSE hands over execution to an external debugger, such as CodeWarrior or GDB, if one is running.
- **Reset:** POSE performs a soft reset of the emulated device, which may allow you to continue running the Emulator without having to start an entirely new session. You can force a soft reset at other times by selecting the Reset menu command in Windows and Unix, or the File ➪ Reset command on the Mac OS. Forcing a reset can be useful in testing an application that is designed to react to a soft reset through the `sysAppLaunchCmdSystemReset` launch code.

Debugging at the Source Level

Source-level debugging is one of the most effective ways to track down and fix bugs in an application. Instead of trying to guess which part of your code is causing an error, you can step through the code line by line and find out exactly what is causing a bug. The CodeWarrior IDE has a built-in debugger that can debug applications running on an actual serial-connected handheld or on POSE, and the GDB debugger that ships with the GNU PRC-Tools can debug applications running on POSE.

Debugging with CodeWarrior

To perform source-level debugging in the CodeWarrior environment, you must first enable the debugger. Select the Project➪Enable Debugger menu command to enable debugging; if Project➪Disable Debugger is displayed in the menu, the debugger is already enabled, because this menu item toggles between Enable Debugger and Disable Debugger.

You may also want to select the Debug➪Show Breakpoints menu command, if it has not already been selected, to display the breakpoint column in the project's code windows. Figure 5-19 shows a CodeWarrior code window with the breakpoint column displayed. By clicking in the breakpoint column, you can enable or disable a breakpoint on a particular line of code; breakpoints are designated with a red dot.

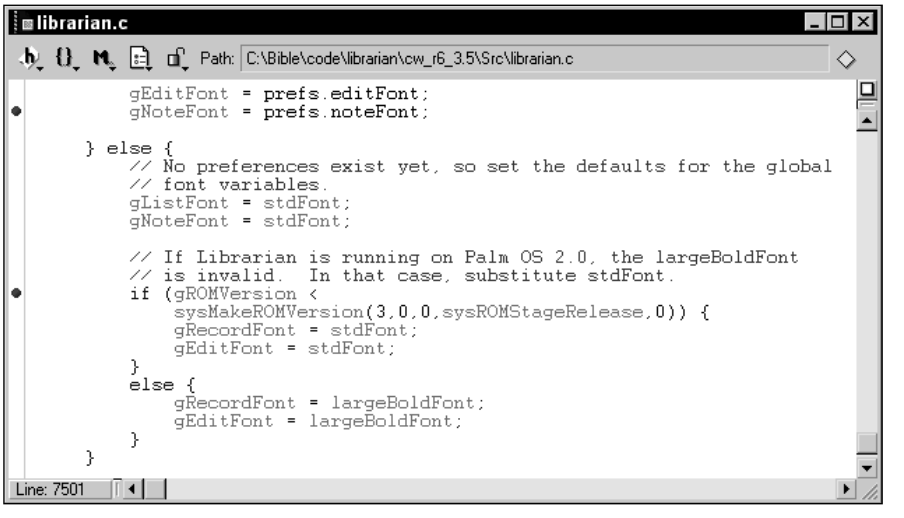


Figure 5-19: A CodeWarrior code window with the breakpoint column displayed

Setting up a debugging target

Because CodeWarrior can debug applications in both POSE and on an actual handheld, you need to set up a target for the debugger to connect to. To set the target, you need to open the IDE Preferences dialog, shown in Figure 5-20, by selecting the Edit➪Preferences menu command, and then selecting the Palm Connection Settings panel in the IDE Preferences Panels list at the left of the dialog. To connect to POSE for debugging, choose "Palm OS Emulator" from the Target drop-down list, as shown in Figure 5-20. You must also set the Emulator text box to the path and file name of POSE on your system; use the Choose button to look for the Emulator using a standard file dialog. Tap the Save button to save the debugger settings, and then close the dialog.

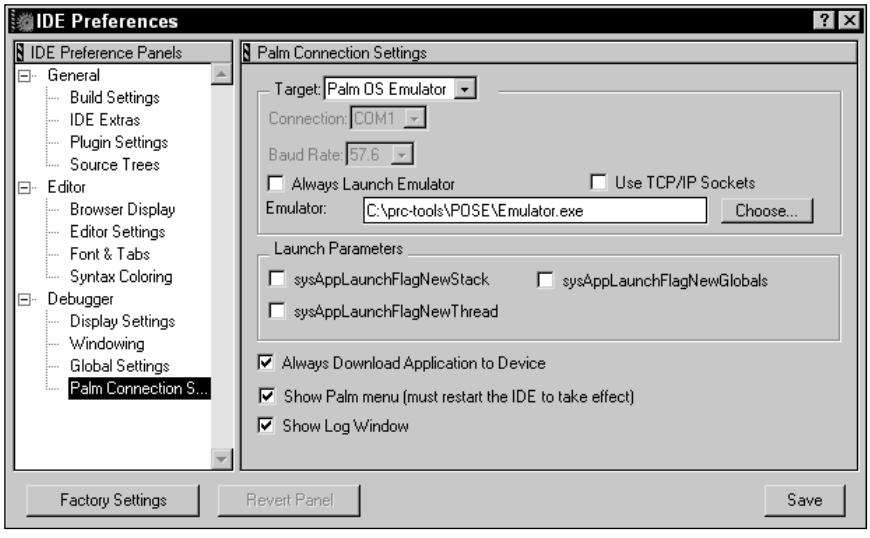


Figure 5-20: The CodeWarrior IDE Preferences dialog, set up to connect to POSE for debugging

To connect to an actual Palm OS handheld, set the Target drop-down to "Palm OS Device." You must also set the Connection drop-down to the serial port where the handheld is connected, and select an appropriate Baud Rate for the connection. Figure 5-21 shows a proper configuration for connecting to a handheld hooked up to COM1.

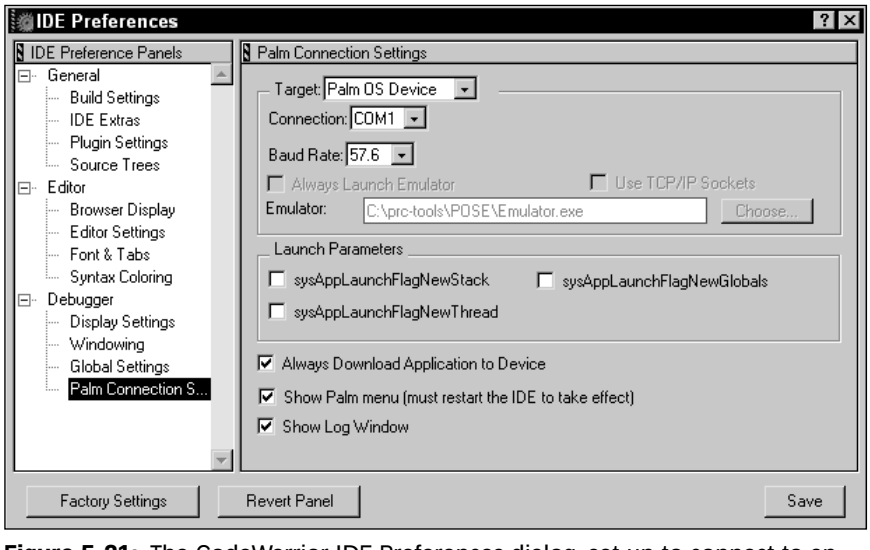


Figure 5-21: The CodeWarrior IDE Preferences dialog, set up to connect to an actual Palm OS handheld

Running the debugger

To start debugging, select the Project➪Debug menu command, or press F5. If you are debugging in POSE, make sure POSE is already running before you start the debugging session, or CodeWarrior will complain and display an error dialog.

One more step is necessary to debug on an actual handheld. When you run the Debug command, CodeWarrior shows you the dialog shown in Figure 5-22, prompting you to put the handheld into console mode with a special Graffiti shortcut sequence.

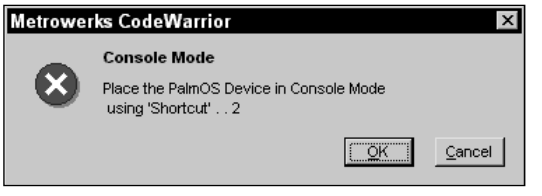


Figure 5-22: The CodeWarrior console mode prompt dialog

Enter the console mode debugging shortcut on the handheld, and then click OK to start debugging. See the "Using Developer Graffiti Shortcuts" section later in this chapter for more details about the console mode debugging shortcut.

If the application you want to debug is already running in POSE or on the connected handheld, CodeWarrior can have trouble downloading the application to the device. To avoid this problem, start a different application in POSE or on the handheld before starting the debugger.

Controlling the debugger

When the debugger finishes downloading the application to POSE or to an actual handheld, CodeWarrior stops at the first line of code in the application and displays a debugging window, shown in Figure 5-23.

The buttons across the top of the debugging window, from left to right, have the following functions, also accessible via shortcut keys and menu commands:

- **Run:** Runs application code until it hits a breakpoint or an error. This button is disabled when the application is not stopped. The Run command is also available from the menus as Project➪Run, or by pressing F5.
- **Stop:** Stops execution of application code. This button is disabled for debugging Palm OS applications, and is useful only if CodeWarrior is debugging applications for other platforms.
- **Kill:** Kills the debugger and soft-resets POSE or the attached handheld. Use this command to end a debugging session. The Kill command is also available from the Debug➪Kill menu command, or by pressing Shift+F5.

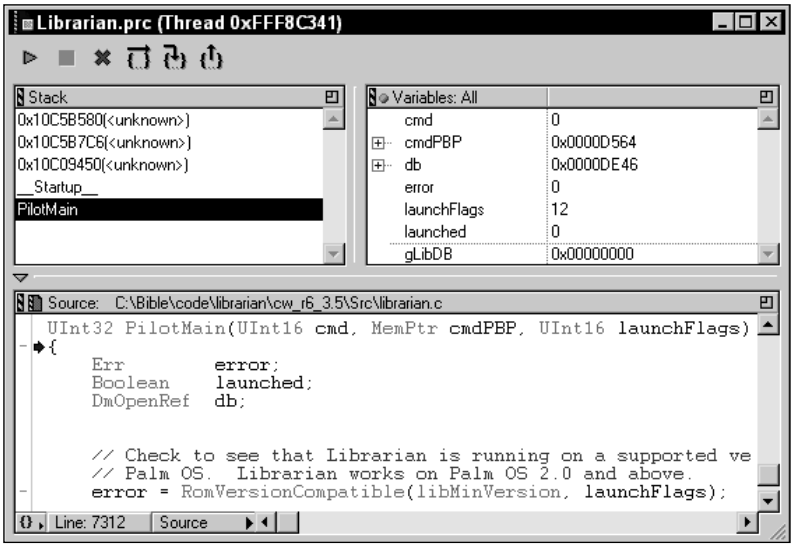


Figure 5-23: CodeWarrior's debugging window

- **Step Over:** Executes the next line of code, stepping over subroutine calls and remaining in the current routine. Step Over is also available as Debug ➪ Step Over and by pressing F10.
- **Step Into:** Executes the next line of code, stepping into subroutines. Step Into is also in the menu command Debug ➪ Step Into, as well as being accessible by pressing F11.
- **Step Out:** Executes the rest of the code in the current routine, and then stops again in the current routine's caller. You can also get to the Step Out command by using the Debug ➪ Step Out menu command, or by pressing Shift+F11.

The debugging window also has a Stack section, which shows the call stack for the application. Selecting a routine from the Stack list displays the code surrounding that particular subroutine call, allowing you to trace which routines called each other to get to the current breakpoint. A Variables list shows you values for all the variables that are currently in scope in the application.

Debugging with GDB

The GNU Debugger (GDB) is part of the GNU PRC-Tools distribution, and it presents a text-based interface for debugging at the source level. GDB runs from the command line in Windows or Unix.

Before you can use GDB to debug a Palm OS application, you must compile and link the application using the `-g` flag to include debugging symbols in the application.

Use the following steps to set up a connection between POSE and GDB:

1. Start POSE and install your application.
2. Start GDB, passing it the name of your application's linked file as a command-line argument. The linked file should have the same name as the executable, minus the `.prc` extension. For example, the following command line starts GDB for debugging a project called `myapp`:

   ```
   m68k-palmos-gdb myapp
   ```

3. Once GDB is running, it will display a prompt: `(gdb)`. At the prompt, enter the command `target pilot localhost:2000` to connect GDB to POSE:

   ```
   (gdb) target pilot localhost:2000
   ```

4. Start the application in POSE.
5. GDB stops the application at the first line of executable code, ready for debugging.

Be sure to use the `m68k-palmos-gdb` version of GDB, not the ordinary `gdb` version that comes with the Cygnus GNU tools or that comes standard on many Unix systems. The `m68k-palmos-gdb` executable has been specially compiled to work with a linked Palm OS symbol file, and you will be very frustrated if you try to use `gdb`, because it does not understand Palm OS files and will simply return an error when used with Palm OS applications.

GDB has been around for a while in Unix programming circles, so it has a lot of features and commands. You can use the `help` command to get comprehensive help about all of GDB's many features. Here are a few basic commands to get you started:

- `continue`: Continues execution of the program until it hits a breakpoint, causes an error, or exits. You may shorten this command to `cont`.
- `next`: Executes the next line of code, stepping over subroutine calls.
- `step`: Executes the next line of code, stepping into subroutine calls.
- `break [breakpoint]`: Sets a breakpoint at the specified function or line number. To set a break at the beginning of a function, give `break` the function name:

  ```
  break StartApplication
  ```

  To set a break at a line number, give `break` the file name and line number to break at:

  ```
  break myapp.c:42
  ```

- `list`: By itself, lists ten lines of code surrounding the current execution point. You can also pass a line number in the current file, a line file name and line number combination (`myapp.c:42`, just like `break`, in the previous example), or a function name to list code near the start of a function.
- `print expression [, expression2 [... , expressionN]]`. Prints variable values in the current scope. For example, you could use the `print` command as follows:

  ```
  print thisStructure.var, myString[4], *pointer
  ```

- `backtrace`. Prints a stack crawl, showing all the functions on the stack and their arguments. You can also pass the parameter `full` to the `backtrace` command to print out all the values of local variables at each step of the stack crawl.
- `quit`. Exits GDB. If the application is still running in POSE, GDB prompts you before quitting; if you decide to quit anyway, GDB soft-resets POSE.

Figure 5-24 shows a short debugging session with GDB.

```
[c:\bible\code\librarian\gcc]m68k-palmos-gdb librarian
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin32 --target=m68k-palmos"...
(gdb) target pilot localhost:2000
Remote debugging under PalmOS using localhost:2000
Waiting... (Press Ctrl-C to connect to halted machine)

Program received signal SIGTRAP, Trace/breakpoint trap.
0x2b840 in PilotMain (cmd=0, cmdPBP=0x0, launchFlags=142) at librarian.c:7290
7290    }
(gdb) break EditFormInit
Breakpoint 1 at 0x2779c: file librarian.c, line 2352.
(gdb) cont
Continuing.

Breakpoint 1, EditFormInit (form=0x1e9e) at librarian.c:2352
2352            curFont = FntSetFont(stdFont);
(gdb) backtrace
#0  EditFormInit (form=0x1e9e) at librarian.c:2352
#1  0x27416 in EditFormHandleEvent (event=0x1917c) at librarian.c:2075
#2  0x10c58550 in ?? ()
#3  0x2b82a in EventLoop () at librarian.c:7287
#4  0x2b8fe in PilotMain (cmd=0, cmdPBP=0x0, launchFlags=142)
    at librarian.c:7356
#5  0x25c1a in start ()
(gdb) list
2347            TablePtr  table;
2348            FontID    curFont;
2349            RectangleType  bounds;
2350
2351
2352            curFont = FntSetFont(stdFont);
2353
2354            // Retrieve the label width for the edit table.
2355            gEditLabelColumnWidth = EditFormGetLabelWidth();
2356
(gdb) next
2355            gEditLabelColumnWidth = EditFormGetLabelWidth();
(gdb) print curFont
$1 = stdFont
(gdb) quit
The program is running.  Exit anyway? (y or n) y

[c:\bible\code\librarian\gcc]
```

Figure 5-24: Debugging a Palm OS application with GDB

Resetting a Palm OS Handheld

When debugging applications on an actual Palm OS handheld, you will need to reset the handheld from time to time. There are a number of different ways to reset a handheld, depending on the situation:

- **Soft reset.** A soft reset clears the dynamic memory on the device, leaving the storage memory alone, so applications and data on the device remain intact. You can perform a soft reset by inserting a narrow blunt object, like the end of an unfolded paper clip, into the hole on the back of the handheld to press the reset button.
- **Hard reset.** A hard reset wipes out everything in RAM, both dynamic and storage. To perform a hard reset, press the reset button while holding down the power button. The system will ask you to confirm whether you want to erase all data on the device before proceeding.
- **No-notification reset.** This kind of reset prevents the system from sending the `sysAppLaunchCmdSystemReset` launch code to all the applications on the device. If you have a bug in the **PilotMain** routine of your application, it is possible for the application to crash the system before it even finishes a reset, sending the device into a vicious cycle of continuous resets. Pressing the reset button while holding the scroll up hardware button allows you to reset the device without the system's sending a launch code to the broken application, allowing you to delete the offending program without resorting to a hard reset.

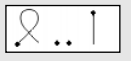
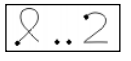
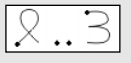
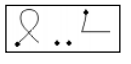
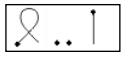
Using Developer Graffiti Shortcuts

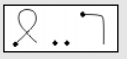
The Palm OS has a number of developer shortcuts built into it, which allow you to perform a number of useful tasks from within any application. The shortcuts are all accessed by writing the Graffiti shortcut character (a cursive lowercase "L") in the Graffiti area, followed by two taps (a period character) and the shortcut code. Table 5-3 describes each available shortcut code.

Some of these shortcuts can leave the serial port open, draining the handheld's batteries. Others may cause data loss or damage to existing data. Use these shortcuts carefully.

Though you do not need to write these shortcuts in an actual text field, the feedback you get from doing so is very useful. Since the Find dialog contains a text field, and it is available from any application, tapping the Find silk-screened button before entering a shortcut is a quick and easy way to see what you are writing when using these shortcuts.

Table 5-3
Special Developer Graffiti Shortcuts

Graffiti	*Shortcut*	*Description*
	.1	Enters debugger mode. The handheld opens the serial port and waits for a connection from a low-level debugger, such as the Palm Debugger that ships with the Palm OS SDK. You must perform a soft reset to exit this mode and close the serial port.
	.2	Enters console mode. The handheld opens the serial port and waits for a connection from a high-level debugger, such as CodeWarrior. You must perform a soft reset to exit this mode and close the serial port.
	.3	Shuts off the auto-off feature. When this shortcut is made, the device will no longer shut itself off automatically, regardless of the current auto-off time set in the system Prefs application. You must perform a soft reset to enable auto-off again.
	.4	Briefly displays the user name assigned to the device, along with a number that HotSync uses internally to identify the user
	.5	Erases the user name and number assigned to the device. At the start of the next HotSync operation, the HotSync manager will treat the device as a brand new handheld that has never been synchronized. This will cause duplicate records to be copied to the handheld from the desktop during the synchronization; to prevent this, you will need to perform a hard reset.
	.6	Displays the date and time at which the ROM was built in the text field that has the focus.
	.7	Toggles between alkaline and NiCd modes for keeping track of remaining battery voltage, which is supposed to change when low battery alerts are displayed to the user. Historically, the NiCd mode has never been particularly accurate; you should probably leave this shortcut alone.
	.i	Temporarily enables the device to receive incoming IR transmissions, even if Beam Receive in the system Preferences application is currently set to "Off."
	.s	Toggles serial IR mode. When active, serial IR mode causes all IR calls to go to the handheld's serial port, instead, which can be very useful for debugging low-level infrared code.

Graffiti	*Shortcut*	*Description*
	.t	Toggles loopback mode for the Exchange Manager. When active, the loopback mode causes all IR beaming operations to loop back to the device, allowing you to test some beaming functions without using a second handheld.

Using the Palm OS Error Manager

The Palm OS error manager is an API that provides mechanisms for displaying errors that might come up during application development. Error manager macros are conditionally compiled, so they only become part of your compiled application if you set certain special constants in your code. During development, the macros are there to help you debug the application. Once you have finished development, you can easily build a version of the application that does not include the error-checking code, resulting in a smaller, faster executable.

There are three macros available in the error manager for displaying runtime errors: **ErrDisplay**, **ErrFatalDisplayIf**, and **ErrNonFatalDisplayIf**. The **ErrDisplay** macro always displays an error dialog, and the other two macros display a dialog only if their first argument resolves to the value `true`. Error dialogs displayed by these macros also include the line number and file name of the source code that called the error macro, making it easier to find where the error occurred.

The `ERROR_CHECK_LEVEL` compiler defined controls which error macros the compiler includes in the compiled application. Table 5-4 shows the constants defined in the Palm OS for each error level, and what each error level means.

Table 5-4
ERROR_CHECK_LEVEL Constants

Constant	*Value*	*Description*
`ERROR_CHECK_NONE`	`0`	The compiler does not compile in any error macros.
`ERROR_CHECK_PARTIAL`	`1`	The compiler compiles only fatal error macros (ErrDisplay and ErrFatalDisplayIf) into the application.
`ERROR_CHECK_FULL`	`2`	The compiler compiles in all error macros.

When you are developing the application, an error-checking value of ERROR_CHECK_FULL is appropriate to catch all the bugs you possibly can. Set ERROR_CHECK_LEVEL to ERROR_CHECK_PARTIAL for alpha and beta tests; the non-fatal errors produced by **ErrNonFatalDisplayIf** should probably have been handled at this point in the development cycle, anyway, or at the very least should already be known to the developer. The ERROR_CHECK_NONE level is appropriate for a final released product.

You can use the **ErrDisplay** macro to always display an error dialog. Use the following syntax to call **ErrDisplay**:

```
ErrDisplay("Insert error message here");
```

The **ErrFatalDisplayIf** and **ErrNonFatalDisplayIf** macros take two arguments, the first of which should resolve to a Boolean value. Only if the first argument is true will these two macros display the error message indicated by their second argument. In general, the first argument will be an inline statement of some kind, which makes for neat and tidy error-checking code that is still removed from compilation when producing a final release build. For example, the following snippet calls a hypothetical function called **MyFunc** and, if its return value is greater than 4, generates a fatal error dialog:

```
UInt16 result = MyFunc();
ErrFatalDisplayIf(result > 4, "Illegal result from MyFunc");
```

When **ErrDisplay** or **ErrFatalDisplayIf** displays an error dialog, the user can clear the error only by tapping the supplied **Reset** button, causing a soft reset of the handheld. The **ErrNonFatalDisplayIf** macro allows the user to continue execution of the program, so **ErrNonFatalDisplayIf** should be used only in situations where you want to check if a certain condition exists but that condition will not prevent the application from continuing (more or less) normally.

Summary

In this chapter, you were given an overview of the most important debugging tools available for Palm OS development. After reading this chapter, you should understand the following:

- The Palm OS Emulator is a fantastic tool for debugging Palm OS applications without going through the slow and painful process of repeatedly installing a program to an actual handheld.
- POSE does have its limitations, particularly in the areas of IR support and accurately representing execution speed on a real device, so you should still make a final test pass for your application on an actual Palm OS handheld.
- POSE Gremlins let you give your application a really good working over, randomly pounding parts of the program that might not be sufficiently tested using more structured testing techniques.

- Source-level debugging with either CodeWarrior or GDB allows you to easily find which line (or lines) of code in your application is causing a particular bug.
- There are multiple ways to reset a Palm OS handheld, including soft resets, hard resets, and no-notification resets.
- Special developer Graffiti shortcuts give you access to some more obscure settings of the operating system and hardware.
- Adding error manager macros to your application can be a useful debugging tool during the program's development, and the macros will not weigh down the application when you release a final version to the public.

✦ ✦ ✦

CHAPTER 6

Creating and Understanding Resources

In This Chapter

Following Palm OS user interface guidelines

Creating resources with Constructor

Compiling resources with PilRC

As you may recall from Chapter 2, "Understanding the Palm OS," resources fall into three categories: system, catalog, and project. Because the compiler takes care of creating system resources for you, like the application's executable code, you need to create only the catalog and project resources to define things such as forms and application icons. This chapter explains how to create most of these resources, using the Constructor tool from CodeWarrior and PilRC from the GNU PRC-Tools.

Before delving into the mechanics of creating resources, though, a discussion about how those resources should be used is in order. Palm Computing provides an extensive list of user interface guidelines, which provide a framework for making applications that are best suited to a handheld device. The Palm OS guidelines also ensure that applications look and operate the same way as the built-in Palm OS applications. Emulating the way the standard applications work makes for an application that users can immediately begin using with little or no instruction, because they will already be familiar with the interface.

Following Palm OS User Interface Guidelines

Although the philosophy behind the Palm Computing platform is responsible for many of Palm's user interface guidelines, many of these guidelines apply equally well to any program running on any platform. User interface is the art of striking a balance between screen space, utility, and ease of use that will make an application useful to the largest number of people.

The Palm OS user interface guidelines dictate three basic rules:

- ✦ Programs must be fast.
- ✦ Frequently used functions should be more accessible than infrequently used functions.
- ✦ Programs should be easy to use.

Making Fast Applications

A fast application is more than just writing good algorithms and optimizing your code. Good handheld applications have an efficient interface that makes it possible to use them quickly. Navigating through different screens, activating commands, and retrieving data should require very little time. For example, when the user enters Graffiti strokes in the list view of the built-in Memo Pad application, the application automatically creates a brand-new memo containing the newly entered text; there is no need for the user to explicitly tap the New button to create a new memo. Looking up information in all the standard Palm applications is just as simple, often requiring only one hand, because the hardware buttons can scroll through the most useful information without your having to use the stylus.

Always try to reduce the number of taps required to perform a particular action. The best way to accomplish this is through intelligent selection of controls; some user interface elements are faster at some tasks than others. Keep the following in mind when designing the interface in your application:

- ✦ Buttons provide the quickest access to a program's functions. They occupy a fair amount of screen real estate, though, and having too many buttons on a form is inefficient because it takes more time to visually search the screen for the correct button.
- ✦ Because push buttons are faster than pop-up lists, requiring only one tap instead of two to make a selection, use push buttons when you have enough screen space for them.
- ✦ A check box is a fast way to change a setting that can be turned on or off, because it requires only a single tap to toggle the box. Unless you need to fit a lot of controls into a single screen, avoid using pop-up lists that contain only two items; a single check box will perform the same action in half the number of taps.
- ✦ A pop-up list is faster than Graffiti or on-screen keyboard input. If your application can offer choices from a list instead of requiring the user to enter text manually, the user will spend much less time entering data.
- ✦ Pop-up lists that contain a lot of elements are slower to use than short lists. Just like having too many buttons on one form, too many list items are hard to take in at a glance. Also, with enough list items, the user may have to scroll the list to find the right item, requiring yet another tap.

Highlighting Frequently Used Functions

Commands in an application that the user will use frequently should be easier to access than those that the user will need only occasionally. Not only should a frequently used command be easy to spot on a form, but also the physical actions the user must perform to activate the command need to be fast.

Something the user is likely to do several times in an hour, such as checking today's events in the Date Book, should be accessible with a single tap. If a particular action may be performed a number of times in a single day, like adding an item to the To Do List, a couple taps or a little bit of text entry is appropriate. Adding a repeating event to the Date Book or other things that the user might need only a few times every week can require several taps or an entire dialog devoted to the task.

The following tips can help you match accessibility to frequency of use:

- Important data that you expect a user to look at most of the time should be the first thing displayed when the application starts. The built-in Date Book is a good example of this because it shows today's events when it first opens. Just by launching the application, a user can check what is going on without even having to use the stylus.
- Keep to a minimum the need to flip through different screens. The more navigation required to get between particular views of the application's data, the more time it will take the user to retrieve that data. Not only is it important to keep this in mind when designing user interface, but it is also vital when determining how your application stores its data. Structure the data so that retrieving records will not require a lot of navigation. Chapter 13, "Manipulating Records," deals with data structure concerns more fully.
- Use command buttons for the most common tasks that the user must perform. All four of the major built-in applications have a New button for creating records, because quickly adding a new record is a vital part of these programs. Command buttons are also perfect for launching frequently used dialog boxes, such as the Details dialog box from the Date Book and To Do List applications.
- Avoid using a dialog box if you don't really need one. Notice that the built-in applications do not prompt the user for a new record's category when the user taps the New button. Instead, the applications assign a reasonable default category to the new item, usually based on the category currently displayed in the list view, and immediately allow the user to start entering data. If the user wants to change the category, another tap on the Details button, or the category pop-up list, allows the user to perform this task. Try to anticipate how someone will use an application, and design it accordingly.
- Except for very infrequently used features, try to avoid displaying a dialog box from another dialog box. Digging back out of nested dialog boxes slows down application use.

Designing for Ease of Use

Ideally, a Palm OS application should be usable with little or no instruction. Within five minutes of picking up a Palm OS device for the first time, a user ought to be able to perform basic tasks and freely navigate between applications. More advanced commands should still be easy to find and use, but they should not obscure the most basic functions of a program.

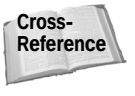
Consistency is key to making a user interface that new users can learn easily. Memorizing a completely new way of interacting with the device for each different application is difficult and time-consuming. When every Palm OS application operates in a similar fashion, the skills learned from interacting with one application will easily apply to any other program.

One of the best ways to ensure that your Palm OS applications are familiar to users, both new and experienced, is to emulate the interface design of the built-in applications. Study how the standard applications display data and offer choices, and design your own user interface to parallel the placement of controls in the built-in applications. Imitation is not only the sincerest form of flattery, but it will also make your applications easier to use.

These suggestions will help you make applications that are easy to use:

- In the title bars of forms, let the user know what application is running, and what view is currently displayed. This kind of context is necessary to prevent the user from becoming lost while navigating between screens and different applications.
- Use clearly labeled buttons for the most important commands. Not only are buttons quick to use, they catch the eye like no other user interface element, making them ideal for stressing key actions to perform. Make sure the label on the button adequately describes its function, though. You must strike a balance between saving screen space and providing enough text to avoid confusing the user about the function of the button. One possibility for saving some screen real estate with buttons is to label them with icons instead of words. For example, a small picture of a trash can might replace a “Delete” caption. Be careful using icons, though. What may be intuitive to you may leave your application’s users hopelessly confused if they can’t make the same logical connection you have made between an icon and its function.
- Keep the design of forms and dialog boxes clean and simple. You may be tempted to put everything right up front on the screen to allow quick access to as many commands as possible. However, doing so makes the application more difficult to learn and can actually slow users down, forcing them to scan the screen in search of the proper control. A few well-placed buttons that open simple dialog boxes are preferable to trying to cram your entire interface into a single screen.

- Make sure actions are consistent throughout the application. For example, if pressing the hardware scroll buttons browses records a screenful at a time in one view, but switches between entry fields in another view, the application becomes harder to learn and use. Reduce the burden on the user's memory by making similar actions perform similar functions throughout an application.
- Navigation between different views in the application should be obvious to the user. The best way to do this is by using command buttons. The address view in the Address Book is a good example. The three buttons, Done, Edit, and New, do what most users expect they will do when tapped: return to the main list view, edit the currently displayed record, or make a new record, respectively.
- Minimize the number of steps required to perform a particular task. This not only speeds up use of the application but also reduces its complexity, making it easier for the user to remember how to perform that task.
- Always provide the standard Edit menu whenever editable text fields are present in a form or dialog box. This will ensure that the clipboard and text editing commands are always available. More important, make sure the on-screen keyboard is accessible to those users who prefer it to Graffiti text entry. On Palm OS version 2.0 and later, be sure to also provide access to the system Graffiti reference dialog box from this menu.



The standard Edit menu is detailed in Chapter 8, "Building Menus."

Maintaining Palm OS Style

There are a number of other considerations to keep in mind when designing an application to fit the expectations of users of Palm OS devices. Anyone used to the standard applications installed on such a handheld will count on certain things working in a particular way. Also, some elements of Palm OS style are necessary to ensure that your application is a good citizen and performs the way the operating system expects.

Navigating within and between applications

Users switch applications by pressing the physical application buttons or using the application launcher. If your application intercepts the hardware buttons for its own purposes (a common occurrence in game programs), be sure to leave the silk-screened application launcher button alone. Otherwise, there will be no way to switch out of your application without resetting the device.

Depending on what information your application must display, it may be appropriate to have two different views of its data — a list view and an edit view. The Memo Pad application operates in this fashion. Figure 6-1 shows the list and edit views of the Memo Pad application. The list view shows some or all of the application's records

a screenful at a time, providing some useful information at a glance. Selecting a record from the list view starts an edit view, where a user may change the data in an individual record. Tapping the Done button in the edit view returns the user to the list view. The Details button in the edit view provides a way to change settings that affect the entire record, such as its category or whether it should be marked private.

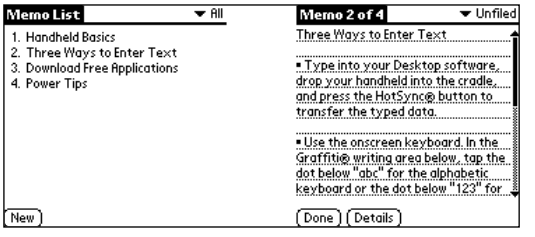
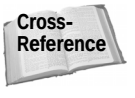


Figure 6-1: The Memo Pad application provides two separate views, list (left) and edit (right).

Even applications such as the Date Book and To Do List, which both display and allow editing of a record in the same screen, have a Details button that operates in similar fashion. Instead of going to an edit view when tapping on a record in these applications, the program enters an editing mode. When in edit mode, the Details button offers the same record-level settings changes that are possible in the Memo Pad and Address Book applications.

If appropriate for your application, consider implementing user-defined categories to allow users to organize the program's data. A pop-up list in the upper right corner of the screen provides category switching in the standard applications. The Palm OS provides facilities to easily manage categories, including a dialog box to allow adding, renaming, and removing categories.



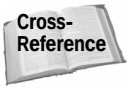
See Chapter 12, "Manipulating Records," for more information about implementing categories.

Repeatedly pressing the hardware application button assigned to one of the built-in applications causes the application to switch between displaying different categories of data. If your application is one that a user might consider assigning to one of the hardware application buttons by using the Preferences application, copying this behavior is a good idea.

When providing a number of text entry fields on one screen, be sure your application handles the next field and previous field Graffiti strokes, pictured in Figure 6-2. You may identify these two characters in code with the constants `nextFieldChr` and `prevFieldChr`. These strokes save time during data entry because the user can just enter the appropriate navigation stroke instead of having to lift the stylus from the Graffiti area and tap in another field. The built-in Address Book's edit view is a good example of using Graffiti strokes to switch fields. In the edit view, you can jump from field to field without ever moving the stylus out of the Graffiti entry area.

Figure 6-2: The next field (left) and previous field (right) Graffiti strokes allow for rapid data entry on forms containing more than one text field.

Because of the small size of the screen, users will often need to scroll the display, both to display many records from a database and to display the information in long records. For on-screen scrolling, provide either a pair of arrow-shaped repeating buttons or a scroll bar. Also, handle the `pageUpChr` and `pageDownChr` key events to allow scrolling via the hardware scroll buttons. The hardware buttons should scroll data a page at a time. Although Palm OS user interface guidelines state that repeating buttons used for scrolling should scroll data a line at a time, both the Address Book and To Do List applications scroll a page at a time when the on-screen buttons are used. So, this is more a matter of what works well with your application's data than a hard-and-fast rule.



Scrolling is a complex topic, covered in detail in Chapter 9, "Programming User Interface Elements."

Designing screen and dialog box layout

The screens and dialog boxes in a standard Palm OS program should follow certain layout rules to achieve a consistent look between applications. The screen from the built-in Memo Pad application shown in Figure 6-3 is a fine example of how to design a screen for a Palm OS application.

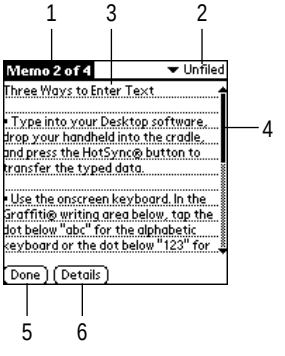
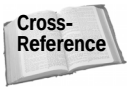


Figure 6-3: Screen layout in the Memo Pad application

The numbers in the following guidelines refer to the Memo Pad screen shown in Figure 6-3:

1. Each screen should have a title bar. If there is enough room, include the name of the application and the name of the current view. The title bar's text should not only let users know what application they are looking at, but should also provide context within the program. Carefully worded title bars prevent users from becoming lost in applications that contain many different views.

2. If the application uses categories to organize records, put a category pop-up list in the upper-right corner. In a screen that displays multiple records, this pop-up should change the view to display records from different categories; in a screen that displays a single record, the category pop-up should change the category of the currently displayed record.
3. Use the whole screen, all the way out to the edge. There is little enough screen real estate on a Palm OS handheld that wasting even a couple pixels to draw a border will make it difficult to fit user interface elements on the screen without crowding them. The hardware case surrounding the screen makes a perfectly suitable frame.
4. Whenever possible, use the standard Palm OS user interface resources. Unless your application has unique interface requirements (for example, games tend to have special interface elements), sticking with the default buttons, fields, and other parts of the default Palm OS provides users with a familiar environment. Familiarity makes the program easier to use.
5. Command buttons should be lined up along the bottom of the screen, left-justified. In particular, any buttons used to navigate between screens in an application will be quicker to use if they are all in a consistent location because users will be able to tap in the same region of the screen without moving the stylus very far.
6. In buttons and other places where text is contained in a border, be sure to leave at least one pixel above and one pixel below the height of the text. Many letters in the Palm OS fonts are difficult to read when a line touches them.



Palm Computing also provides recommended settings for individual user interface elements. These guidelines are detailed with their appropriate resources in Chapter 9, "Programming User Interface Elements."

Dialog boxes serve a different function from screens in a Palm OS program. A dialog box provides a place for the application to query the user for input, or for the user to change record and application settings. The dialog box from the To Do List application shown in Figure 6-4 demonstrates good dialog box design principles.

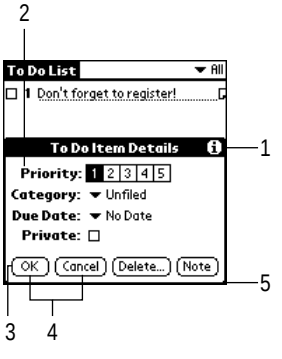
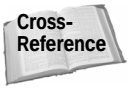


Figure 6-4: Dialog box layout from the To Do List application

Numbered guidelines below refer to the To Do List dialog box shown in Figure 6-4:

1. The Palm OS provides facilities for online help in dialog boxes. Tips, accessed from the "i" icon in the upper right corner of a dialog box, are an easy way to provide users with details about what all the interface elements in a dialog box actually do.



Adding tips to a dialog box is easy, and requires no code at all; simply create a string resource and associate it with the dialog box form. Chapter 7, "Building Forms," tells you how.

2. Labels in a dialog box should be right-justified, followed by left-justified elements that may be edited. Use bold text for the labels and nonbold text for editable items to visually differentiate them.
3. Unlike screens, which have no border, dialog boxes need three pixels of space between buttons and the edge of the dialog box.
4. Particularly in alerts that prompt the user to make a decision, be sure to put positive response buttons on the left and negative responses on the right. Maintaining this kind of consistency speeds use of the program and helps to prevent the user from making errors.
5. Dialog boxes should always be aligned with the bottom of the screen. Placing those that are shorter than the height of the entire screen at the bottom ensures that the title bar of the screen behind the dialog box is still visible, thereby reminding users of what application is currently running and where they are within that program.

Keeping other Palm guidelines in mind

There are a number of other considerations to keep in mind that will ensure your application looks and behaves like other Palm OS applications:

- Every application should have an icon to identify it in the launcher, as well as a short name to identify the icon. For programs running on Palm OS version 3.0 or later, you should also provide a small icon for the launcher's list view. Adding icons and icon names is covered later in this chapter.
- Palm applications contain a version resource, which the launcher in Palm OS version 3.0 and later can display from the Options ➪ Info menu command. Version resources are also useful if you wish to write multiple applications that cooperate with one another or share data. Should the data format of one of the applications change between versions, the other programs can query that application to determine its version and act accordingly. Adding version resources is also detailed later in this chapter.

- All functions of an application that require tapping the screen should be accessible with single taps. Double-clicking in a desktop environment can be difficult for some users to manage, because a mouse tends to move around a bit when its buttons are clicked, and double-tapping on a handheld can be even more difficult because the device is usually held in the off hand instead of on a stable surface. Double taps are also counterintuitive. Without explicit instruction, it is impossible to tell from looking at an application that certain commands are activated by double taps.
- Where possible, make buttons large enough to allow finger navigation. Navigating between different views in the application should be possible without the user's having to pull out the device's stylus. The buttons in the Palm OS built-in applications are a good size for this purpose.
- If a menu item or user interface element is currently disabled or not available, remove it from the screen entirely. The Palm OS does not provide any facilities for "graying out" controls and menus, and there is little enough screen space available that removing the item entirely is preferable.
- Many desktop applications duplicate commands by making them accessible from a button and from the application's menus. Avoid this kind of duplication in Palm OS applications. Not only does it increase the size of the program, but it also goes against the paradigm of highlighting frequently used functions. Important commands that the user will access regularly should be on the screen itself; less often used commands should be relegated to menus.
- Likewise, provide Graffiti command shortcuts only for those menu items that really need them. For example, cutting and pasting text are actions that need to be performed quickly, so these commands are good candidates for command shortcuts. On the other hand, an about box for an application is something the user will look at only occasionally, so displaying it does not require a shortcut.

Creating Resources with Constructor

Constructor is a graphical tool for resource creation. Its "what you see is what you get" (WYSIWYG) approach to resource editing lets you see exactly what your resources will look like during every step of their creation.

When it first opens, Constructor consists of only a title bar and a menu bar underneath it, pictured in Figure 6-5. Only the Windows version of Constructor is pictured in this book; however, the Mac OS version of Constructor is very similar in appearance.

Figure 6-5: The Constructor menu bar

Understanding Resource Forks

After saving a project on a Windows machine, you may be wondering why Constructor creates a directory called `Resource.frk` in the same directory where you chose to save the project file. Constructor does this to allow transfer of resource files between Windows and the Mac OS. On the Mac OS, Constructor stores resource information in the *resource fork* of the project file. Because Windows files do not contain a resource fork, Constructor saves an empty `.rsrc` file to mimic the Mac OS data fork, and then creates the `Resource.frk` directory and saves the resources themselves in that directory, using the same file name as the data file.

Keep in mind that if you wish to move your resource files around in Windows, you need to copy both `.rsrc` files, and the resource fork file must be in a `Resource.frk` directory in the same location as the data fork `.rsrc` file. Failure to copy both files correctly prevents Constructor from working with the files.

When opening a project file for editing, you may select either of the `.rsrc` files. Constructor is smart enough to figure out which file contains the data fork and which contains the resource fork.

Constructor organizes all the resources for a particular application in a *project*. To create a new project, select File➪New Project File, or press Ctrl+N.

Alternatively, you may open an existing project file by selecting File➪Open Project File, or by pressing Ctrl+O. Constructor prompts you for the location of the project file you wish to edit.

When you have finished making changes to the project, save your work to disk with the File➪Save menu command, or press Ctrl+S. Constructor prompts you for a location to which you would like to save the project.

Exploring the Project Window

The first window that Constructor displays when you open an existing project, or when you create a new project, is the *project window*. The project window for the Hello World application from Chapter 4 is displayed in Figure 6-6. Notice that the project window is divided into two sections, the Resource Type and Name list and the Project Settings box.

Besides listing project resources and settings, the project window also informs you which resources or settings you have changed since the last time you saved the project. Constructor indicates changed resources or settings with a small black dot to the left of the appropriate item. When you save the project, all the dots disappear.

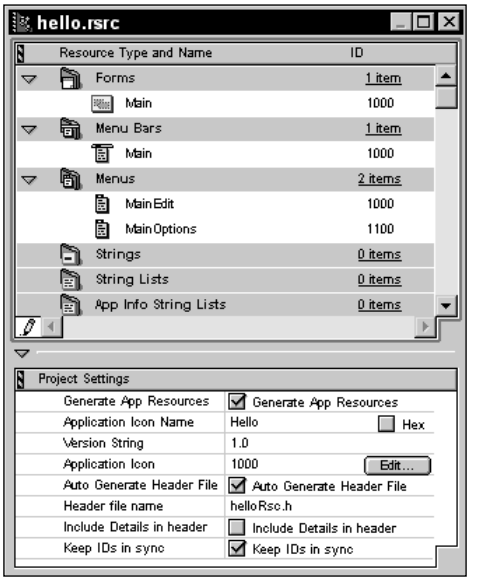


Figure 6-6: Constructor's project window

Located at the bottom of the project window, the Project Settings box allows you to change settings for the entire project. From top to bottom, the Project Settings box contains the following items:

- **Generate App Resources.** This check box, when checked, tells Constructor to generate application icon name and version resources for the project. Unchecking Generate App Resources disables the Application Icon Name and Version String fields, in case you want to generate other resource types without creating icon name and version string resources. Leaving the Generate App Resources check box unchecked can be useful if you use multiple resource files to create different sections of a large application, because generating application resources in multiple resource files will cause the compiler to complain when it encounters duplicate resources for the icon name and version number resources.
- **Application Icon Name.** This text field contains the name that will appear in the launcher next to the program's icon. The Application Icon Name field may contain up to 31 characters, but you should use only half that to prevent the icon name from overlapping other icon names in the launcher, which has very little space available for icon names in both its list and icon views. Use the Hex check box to toggle display of the icon name between hexadecimal and normal text.
- **Version String.** The version string is handy for branding the application with its current version. This field accepts up to 15 characters, and it is not limited to just digits and decimal points. Allowing non-numerical characters permits the use of interim version numbers, such as "1.2e," or of beta version numbers, such as "0.9b."

- **Default App Category.** In this field, you can specify the name of a default category that the application should be added to when the application is installed on a handheld. If this category does not already exist on the handheld, the system will create it. If this field is left blank, the application becomes part of the Main category on the handheld by default.
- **Application Icon.** This field contains the resource ID number of the application's icon, which is displayed in the launcher. If you have not created an icon for the current project, the button next to the ID number is labeled Create, and clicking the button will open a bitmap editor and allow you to create the icon. If an icon with resource ID 1000 already exists in the project, the button is labeled Edit, and clicking it will open the existing icon in a bitmap editor. Note that changing the ID number from 1000 is a bad idea; the Palm OS assumes that an application's icon has an ID of 1000, and if no icon with that ID exists, the OS displays a blank spot in the launcher instead of an icon.
- **Auto Generate Header File.** Checking this check box tells Constructor to automatically create a header file containing `#define` statements that map constant definitions to resource ID numbers. Using resource constants is much easier than trying to remember the four-digit number assigned to each resource in an application, so keeping this option checked will save you a lot of headaches later on. The following two lines in the Project Settings box do not do anything if Auto Generate Header File is not checked.
- **Header file name.** This text field contains the name of the header file to generate if the Auto Generate Header File box is checked. If you do not specify a header file name here, Constructor will provide one for you. On Windows, this name is composed of the name of the project file with `_res.h` appended to it; on the Mac OS, the name consists of just the name of the project file with `.h` appended to it. Likewise, if the file name you enter here does not have an `.h` extension, Constructor will tack `.h` onto the end of the name you entered when you save the project.
- **Include Details in header.** Checking this box will add comments to the automatically generated header file, describing individual properties of each of the resources listed in the header. This option is strictly optional because it has no bearing at all on how CodeWarrior will compile the resources.
- **Keep IDs in sync.** When this box is checked, Constructor will maintain consistent resource IDs. In particular, Constructor will keep form resource IDs the same as its own internal form IDs, and when you change the resource ID of a form, Constructor will change all object IDs of objects in that form to match the beginning of the form's ID. In general, you should keep this box checked unless you really want mismatched object and form IDs.

The Resource Type and Name list shows all the project resources in an application, including forms, alerts, strings, menus, bitmaps, and other resources. Each gray bar represents a different type of resource. Constructor lists underneath that category's gray bar all of the resources in the project that belong to a particular category.

You create new project resources by selecting one of the gray bars and then selecting the Edit➪New *Type* Resource menu command, where *Type* represents the kind of resource you have selected. A quicker way is to select the appropriate bar and press Ctrl+K.

Alternatively, you may also select Edit➪New *Type* Resource, or press Ctrl+R, to open the Create New Resource dialog box. The Create New Resource dialog box is pictured in Figure 6-7. Clicking the Create button in this dialog box makes a new resource with the properties you set in the dialog box.

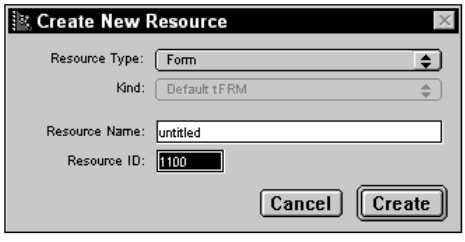


Figure 6-7: The Create New Resource dialog box

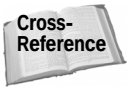
The two commands on the Edit menu for creating new resources look almost identical. Use the first, Edit➪New *Type* Resource, with the shortcut key Ctrl+K, to quickly create a new resource based on the currently selected resource type in the project window. The second command, Edit➪New *Type* Resource, with the shortcut key Ctrl+R, simply opens the Create New Resource dialog box.

To confuse matters further, when no resource type is selected in the project window, both commands open the Create New Resource dialog box. Either command is perfectly valid, but the Ctrl+K method is much quicker to use.

You can change the name or resource ID of any resource in the project window by clicking once on its name or ID, and then typing the new name or ID number. To edit a particular resource, either double-click it, or select it and press Enter. An editor window appropriate to the type of resource appears. You may also use the Edit➪Edit Resource command to open the editor for a selected resource.

Delete resources by selecting them and choosing Edit➪Clear Resource or pressing either the Delete or Backspace key.

Most of the resources you can create from the project window are detailed in the following sections. Forms and menus are complex enough that they merit their own chapters.



For detailed information about making form and menu resources, see Chapter 7, “Building Forms,” and Chapter 8, “Building Menus.”

Strings

A string resource simply contains a string of text characters. String resources are the usual way to provide online help in your applications, and they can also serve to contain default values for text fields or other miscellaneous text that remains constant between different executions of an application. Figure 6-8 shows the string editor window.

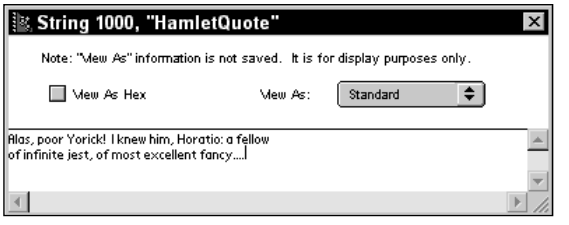


Figure 6-8: The string editor window

You may edit the string's text in the lower half of the editor window. The View As Hex check box converts the contents of the window to hexadecimal, and the View As pop-up list allows you to select the font in which to display the text. Note that these two "View As" controls have no effect on the actual resource that Constructor generates; they are for display purposes only.

String lists

A string list is an indexed list of text strings with a specific prefix string. Figure 6-9 shows the string list editor window, which has the same "View As" capabilities as the string editor.

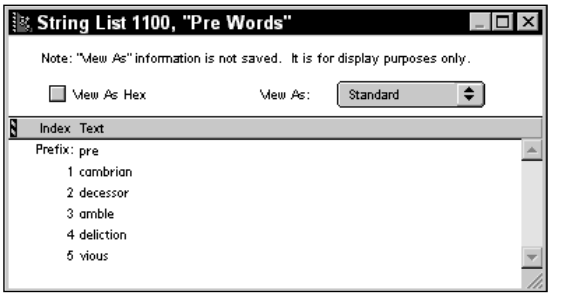


Figure 6-9: The string list editor window

You can use the **SysStringByIndex** API function to access string list resources from your application. **SysStringByIndex** returns the string list's prefix string, concatenated with a specific string from the list.

Constructor starts listing string list indices at 1, but the SysStringByIndex function uses 0 to indicate the first string in the list. Keep this in mind to avoid off-by-one errors in your code.

To enter the prefix string, click to the right of "Prefix:" in the window and type the prefix string. Note that a prefix string is not required, and in some cases, not desired because the **SysStringByIndex** function tacks the prefix string to the front of every string it returns.

To add individual strings to the list, select Edit ➪ New String, or press Ctrl+K. You can also change the order of items in the list by clicking the index number of a string you wish to move and dragging it to its new location in the list. To delete a particular string, select it, and then select Edit ➪ Clear String or press either the Delete or Backspace key.

App info string lists

An app info string list holds the initial categories for an application. The **CategoryInitialize** function uses this information to set default values for an application's categories. The Palm OS category manager expects to find static category names at the top of an app info string list. If your application will have categories that the user cannot edit, such as the "Unfiled" category common to most of the Palm built-in applications, be sure to put these category names at the beginning of the list.

Figure 6-10 shows the app info string editor window, which is quite similar to the string list editor window.

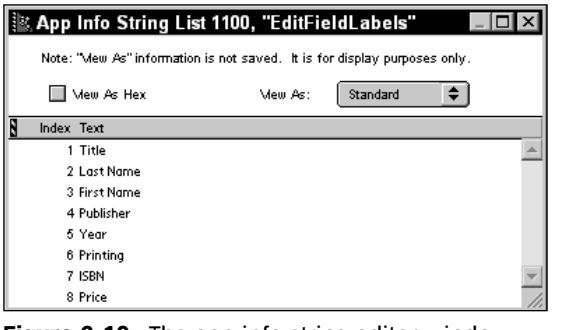


Figure 6-10: The app info string editor window

Except for the lack of a prefix item, you can add, edit, and delete items in an app info string list using the same techniques listed above for regular string lists.

Constructor lets you enter as much text as you like for each category, but the Palm OS allows only 15 characters for a category name. Be sure not to exceed this limit when entering category names.

Alerts

The alert editor window, pictured in Figure 6-11, provides you with everything you need to make alert dialog boxes. It also shows a preview of what the alert will look like in the finished application.

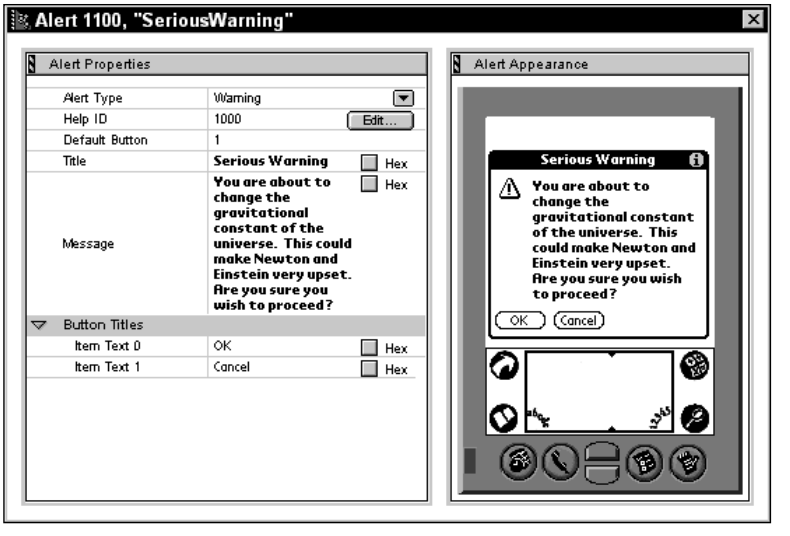


Figure 6-11: The alert editor window

From top to bottom, the alert editor window contains the following items:

- **Alert Type.** This pop-up list allows you to select one of the four alert types: information, confirmation, warning, or error.
- **Help ID.** This field contains the resource ID of a string resource that will serve as the online help for this alert. If you have not assigned a string resource to this alert yet, the resource ID displayed is 0, and the Create button to the right of the ID is disabled. If you set Help ID to a value that does not correspond to any existing string resource, the Create button becomes enabled, and clicking it will open a string editor window to allow you to create a new string resource. If you set the Help ID to the resource ID of an existing string resource, the button's caption changes to Edit, and clicking the button opens that string resource in a string editor window.

- **Default Button.** You can set the default button for the alert dialog box in this field. When dismissing an alert, usually because of switching from the displayed alert's application to another application, the Palm OS simulates a tap on the default button, allowing you to execute appropriate default code on the way out of the dialog box. The Default Button field is disabled if the alert contains only one button.
- **Title.** Enter in this field the text to display in the alert's title bar.
- **Message.** Whatever message the alert should display goes into this field.
- **Button Titles.** Clicking on the arrow in the left of this item hides or displays the list of buttons for the alert resource. Each button on the form is listed as Item Text *n*, where *n* is the index number of the button; an index of 0 represents the leftmost button. To add more buttons to the alert, select an existing button title, and then choose Edit ➪ New Button Title or press Ctrl+K. A new button title appears below the selected button title. You can also insert a new button before the first button by selecting the gray Button Titles bar and then creating a new button as described above. You may delete a selected button by selecting the Edit ➪ Clear Button Title command, or by pressing Backspace. Constructor will allow you to add more buttons than the Palm OS can actually display in an alert at run time, so be sure to keep the number of buttons
reasonable, say only three or four, with short captions.

Unlike other places in the Windows version of Constructor, where pressing Delete will delete an item, Delete does nothing at all. Use the Backspace key instead if you want to use a keyboard shortcut instead of selecting the menu option.

Multibit icons

The multibit icon editor window, pictured in Figure 6-12, is the place to create and edit both black-and-white and grayscale icons.

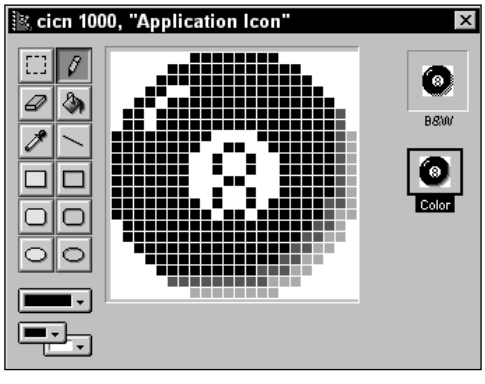


Figure 6-12: The multibit icon editor window

A multibit icon actually contains two different icons: one in black and white, and another in grayscale. Prior to version 3.0, the Palm OS does not support grayscale icons. However, if you include a black and white version in a multibit icon, earlier versions of the OS can read the black and white part without difficulty. The editor displays a preview of both the black and white icon and the grayscale icon, which are labeled B&W and Color, respectively. Select the appropriate small preview image to edit it in the canvas, or drawing area, of the editor window.

Starting with Palm OS 3.5, the operating system supports color icons in the launcher application. To provide backward compatibility with older versions of the Palm OS that do not support color, the version of Constructor that ships with the Palm OS 3.5 SDK can be used to create app icon families, which hold multiple copies of a single image at different color depths. If your application is intended to run on Palm OS 3.5, you can skip creating normal or multibit icons entirely, and instead create a number of bitmaps and add them to an app icon family. For more details of this process, see "App icon families," later in this chapter.

Every application should have two multibit icons to display in the launcher, both a large icon for the regular icon view and a small icon for the launcher's list view. Table 6-1 shows the properties these icons must have.

Table 6-1
Launcher Icon Properties

Icon	*Resource ID*	*Size*
Large launcher icon	1000	22 × 22 pixels
Small launcher icon	1001	15 × 9 pixels

The canvas of the multibit icon editor window is conveniently sized at 22 × 22 pixels, making large icon creation straightforward. For the small icon, use the 15 × 9 pixel area in the upper-left corner of the canvas. The Palm OS ignores anything outside this small area when displaying the small icon.

Although the multibit icon editor window provides a number of useful tools for creating an icon, many of these tools do not work as described in the Metrowerks Constructor documentation. Often, the editor can be downright random about how it interprets your mouse clicks, leading to a lot of frustration when editing icons. Fortunately, it is possible to create the icon in a bitmap editor of your choice, copy it to the clipboard, and then paste it into the multibit icon editor.

To paste a bitmap from the clipboard into the multibit icon editor, follow these steps:

1. Select the preview image of the icon you wish to paste into, either B&W or Color.
2. Select the canvas to make it active.
3. Select Edit ➪ Paste, or press Ctrl+V, to paste the image into the canvas.

If the image you paste into the canvas is larger than 22 × 22 pixels, Constructor will not resize it, but simply crop it at the edges. Fortunately, right after pasting, you may drag the image around the canvas to control what part of it is cropped.

Icons

Constructor provides an editor window for standard black and white icons, pictured in Figure 6-13. Note the similarity to the multibit icon editor.

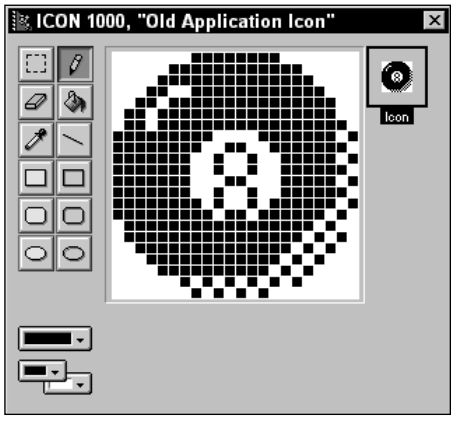


Figure 6-13: The icon editor window

Before the introduction of multibit icons in Palm OS 3.0, the icon editor window was the only way to create an application icon in Constructor. The icon editor window allows creation and editing of black and white icons only, and its main function is to provide backward compatibility with resources generated for earlier versions of the Palm OS.

If you have a black and white icon from a project file created for a version of the Palm OS prior to 3.0, you can convert it to the newer multibit icon resource by following these steps:

1. Open the project file containing the old icon. The project window appears.
2. Open the icon by double-clicking it in the project window. The icon editor window appears.

3. Choose Edit ➪ Select All, or press Ctrl+A, to select the entire canvas in the icon editor window.
4. Select Edit ➪ Copy, or press Ctrl+C, to copy the icon to the clipboard.
5. Select Multibit Icons in the project window and choose Edit ➪ New icon Resource, or press Ctrl+K, to create a new multibit icon. The multibit icon editor window appears.
6. Select the B&W sample view in the multibit icon editor window, and then select the canvas to make it active.
7. Select Edit ➪ Paste, or press Ctrl+V, to paste the old icon into the new multibit icon.
8. Drag the B&W sample view onto the Color sample view to copy the black and white icon into the grayscale portion of the multibit icon.

Bitmaps

The bitmap editor window, pictured in Figure 6-14, provides a space for editing miscellaneous bitmaps to include in your application. Notice that the interface for the bitmap editor is similar to the two icon editors; it is missing only the small preview views to the right of the canvas.

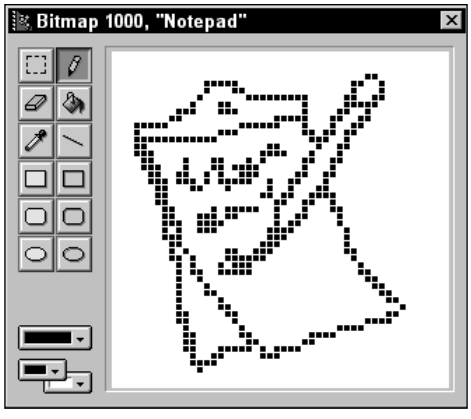


Figure 6-14: The bitmap editor window

You can create black and white, grayscale, or color bitmaps in the bitmap editor window. To change the color depth of the bitmap, choose an appropriate depth from the Colors menu. The options available are Black & White (1-bit depth), 4 Grays (2-bit depth), 16 Grays (4-bit depth), and 256 colors (8-bit depth). A single bitmap resource may only have one color depth, so if you need to have multiple depths for a single image, you will need to create a bitmap family to contain a number of bitmaps of varying depths. See the “Bitmap families” section later in this chapter for details about how to create a bitmap family.

To create a grayscale bitmap for use in Palm OS version 3.0 or 3.1, you must follow specific naming conventions for your bitmap resources. Otherwise, the OS does not recognize the images as grayscale and displays them as black and white. Table 6-2 shows the prefix codes to include at the beginning of the bitmap resource's name, depending on the type of bitmap you wish to create.

Table 6-2
Grayscale Bitmap Prefixes for Palm OS 3.*x*

Prefix	*Meaning*
-1	1-bit (black and white) bitmap
-2	2-bit (four-color grayscale) bitmap
-c	Uncompressed bitmap

The prefix codes must be surrounded by slash (/) characters, and they must be the first part of the bitmap resource's name. For example, to create an uncompressed grayscale bitmap called "foo," name the bitmap resource `/-2 -c/foo`. Without any prefix at all, Constructor creates compressed black and white bitmaps, the equivalent of a `/-1/` prefix.

In order to create a bitmap resource that contains both 1-bit and 2-bit bitmaps, you must use the prefix `/-1 -2/`. Just `/-2/` will create a 2-bit bitmap.

By default, Constructor generates compressed bitmaps. Be sure to include `-c` as part of the prefix if you want an uncompressed bitmap.

If your application is intended to run on Palm OS 3.5 or later, you can skip all of the special naming conventions listed previously. Color support is much more advanced in Palm OS 3.5, which has no difficulty with 2-bit and 4-bit grayscale images, regardless of how they are named.

App icon families

An app icon family resource contains multiple copies of an image at different color depths, for use as an application icon. If an application needs a color icon for the Palm OS 3.5 application launcher, but it still needs to run under earlier versions of the Palm OS, an app icon family allows the application to display an icon with an appropriate color depth on older versions of the system launcher, which do not support color.

To create an app icon family, you first need to create a number of bitmaps, one for each color depth that will appear in the app icon family. Usually, you should create four bitmaps, one each at 1-, 2-, 4-, and 8-bit color depths. These bitmaps should be 22×22 pixels in size for a large application icon, or 15×9 for a small icon.

The next step is to add the bitmaps to an app icon family resource. In the app icon editor window, pictured in Figure 6-15, you can add a new image by selecting Edit ➪ New Family Element, or by pressing Ctrl+K. An app icon family may contain a maximum of four images. Also, be sure to set the appropriate Width and Height values for the app icon family.

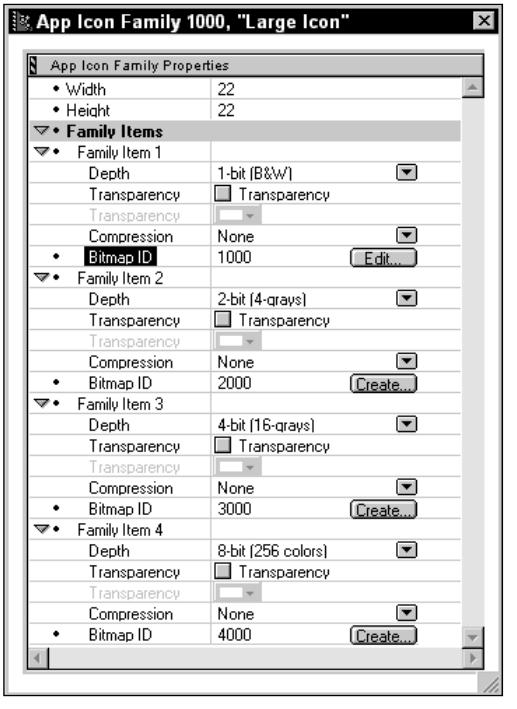


Figure 6-15: The app icon family editor window

The quickest way to add all the images needed in an app icon family is to select the Family Items line in the editor window, then press Ctrl+K once for each image you need in the family. Using this process also automatically increments the Depth field for each image, so you do not need to manually set the color depth for images one at a time.

For each image, you can set a number of options:

- **Depth.** This option specifies the color depth of the bitmap image. There should only be one bitmap of a particular color depth within a particular app icon family.

- **Transparency.** If checked, one of the colors in the icon is treated as transparent. Checking the Transparency check box enables the Transparency drop-down, from which you may choose one of the colors in the image to be the transparent color. Pixels in the image colored with the selected color are treated as transparent.
- **Compression.** This drop-down allows you to select the style of compression used for the image. In general, icons should be left uncompressed.
- **Bitmap ID.** Enter the resource ID of the appropriate bitmap image in this text field. Once an ID has been entered, clicking the Edit button opens the selected bitmap in the bitmap editor window.

Images in an app icon family must be entered from lowest bit depth to highest, which usually means 1-bit (B&W) at the top of the list and 8-bit (256 colors) at the bottom. Constructor does not enforce this policy; if you enter the images in the wrong order, you will get unpredictable results when the launcher tries to display your application's icon.

Just like ordinary and multibit icons, you need to use resource ID numbers of 1000 and 1001 to represent the application's large and small icons, respectively.

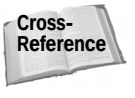
Bitmap families

Bitmap families function in exactly the same manner as app icon families, except they are used for general bitmap images rather than an application's launcher icon. The interface for creating and editing a bitmap family is identical to that for manipulating an app icon family. Also, the same caveat regarding color depth order applies to bitmap families; be careful to list the individual images from lowest bit depth to highest.

Creating Catalog Resources

Catalog resources are user interface elements that may be contained within a form, such as buttons and scroll bars. To create a catalog resource in Constructor, you must drag it from the Catalog window to an existing form editor window. Because the Catalog window, pictured in Figure 6-16, is not available when you first start Constructor, you must open it by selecting Window ➪ Catalog, or by pressing Ctrl+Y.

Once the Catalog window is open, you can drag user interface elements from the Catalog window to any open form editor window to add those elements to the form.



The full details of adding catalog resources to a form are too lengthy to include here; see Chapter 7, "Building Forms," for more information.

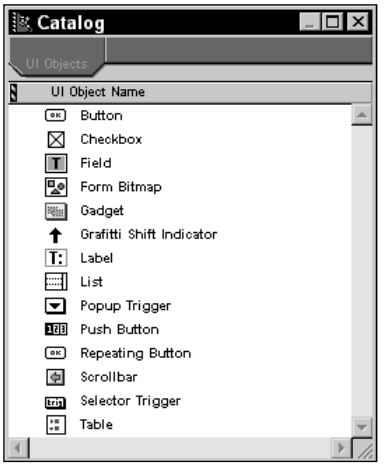


Figure 6-16: The Catalog window

Creating Resources with PilRC

PilRC uses a different strategy from Constructor's to produce Palm OS resources. Instead of graphically assembling the interface on-screen, you make a text file describing the resources you want, which PilRC compiles into a form that the GNU Palm tools can build into a working `.prc` file.

A PilRC resource file has an `.rcp` extension. To compile an `.rcp` file into resources, use the following command line

```
pilrc file.rcp
```

where *`file.rcp`* is the resource file you wish to compile. You may also specify an output directory, like this:

```
pilrc file.rcp C:\resources
```

The previous command line would compile resources from `file.rcp` and write them to the directory `C:\resources`. If you omit the output path, PilRC writes resources to the current directory.

Each resource that PilRC compiles becomes its own separate file with a `.bin` extension. Compiled resource file names are composed of the resource's four-character type, followed by the resource ID in hexadecimal. For example, a form (type `tFRM`) with resource ID of 1000 becomes a file named `tFRM03e8.bin`.

Creating Application Resources

The following sections detail the syntax required to create specific project resources in PilRC. Words in `ALL CAPS` represent actual keywords that should be typed as-is in the `.rcp` file. Words contained in angle brackets ("<" and ">") represent required fields for which you must provide a value, and anything contained in square brackets ("[" and "]") represents an optional field.

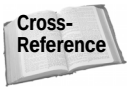
Both C and C++ style comments work within `.rcp` files, provided you place them between different resource commands in the file, not within them. For example, the following lines will result in a PilRC error:

```
ALERT ID 1000
INFORMATION
// This comment will cause an error
BEGIN
    TITLE   "About Hello World"
    MESSAGE "Hello World\nVersion 1.0"
    BUTTONS "OK"
END
```

Many PilRC commands take string arguments. Within a string argument, PilRC understands certain character escapes to indicate special characters. Table 6-3 outlines these special characters.

Table 6-3
PilRC Special Characters

Character	*Meaning*
`\t`	Tab character
`\n`	Linefeed
`\nnn`	`nnn` stands for a three-digit octal character code. For example, `\141` represents a lowercase `a`.



Forms and the catalog resources they contain are covered later, in Chapter 7, "Building Forms." Menus are detailed in Chapter 8, "Building Menus."

Application icon name

PilRC uses the following command to generate the application name that will appear next to the application's icon in the launcher:

```
APPLICATIONICONNAME ID <resourceID> <application name>
```

For example, the following line will create an application icon name of "Hello" with resource ID 100:

```
APPLICATIONICONNAME ID 100 "Hello"
```

Because the application icon name is a required parameter for the `build-prc` tool, it is not strictly necessary to include an `APPLICATIONICONNAME` directive in your `.rcp` files. The `build-prc` tool ignores the PilRC-generated application icon name in favor of its command line argument.

Application icons

The command line for creating icons is

```
ICON <icon file name>
```

The `ICON` directive creates a standard Palm OS application icon with a resource ID of 1000. `icon file name` should specify the file name of a Windows 32×32, 32×22, or 22×22 bitmap.

The following example creates an icon from the file `largeicon.bmp`:

```
ICON "largeicon.bmp"
```

To create a small icon for the list view of the application launcher, use the `SMALLICON` command:

```
SMALLICON <icon file name>
```

For small icons, `icon file name` should be the file name of a Windows 15×9 bitmap.

In order to support multiple color depths, you can also create large and small icon families, which contain multiple copies of the same image in different color depths. The directive for creating a family of large icons is `ICONFAMILY`, whose syntax looks like this:

```
ICONFAMILY <1-bit icon file name> <2-bit icon file name>
    <4-bit icon file name> <8-bit icon file name>
    [TRANSPARENT <r> <g> <b>] [TRANSPARENTINDEX <index>]
```

For each of the icon file names in the list, you may skip a particular color depth by placing an empty set of double quotes in that icon's position in the list. For example, the following would create an icon family containing only 1-bit and 8-bit images:

```
ICONFAMILY "1-bit.bmp" "" "" "8-bit.bmp"
```

The TRANSPARENT option allows you to specify the RGB (red, green, blue) value of a color in the image that should be treated as transparent. You may alternately specify TRANSPARENTINDEX to make the color at a particular index within the image's color palette transparent. The following examples show the two transparency options in action; the first makes magenta the transparent color, and the second uses the color at index 255:

```
ICONFAMILY "1-bit.bmp" "" "" "8-bit.bmp" TRANSPARENT 255 0 255
ICON FAMILY "1-bit.bmp" "" "" "8-bit.bmp" TRANSPARENTINDEX 255
```

To make an icon family for an application's small icon, simply substitute SMALLICONFAMILY for ICONFAMILY.

Template Windows bitmaps for the large and small application icons may be found in the GNU PRC-Tools portion of the CD-ROM accompanying this book. The large icon template also includes the "big black dot," which appears as the background for the Palm OS's built-in application icons, as well as in the icons of many third-party programs.

Version string

Use the VERSION command to create an application's version string. For example

```
VERSION ID <resourceID> <version string>
```

The following example creates a version resource with resource ID 1 and a version string of "1.0b":

```
VERSION ID 1 "1.0b"
```

Strings

The STRING command creates a string resource. For example:

```
STRING ID <resourceID> <string>
```

For readability, the string parameter may span more than one line by your appending a backslash (\) to the end of each line of the string, as in the following example:

```
STRING ID 1000 "But, soft!  What light through yonder"\
    " window breaks?\nIt is the east, and Juliet is"\
    " the sun."
```

For particularly large amounts of text, or for easier localization of strings, you may also use the following syntax to import string resources from separate text files:

```
STRING ID <resourceID> FILE <file name>
```

The file name parameter should contain the path and file name of a plain text file that contains the appropriate string.

Categories

PilRC's CATEGORIES command generates an app info string list, suitable for passing to the API function **CategoryInitialize** to set the default category names in an application. For example,

```
CATEGORIES ID <resourceID> <category1> <category2> ...
```

Category names may be only 15 characters long, and there is a maximum of 16 categories. The following example produces some common default categories:

```
CATEGORIES ID 1000 "Unfiled" "Business" "Personal"
```

Alerts

The ALERT command in PilRC generates an alert dialog box resource. For example

```
ALERT ID <resourceID>
[HELPID <help ID>]
[DEFAULTBUTTON <button index>]
[INFORMATION] [CONFIRMATION] [WARNING] [ERROR]
BEGIN
    TITLE <alert title>
    MESSAGE <alert message>
    BUTTONS <button1> <button2> ...
END
```

To include online help in an alert dialog box, fill in help ID with the resource ID of a string resource.

- DEFAULTBUTTON specifies which of the alert dialog box's buttons should be activated when the Palm OS dismisses the dialog box, which usually occurs when the user switches to another application without tapping any of the dialog box's buttons. The button index parameter starts at 0 for the left-most button in the alert.
- One of INFORMATION, CONFIRMATION, WARNING, or ERROR should be specified to indicate the icon the alert displays.
- The TITLE specifies the text that will appear in the title bar of the alert dialog box. Text in MESSAGE will form the main text displayed by the alert. The same multiline continuation available for string resources (see above) may be used for an alert's message string.
- The BUTTONS parameter defines and names the buttons at the bottom of the alert dialog box. All alerts must contain at least one button.

The following example produces the alert pictured in Figure 6-17:

```
ALERT ID 1200
INFORMATION
BEGIN
```

```
    TITLE   "About Hello World"
    MESSAGE "Hello World\n"\
            "Version 1.0"
    BUTTONS "OK"
END
```

Figure 6-17: A sample dialog box produced in PilRC

Bitmaps

The BITMAP command converts bitmaps in .bmp, .pbitm, .xbm, and .pbm formats into Palm bitmap resources. For example

```
BITMAP ID <resourceID> <bitmap file name> [NOCOMPRESS]
    [COMPRESS] [FORCECOMPRESS]
```

Specifying COMPRESS will result in a compressed bitmap if compression would result in a smaller resource than an uncompressed bitmap; FORCECOMPRESS compresses the bitmap regardless of the resulting resource size. By default, the NOCOMPRESS directive is in force, resulting in no compression at all, which should be suitable for most purposes.

The following example converts a Windows bitmap named picture.bmp into a compressed bitmap resource with resource ID 1002:

```
BITMAP ID 1002 "picture.bmp" COMPRESS
```

You may also convert 2-bit and 4-bit grayscale images, as well as 8-bit color images, using the following syntax:

```
BITMAPGREY   ID <resourceID> <bitmap file name>
BITMAPGREY16 ID <resourceID> <bitmap file name>
BITMAPCOLOR  ID <resourceID> <bitmap file name>
```

In much the same fashion as creating application icon families, you may also create bitmap families that contain multiple images to support more than one color depth. Use the BITMAPFAMILY directive, which works in similar fashion to the ICONFAMILY or SMALLICONFAMILY directives:

```
BITMAPFAMILY ID "1-bit.bmp" "" "" "8-bit.bmp"
    TRANSPARENTINDEX 255
```

Previewing the Interface in PilrcUI

As a resource compiler, PilRC does not offer the immediate graphic feedback possible using CodeWarrior's Constructor tool. However, PilrcUI, a companion program shipped with PilRC, does provide a quick preview of the resources defined by an `.rcp` file. PilrcUI's display is not as faithful to the actual appearance of the Palm OS as the views displayed in Constructor, but it is useful for getting an approximate idea of what the resources in an `.rcp` file will look like.

You can start PilrcUI from the command line, passing it the name of the `.rcp` file you wish to view:

```
pilrcui hello.rcp
```

Alternatively, you can run PilrcUI, and then select File➪Open and specify the `.rcp` file you wish to open.

It can be useful to switch between the `.rcp` file in your text editor and PilrcUI as you code your resources. Once an `.rcp` file is open in PilrcUI, selecting File➪Reload, or simply clicking anywhere in the PilrcUI display, reloads the currently displayed `.rcp` file. You can also switch between different forms contained in the same `.rcp` file by choosing a specific form resource from PilrcUI's Form menu.

Assigning Constants to Resources

Using raw resource ID numbers in your source code can be a debugging nightmare, because few programmers can successfully memorize a bunch of four-digit numbers and the resources they represent. Fortunately, PilRC allows you to include constant definitions from a standard C `.h` file, allowing you to substitute symbolic constants for resource ID numbers.

All that is required in the PilRC file to include a header file full of constant definitions is an `#include` statement at the head of the `.rcp` file. For example, the following line includes the header file `helloRsc.h`:

```
#include "helloRsc.h"
```

The header file should contain standard C-style `#define` statements. This line, taken from the `helloRsc.h` file in Chapter 4, defines a constant for the resource ID of the Hello World application's about box:

```
#define AboutAlert 1200
```

Now, instead of using the resource ID `1200` in the `hello.rcp` resource file, the following will work:

```
ALERT ID AboutAlert
```

Summary

In this chapter, you learned about Palm Computing's user interface guidelines, as well as resource creation. After reading this chapter, you should understand the following:

- ✦ Palm OS user interface guidelines are a combination of making fast programs, matching frequency-of-use to accessibility, and making programs that are easy to use.
- ✦ Following the Palm OS user interface guidelines allows you to make applications that better fit the expectations of Palm OS handheld users, decreasing your application's learning curve and its users' level of frustration.
- ✦ CodeWarrior Constructor is a graphical tool that provides a WYSIWYG interface for creating Palm OS resources.
- ✦ PilRC provides a different approach to resource creation from Constructor's, compiling resource descriptions from a text file into actual resources.

✦ ✦ ✦

CHAPTER 7

Building Forms

In This Chapter

Building forms with Constructor

Building forms with PilRC

Creating increment arrows

The primary interface a Palm OS application presents to users is contained within forms. Forms are the user's windows to working with an application, providing displays for the user to view data and controls to allow the user to manipulate that data. This chapter will show you how to construct forms and the catalog resources that make up the user interface elements within those forms, using both CodeWarrior's Constructor and the GNU PRC-Tools's PilRC tools.

Building Forms with Constructor

To create a new form in Constructor, select the gray Forms bar in the project window, and then select Edit➪New Form Resource, or just press Ctrl+K. Double-clicking the name of the newly created form, or selecting it and pressing Enter, opens the *form layout window*, pictured in Figure 7-1. The form layout window is where you add, position, and modify the user interface elements in a particular form.

The right side of the form layout window, labeled Layout Appearance, displays the form and provides a workspace for selecting, positioning, and resizing the form's contents. Layout Properties, on the left side, shows a list of properties for whatever object is currently selected in the Layout Appearance side of the window.

When you create a form, the object ID numbers in the Layout Appearance part of the form layout window tend to obscure the user interface objects, making it difficult to see what the completed form will look like. To hide the object IDs, click anywhere in the form display, and then select Layout➪Hide Object IDs.

Simply click an object to select it. You may position a selected object by dragging it around the Layout Appearance view. Use the arrow keys to make fine adjustments, one pixel at a time, to an object's position. Dragging the black boxes in the corners of an object resizes it.

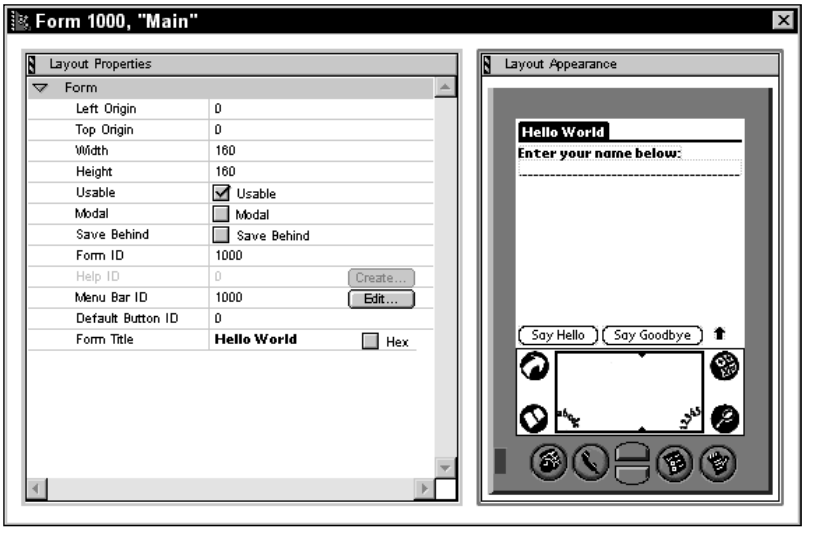


Figure 7-1: The form layout window

Constructor also provides another way to view and select a form's contents. The *hierarchy window*, pictured in Figure 7-2, lists a form and its contents. To open the hierarchy window, select Layout➪Show Object Hierarchy, or press Ctrl+H.

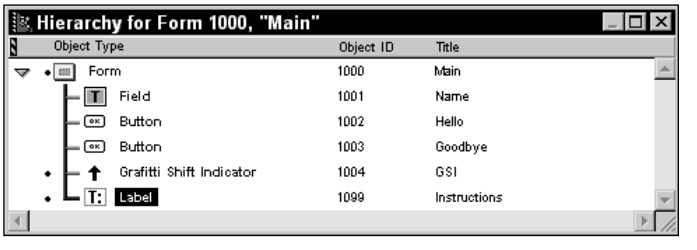


Figure 7-2: The hierarchy window

Selecting an object in the hierarchy window also selects that object in the form layout window. Likewise, picking different objects from the form layout window changes which object is selected in the hierarchy window.

If your form contains objects that overlap or completely cover each other, it can be difficult to select them in the Layout Appearance side of the form layout window. Use the hierarchy window to select objects that are buried under other objects.

The Catalog window, pictured in Figure 7-3, is the source for all new form objects. To open the Catalog window, select Window➪Catalog or press Ctrl+Y.

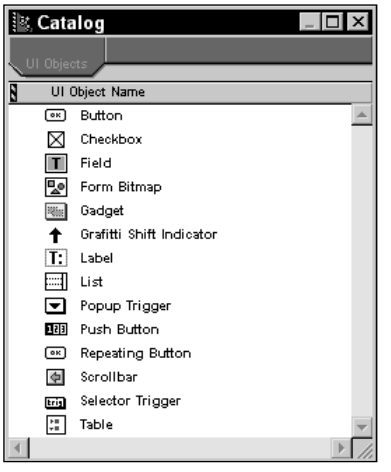


Figure 7-3: The Catalog window

Creating new objects on a form is a simple matter of dragging the appropriate object from the Catalog window to the form layout window, where you can resize and set the new object's properties. You may also copy objects from other forms, even forms in other projects, and then paste them into a new form. To copy an object, select it in either the form layout window or the hierarchy window, and then select Edit➪ Copy Object or press Ctrl+C. Paste an object into a form by selecting Edit➪ Paste Object or pressing Ctrl+V. Objects copied in this way retain most of their original properties; Constructor changes only the object ID to a number appropriate to the object's new form.

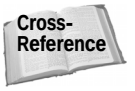
You can save time designing an interface by copying and pasting from the example applications provided by Palm Computing, or from other applications you have written yourself. Copying resources from the built-in applications also ensures that your own program has a similar look and feel to the applications with which most Palm OS users are familiar.

To delete an object, select it in the form layout window or the hierarchy window, and then select Edit➪ Clear Object or press Delete.

Setting Common Object Properties

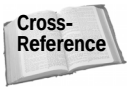
Many interface objects share common properties, which are described as follows:

- Object Identifier**:** Not to be confused with an object's resource ID, the Object Identifier allows you to give a name to an object that is more readable than the numeric resource ID. The Object Identifier also appears next to objects in the hierarchy window. Besides offering a human-readable alternative to the resource ID, Constructor also uses the Object Identifier to create resource constants when automatically generating a header file.



See Chapter 6, "Creating and Understanding Resources," for more details about automatically generating a header file in Constructor.

- Left Origin and Top Origin**:** These two properties control the location of the upper-left corner of the object in relation to the screen. The upper left corner of the screen has the coordinates 0, 0. Coordinates increase from left to right and from top to bottom, with the lower-right corner having coordinates 159, 159.
- Width and Height**:** These properties control the width and height of the object, in pixels.
- Font**:** Many user interface objects display text as part of the object. The Font property controls which font the Palm OS uses to draw the object's text. Table 7-1 shows examples of the available fonts. The Palm OS identifies fonts by means of an enumerated type called `FontID`. Table 7-1 also lists the constants in the `FontID` enumeration. Note that the Bold 12 font was introduced in Palm OS 3.0, and is therefore not available in earlier versions of the operating system.
- Usable**:** If an object's Usable property is unchecked, the Palm OS neither draws the object nor allows user interaction with the object. Application code can set the Usable property of an object at run time. This allows for user interface elements that are hidden when the application first displays a form, and then appear in response to user input.



Chapter 9, "Programming User Interface Elements," contains further information on changing the Usable property.

Table 7-1
Fonts in the Palm OS

Font Name	*#*	*FontID Constant*	*Example*
Standard	0	`stdFont`	The quick brown fox jumps over the laz
Bold	1	`boldFont`	**The quick brown fox jumps over**
Large	2	`largeFont`	The quick brown fox jumps o
Symbol	3	`symbolFont`	
Symbol 11	4	`symbol11Font`	
Symbol 7	5	`symbol7Font`	
LED	6	`ledFont`	,-.e0123456789
Bold 12	7	`largeBoldFont`	**The quick brown fox jumps over**

Setting Individual Object Properties

The following sections explain how to set various properties for forms and the user interface resources they may contain. Where appropriate, the object descriptions also contain guidelines for properly using each object.

Forms

The property that affects how a form behaves the most is its Modal property. If Modal is checked, the Palm OS draws the form with a border and ignores stylus taps outside the edges of the form. Check the Modal property when creating a dialog box, and leave it unchecked for full-screen views.

Another property that forms have is Save Behind. When the Palm OS draws a form with the Save Behind property checked, the system saves the region of the screen occupied by the form, and then redraws that region after erasing the form. This can save some time when returning from a dialog form because the form underneath does not have to redraw all its contents, just the portion that was covered by the dialog form.

There is one case where the system may not restore the pixels behind a modal form. If there is not enough memory available to save the screen area behind the modal form, the Palm OS posts a `frmUpdate` event to the underlying form, instead, requesting the underlying form to redraw its contents. In fact, the debug versions of Palm OS 3.5 always post an update event in lieu of saving the screen area behind a modal form. Relying on the Save Behind property to redraw a form for you may not always work, so be sure that any form that will have modal dialog boxes displayed over it handles `frmUpdate` in its event handler.

When should you use the Save Behind property, then? To illustrate, imagine an application with two forms, called A and B. If you open form B over the top of form A, and form A is not likely to change its contents while covered, form B should have the Save Behind attribute set. Likewise, if form A might change its contents while form B is open, not setting Save Behind on form B is a better idea, because form A will need to redraw its contents anyhow, and saving the area behind form B is simply a waste of system resources.

You can set the title of the form by changing the Form Title property. In full-screen forms, the system draws the title left-justified at the top of the form; in modal dialog boxes, the title is centered. Regardless of whether the form is modal or not, the title occupies the top 13 pixels of the form and can contain only a single line of text.

In modal dialog boxes, you may set the Help ID to the resource ID of a string resource. If Help ID is set, the operating system draws a small “i” icon in the upper-right corner of the form’s title bar. Tapping the icon displays the string resource in a dialog box labeled “Tips.” You can provide the user with online help for dialog boxes using the Help ID mechanism.

Setting a form's Menu ID attaches a specific menu bar resource to that form. When the user taps the Menu silk-screened button, the system displays the menu attached to the currently displayed form. Creating menu resources is covered later in this chapter.

The Default Button ID specifies which button the operating system should pick in a modal dialog box if the user switches to another application instead of exiting a dialog box after having made a button choice. Default Button ID should contain the resource ID of the appropriate button.

A modal dialog box should occupy the entire width of the screen and rest at the bottom, obscuring any command buttons in the application beneath it and leaving the application's title bar visible. Three pixels of space should separate the top of the dialog box's title bar from the bottom of the application's title bar; if the dialog box is too large to allow this, the dialog box should occupy the whole screen. The border around a modal dialog box is not included in the width and height that you set for a form, so you should allow for an extra two pixels on the sides, top, and bottom of the form. For example, a full-screen modal dialog should have both its Left Origin and Top Origin set to two, and its Width and Height set to 156.

Buttons

The Anchor Left property of a button does not actually do anything. Anchor Left is useful in some other controls, such as pop-up triggers and selector triggers, but it serves no useful purpose for buttons, so you can safely ignore it.

Check the Frame property to give the button a frame, which the Palm OS draws as a rectangle with rounded corners. Most buttons should have a frame. The major exception to this is an *increment arrow*, whose caption is a single arrow character from one of the Palm OS symbol fonts. Increment arrows do not need a frame. The sidebar "Creating Arrow and Scrolling Buttons" in this chapter describes how to create increment arrow buttons.

Non-Bold Frame controls the thickness of the frame to draw around the button. When checked, the system draws the button with a single-pixel frame. Left unchecked, Non-Bold Frame causes the button to have a bold, two-pixel frame. By default, buttons in Palm OS applications should have non-bold frames.

The Label field controls the text displayed in the button. The system draws button labels centered in the middle of the button, clipping the left and right edges if the text is longer than the width of the button.

In Palm OS 3.5 and later, you can create a graphic button, which displays a graphic image instead of text. A graphic button has two bitmaps associated with it, one that normally appears on the button, and another that only shows up when the user taps the button. Set the normal image by entering the resource ID of an existing bitmap resource into the Bitmap Resource field, and set the image to display while selected in the Selected Bitmap field.

Although prior to Palm OS 3.5 you cannot use a bitmap as a button's label, it is still possible to make a button that appears to have a picture on it. Simply create a form bitmap resource the same size as the button and place both the button and the bitmap in the same location on the form. The Palm OS will even invert the bitmap (turning black pixels white and vice versa) when the user taps the button.

Creating Arrow and Scrolling Buttons

Most of the Palm Computing built-in applications make use of *increment arrows*, which appear as arrow-shaped buttons without frames. One good example is the pair of scroll arrows located in the lower-right corners of the To Do and Address Book applications.

Increment arrows are simply regular button or repeating button resources that do not have frames. Instead of a normal text caption in the button, a single character from one of the Palm OS symbol fonts serves to make the button appear to be arrow-shaped. The symbol fonts contain a variety of arrows, as well as many other small graphic elements used by the operating system, such as check boxes and Graffiti shift indicator icons.

To make an increment arrow, do the following:

1. Create a regular button or repeating button resource.
2. Uncheck the button's Frame property in Constructor or use the `NOFRAME` attribute in PilRC.
3. Set the button's font to the symbol font containing the arrow character you want to use. The three symbol fonts are Symbol, Symbol 11, and Symbol 7, which you can identify in PilRC with the numbers 3, 4, and 5, respectively.
4. Set the button's label text to the appropriate arrow character. In Constructor, check the Hex check box next to the Label property and enter the character's number in hexadecimal. For a PilRC resource file, specify the button's label as an octal number preceded by a backslash. For example, `BUTTON "\001"` sets the button's label to be character 1 (decimal).
5. Resize the button to about the same size as the character in the button's label.

The following table lists arrow characters in the Palm OS symbol fonts and the information you need to use those characters as increment arrows.

Symbol	*Font*	*Number (decimal)*	*Width*	*Height*
	Symbol	3	12	12
	Symbol	4	12	12
	Symbol	5	12	12
	Symbol	6	12	12

Continued

Continued

Symbol	*Font*	*Number (decimal)*	*Width*	*Height*
	Symbol	7	11	10
	Symbol	8	11	10
	Symbol 11	2	8	13
	Symbol 11	3	8	13
	Symbol 7	1	13	8
	Symbol 7	2	13	8
	Symbol 7	3	13	8
	Symbol 7	4	13	8

The standard vertical scroll arrows in the built-in applications are repeating buttons using characters 1 and 2 (decimal) from the Symbol 7 font. The Palm OS provides the FrmUpdateScrollers function to gray out one or the other of the scroll arrows when at the top or bottom of the data a form can display. See Chapter 9, "Programming User Interface Elements," for more details about using FrmUpdateScrollers.

Check boxes

The Selected property of a check box controls whether or not the box should appear checked by default when the Palm OS draws it on a form. If Selected is checked, so is the check box.

You can group multiple check boxes together to make them mutually exclusive using the Group ID property. Only one check box in a group sharing the same Group ID may be checked at a time. Be sure to set the Selected property of only one check box in a group; setting more than one causes strange behavior at run time.

Label controls the text displayed to the right of the check box. Tapping the text in the label toggles the check box just as if the box itself had been tapped. In fact, if you size a check box larger than just the area occupied by its box and label, all of the space surrounding the check box, out to the edges of its height and width, will toggle the box when tapped.

Although the check box resource does not offer an option to create a label to the left of the check box, it is possible to create one yourself. Leave the Label property of the check box blank, and then create a separate label resource and place it to the left of the check box. The user cannot toggle the check box by tapping this label, but tapping the box itself still performs the expected toggling.

Fields

A text field's Editable property controls whether the user may change the contents of the field. When Editable is checked, the field is user-editable, and when unchecked, the field does not accept user input. You may still change the contents of the field programmatically, even when Editable is not set. A non-editable text field can be used to display variable-length text without resorting to the basic Palm OS drawing APIs to manually write characters to the screen.

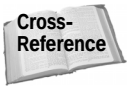
Underline controls whether a field should have a gray (dotted) underline under each row of text. If this property is checked, the field is underlined. Otherwise, the field's text displays without an underline. Editable text fields should always have an underline; without an underline, there is no indication to the user that the field even exists.

The Windows version of Constructor that ships with CodeWarrior for Palm Computing platform R5 contains a bug in how it generates underlined field resources. When displaying an underlined text field generated by this version of Constructor, Palm OS version 3.1 and later draws a solid underline beneath the text in the field. The underline setting for fields is a two-bit value. In Palm OS version 3.0 and earlier, 0 represented no underline, and 1 represented a gray, dotted underline. The operating system code performed a simple `if` statement to determine whether or not the field should be underlined, not caring what the non-zero value was.

For version 3.1 and later, Palm Computing added the value 2, indicating a solid underline. Unfortunately, the R5 Constructor for Windows has always saved underlined field resources with an underline value of 2, and now that the operating system actually distinguishes between different non-zero values for the underline attribute, any underlined field resource generated by Constructor on Windows displays with a solid underline in recent versions of the OS. You can work around this limitation by manually setting the underline attribute in your code for each underlined field before displaying them in your application; see Chapter 9, "Programming User Interface Elements," for more details. Fortunately, the R6 version of CodeWarrior for Palm Computing platform fixes this problem for Windows developers.

Check the Single Line property to create a *single-line field*. A single-line field displays only one line of text, and it does not accept Tab or Return characters. Attempting to enter text past the end of the line causes the system to beep. When the Single Line property is not checked, the field becomes a *multiline field*. A multiline field scrolls when the user enters more text than the field can display at once, or when the user drags the stylus to select text outside what the field is currently displaying.

The Dynamic Size property, when checked, tells the field to put a `fldHeightChangedEvent` in the event queue when the user enters enough text to cause the field to scroll, either up or down. You can intercept this event in your code to allow your application to resize the field as the user enters text.



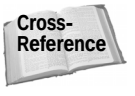
See Chapter 9, "Programming User Interface Elements," for details about handling the `fldHeightChangedEvent`.

If a field's Editable property is not set, the Left Justified check box becomes available. Leaving Left Justified checked means that text in the field is justified to the field's left margin; unchecking Left Justified changes the field to right justification. Note that user-editable fields cannot be right justified, which is why this property cannot be changed when Editable is checked.

Max Characters sets the maximum number of characters that the field can contain. The Palm OS has an absolute maximum of 32767 characters in a single field. All fields require a Max Characters value to work properly.

If Auto Shift is checked, Palm OS version 2.0 and later will perform automatic capitalization at the beginning of sentences in this field. Setting this property is a good idea in order to maintain consistency with other Palm applications.

The Has Scroll Bar property, when checked, causes the field to put more `fldHeightChagnedEvent` events onto the queue. Setting this property is important for text fields with associated scroll bar resources that must be updated regularly as the contents of the field change.



See Chapter 9, "Programming User Interface Elements," for more information about implementing scrolling text fields.

When checked, the Numeric property restricts character entry to numbers. The field will ignore any input that does not consist of numeric characters (0–9) or the decimal character, as currently defined in the Palm OS number format settings.

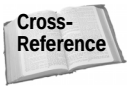
It is still possible to paste non-numeric data into a numeric field from the clipboard, so you cannot rely entirely on the Numeric property to validate the text field's contents.

Form bitmaps

A form bitmap resource serves as an anchor to attach a bitmap resource to a form. The form bitmap's Bitmap Resource ID property contains the resource ID of an existing bitmap. You may also click the Create button in the Bitmap Resource ID field to open the standard bitmap editor; in this case, Constructor creates the bitmap resource for you and assigns its resource ID to the form bitmap's Bitmap Resource ID property.

Gadgets

Other than setting its position and size, you really cannot do a whole lot with a gadget resource at design time. Everything interesting about a gadget occurs at run time.



Chapter 9, "Programming User Interface Elements," contains further information about programming gadget resources.

Graffiti shift indicator

Any form that contains an editable text field should also have a Graffiti shift indicator. Only one indicator should be placed on a form; extra shift indicators are not only redundant, they also confuse the Palm OS field-handling routines, resulting in unpredictable field behavior. You should place a Graffiti shift indicator in the lower-right corner of a form. Once it has been placed on a form, you can effectively ignore a Graffiti shift indicator when developing your application; the Palm OS handles updating the indicator automatically.

Labels

A label lets you add text to a form that the user cannot edit. Changing a label at run-time through code is also rather limited. Any new text you assign to a label using the **FrmCopyLabel** function may be no longer than the original text as defined in the Constructor label resource. Labels are intended only for static text display.

The Text property of a label contains the text it should display. Keep in mind when editing the Text property that a label's text may contain multiple lines. Pressing Enter while editing a label's caption inserts a line break into the text displayed by the label.

Lists

List resources serve double duty. By themselves, lists are stationary interface elements that occupy space on a form. When combined with a pop-up trigger resource, lists become hidden elements that appear for selection only when the user has need of them. List properties must be set a little differently depending on whether you plan to use them as-is or in conjunction with pop-up triggers.

A list resource does not have a Height property. Instead, you control how tall the list is by specifying in the Visible Items property the number of rows the list should display. If the list contains more items than the value of the Visible Items property, the list contains arrow buttons to allow scrolling.

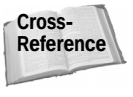
If you specify a value of zero (0) for Visible Items, the Palm OS will draw all the items in the list. Constructor does not show any rows for a list with zero Visible Items, but the operating system draws them anyway, which can result in a bit of a mess if you have another interface element in the "empty" space below the list. You should usually use a Visible Items setting of zero only when the list is intended as a pop-up. Specifying a positive value for Visible Items in a non-pop-up list ensures that the list looks exactly the way you want it to at run time.

Setting Visible Items to a value greater than the number of items actually in the list can result in an error. If you plan to fill the list dynamically at run time, you must either include enough placeholder list items to equal or exceed the Visible Items setting, or you must call LstSetListChoices to set the number of items in the list.

For lists that appear as part of a form's regular interface, be sure to check the Usable property. A pop-up list should have the Usable property turned off to keep the system from drawing it on the form until the user taps its associated pop-up trigger.

To add items to a list, select the List Items row in the Layout Properties side of the form layout window. Then, select Edit➪New Item Text or press Ctrl+K. You can remove list items by selecting them, and then choosing Edit➪Clear Item Text or pressing Backspace.

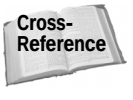
When the Palm OS displays a list attached to a pop-up trigger, the list appears in the position specified by its Top Origin and Left Origin properties, not relative to the location of the pop-up trigger. Be sure to place the list so that it covers the entire pop-up trigger to prevent ugly bits of the trigger from peeking around the edges of the displayed list. If your pop-up list needs to change location, you must handle this yourself in code. For example, the pop-up triggers for selecting a type of phone number in the Address Book application's Edit view are part of a table, so the program cannot assume that they will always be in the same position. Address Book handles this by changing the location of the phone number type list before displaying it.



Chapter 9, "Programming User Interface Elements," explains how to dynamically change the location of a pop-up list.

Pop-up triggers

A pop-up trigger is useful only when combined with a list resource, but it does not necessarily need to display an item from its attached list. The Label property controls the text initially displayed in a pop-up trigger's label when the system first draws the pop-up trigger control. By default, the system changes a pop-up trigger's label to the most recently selected list item in the pop-up's attached list, but you can modify the label in code if you want to display something other than a list value. For example, the Details dialog box in the built-in To Do application has a Due Date pop-up, which displays an actual date instead of "Today," "Tomorrow," or the other values in the pop-up trigger's associated list.



Dynamically changing a pop-up trigger's label in response to user input is covered in Chapter 9, "Programming User Interface Elements."

Set the List ID property to the resource ID of the list that the pop-up trigger should display when tapped.

When the pop-up trigger's label text changes, the width of the entire pop-up control also changes to accommodate the new label. Text longer than the current label causes the trigger to grow, whereas shorter text shrinks the control's width. The Anchor Left property, when checked, nails down the left side of the control, causing the right end of the trigger to do all the growing and shrinking. Unchecking Anchor Left reverses this behavior, fixing the position of the right side of the pop-up trigger and causing the system to resize the left end of the control. Keep in mind when designing a form that the size of a trigger is dynamic; user interface elements on the sizeable end of a pop-up trigger could collide with the trigger's label if it grows long enough.

A common use for a pop-up trigger with the Anchor Left property unchecked is for a right-justified category selector in the upper-right corner of the application. Category pop-up triggers in the built-in applications have the following properties:

- Left Origin: 160
- Top Origin: 0
- Width: 0
- Height: 13
- Anchor Left: Unchecked

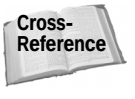
Push buttons

Like check boxes, push buttons have a Group ID property to assign a number of push buttons to an exclusive group. Only one push button in a particular group may be selected at a time. Unlike check boxes, it does not make sense to leave push buttons with a Group ID of zero (0). Push buttons should always occur in groups, never individually; however, the Palm OS does not enforce this.

Set the text inside a push button by editing its Label property. Like that on regular buttons, push button text is centered and clipped at the edges of the button.

In Palm OS 3.5 and later, you may create a graphic push button, which can display an image instead of text. Like a graphic button, there are two images associated with a graphic push button, one for normal display and one that appears when the user taps the push button. Set the normal image in the Bitmap ID field, and the selected image in the Selected Bitmap ID field.

There is no way to set at design time whether a push button is selected or not. You must set the currently selected push button in a group through code when you initialize the form containing the push buttons.



See Chapter 9, "Programming User Interface Elements," for more information about initializing push buttons.

A group of push buttons should be arranged either in a single row or a single column. Buttons in a row should all have the same height; likewise, buttons in a column should all be the same width. Adjacent push buttons share a border with each other, resulting in a single-pixel line between buttons.

Unfortunately, there is no simple way to create a series of push buttons all at once. You must create each button in a group individually, manually positioning it in relation to its brethren and sizing it to match the other buttons' height or width. The quickest way to create consistently sized push buttons is to make one button with the proper size, and then select Edit ➪ Duplicate Object or press Ctrl+D to make an exact copy of it. Using the arrow keys to perform fine adjustments to the buttons' positions, or just filling in the Top Origin and Left Origin properties, can also be less frustrating than using the mouse to move the buttons into their final positions.

Repeating buttons

The properties of a repeating button are exactly the same as those of a regular button; only the button's behavior differs. The built-in applications commonly use repeating buttons to provide scrolling. When used as a scroll button, a repeating button should not have a frame, and its Label property should contain a single arrow character from one of the Palm OS symbol fonts. The sidebar "Creating Arrow and Scrolling Buttons," earlier in this chapter, provides details on duplicating the scroll buttons of the built-in applications.

In Palm OS 3.5 and later, you may create a graphic repeating button, which can display an image instead of text. You can set the normal image for a graphic repeating button in the Bitmap Resource field, and the selected image in the Selected Bitmap field.

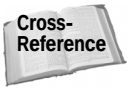
Scroll bars

A scroll bar resource can scroll only vertically, so you should not alter the default width of 7 pixels that Constructor provides. Height is another story; a scroll bar is usually as tall as the table or multiline text field it controls.

The Value, Min Value, Max Value, and Page Size properties affect the initial appearance and behavior of the *scroll car*, the black bar in the middle of the scroll bar. Min Value, usually zero (0), represents the numeric value of the scroll bar when the scroll car is at the top of the bar. Similarly, Max Value is the value when the scroll car is all the way at the bottom of the control. Value indicates the starting position of the scroll car when the Palm OS initializes the scroll bar control. The Page Size property controls the number of units the scroll car moves when the user taps the gray area above or below the car.

That said, most applications should probably leave the values of all four of these properties set to zero (0). The data associated with a scroll bar and displayed in an accompanying table or multiline text field is often dynamic in nature, requiring application code to initialize the scroll bar's properties at run time. If the data controlled by a scroll bar in your application is static, you can get away with setting the scroll

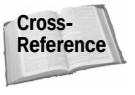
bar's properties at design time and leaving them alone, but more likely than not, you will need to write code to handle initializing and updating the scroll bar.



Setting a scroll bar's properties through code is covered in Chapter 9, "Programming User Interface Elements."

Selector triggers

A selector trigger performs a function similar to that provided by a pop-up trigger. Instead of displaying a list when tapped, though, a selector displays an entire dialog box for user input. After the user selects a value from the dialog box and returns to the form containing the selector trigger, the trigger displays a new value based on user input to its attached dialog box. The built-in applications often use selector triggers with the time and date pickers, which are part of the Palm OS itself.



Chapter 9, "Programming User Interface Elements," discusses using the standard Palm OS date and time pickers.

Like the pop-up trigger, a selector trigger's Anchor Left property controls how the trigger resizes as its label text changes. When checked, the left side of the selector trigger is fixed in place; when unchecked, the right end of the trigger remains stationary.

The Label property provides a place to enter the text the selector trigger initially contains. Not only does the Label property control what displays in the control the first time the application displays the trigger, it also reserves memory space to contain the selector trigger's label. If you try to set a selector trigger's label to a string that is longer than the Label you set at design time, your program will probably crash, because it will not have enough memory for the longer string. Be sure to set a Label that is large enough to contain the largest string the application might assign to the trigger label.

Sliders

A slider's Minimum Value and Maximum Value properties control what value the left and right ends of the slider control represent. You can also set the Initial Value property to define where the thumb appears on the slider, somewhere between the minimum and maximum values. You can control how far the thumb jumps when the user taps to either side of the thumb by setting the Page Jump Amount property.

It is possible to customize the look of a slider control by assigning bitmap images for its thumb and background. Change the thumb image by entering the resource ID of an existing bitmap into the Thumb Bitmap field and background image by changing the Background property to an appropriate resource ID.

The properties for a feedback slider are identical to those for a regular slider control.

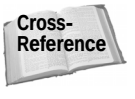
Tables

A table's Rows property controls the number of table rows visible at one time. Constructor divides the height of a table by its number of rows to determine how tall each row should be, so if you resize a table by changing its Height property, you may also need to change the Rows property to prevent stretching or shrinking the height of individual rows.

Be sure that your rows are all the same height. If a table is even a single pixel too short, the table's last row will not display at run time. Tables that are too tall or too short may also result in strange selection behavior or other difficult-to-diagnose bugs. The Height property for a table should be equal to the number of rows times the height of each row. For most tables that use the Standard font, each row should be 11 pixels high.

When you create a table, it already has one column defined. To add more columns, select Column Widths in the form layout window, and then choose Edit➪New Column Width or press Ctrl+K. The value for each of the Column Width properties represents that column's width in pixels.

Setting the types of data that may be displayed in each of a table's cells cannot be done at design time. Instead, you must initialize a table in code before your application can successfully interact with it.



See Chapter 9, "Programming User Interface Elements," for more on initializing and using tables.

Building Forms with PilRC

Like the previous chapter's section on PilRC, this chapter uses the following conventions when describing the syntax PilRC uses to define resources:

- ✦ Words in `ALL CAPS` represent actual keywords that should be typed as-is in the `.rcp` file.
- ✦ Words contained in angle brackets (`<` and `>`) represent required fields for which you must provide a value.
- ✦ Words contained in square brackets (`[` and `]`) represent optional fields.
- ✦ Words separated by a pipe (`|`) are alternates; you should specify one or the other. For example, `USABLE|NONUSABLE` means either `USABLE` or `NONUSABLE`, but not both.

Creating a Form Resource

The form resource in an .rcp file has the following syntax:

```
FORM ID <resourceID> AT (<Left> <Top> <Width> <Height>)
[MODAL]
[SAVEBEHIND|NOSAVEBEHIND]
[HELPID <resourceID>]
[DEFAULTBTNID <resourceID>]
[MENUID <resourceID>]
BEGIN
    [TITLE "Form Title"]
    <OBJECTS>
END
```

The resourceID parameter is simply the resource ID of the form, or a constant defined to represent the form's resource ID.

The AT (<Left> <Top> <Width> <Height>) construct occurs in all the user interface object definitions, and it controls the location and size of the form or control. Left and Top are the left and top coordinates of the object, respectively, and Width and Height represent the object's width and height in pixels. For example, the following line produces a standard, full-screen form:

```
FORM ID 1000 AT (0 0 160 160)
```

Specifying the MODAL attribute produces a form with a two-pixel border, suitable for use as a dialog box. A form with the MODAL attribute also ignores stylus taps outside its own borders. Keep in mind that you must allow extra space for the size of the form's border when defining a modal dialog box. The following lines create a modal dialog box that occupies the entire screen:

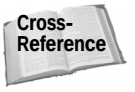
```
FORM ID 2000 AT (2 2 156 156)
MODAL
```

The SAVEBEHIND attribute tells the Palm OS to save the contents of the screen in the same way as the Save Behind property in Constructor. See earlier in this chapter for guidelines on how to use SAVEBEHIND.

You can provide online help for a modal dialog box by setting the HELPID attribute to the resource ID of a string resource. The system places a small "i" icon in the right of the dialog box's title bar, which displays the associated string resource when the user taps it.

The DEFAULTBTNID attribute specifies the resource ID of the form's default button. When the user switches applications before dismissing a dialog box, the system simulates tapping this button to allow your program the chance to perform any cleanup operations required before launching the other application.

To attach a menu to a form, put the menu's resource ID in the form's MENUID attribute.



See Chapter 8, "Building Menus," for more information about creating menu resources.

The section of a form definition between the `BEGIN` and `END` lines defines the objects that occupy the form. One exception to this is the `TITLE` directive, which merely assigns a string to appear in the title bar of the form. Anything else between `BEGIN` and `END` creates an individual catalog resource for the appropriate user interface element.

Adding Objects to a Form

PilRC provides several useful keywords for use in positioning and sizing objects. Table 7-2 describes keywords that you may use in place of numbers when describing an object's size and location. These keywords take the place of parameters in the `AT` portion of an object's definition:

```
AT (<Left> <Top> <Width> <Height>)
```

Table 7-2
PilRC Position and Size Keywords

Keyword	*Meaning*
`AUTO`	Automatically generates width or height of an object, based on its text label. `AUTO` is valid only for the `Width` or `Height` of an item.
`CENTER`	Centers the object with respect to its form. This keyword is only valid in the `Top` or `Left` parts of the `AT` statement. If `Top` is set to `CENTER`, the object is centered vertically on the form; if `Left` is set to `CENTER`, the object is centered horizontally on the form.
`CENTER@<number>`	Places the center of the object at the coordinate specified by `number`. This keyword is valid only when used to define the `Top` or `Left` of an object.
`RIGHT@<number>`	Aligns the item with its right side at the coordinate specified by `number`. This keyword is valid only in the `Left` parameter.
`BOTTOM@<number>`	Aligns the item with its bottom side at the coordinate specified by `number`. This keyword is valid only in the `Top` parameter.
`PREVLEFT`	Represents the previous item's left coordinate.
`PREVRIGHT`	Represents the previous item's right coordinate.
`PREVTOP`	Represents the previous item's top coordinate.
`PREVBOTTOM`	Represents the previous item's bottom coordinate.
`PREVWIDTH`	Represents the previous item's width.
`PREVHEIGHT`	Represents the previous item's height.

Along with the keywords in Table 7-2 and normal numbers, you may also specify an object's coordinates and size with simple arithmetic expressions. For example, the following definitions create three buttons at the bottom of the screen with regular spacing between them, using the text labels of the buttons to determine their sizes:

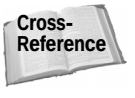
```
BUTTON "New"        ID 1000 AT (1 BOTTOM@159 AUTO AUTO)
BUTTON "Details..." ID 1001 AT (PREVRIGHT+5 PREVTOP AUTO AUTO)
BUTTON "Show..."    ID 1002 AT (PREVRIGHT+5 PREVTOP AUTO AUTO)
```

PilRC also provides another useful keyword to simplify creation of objects on forms; in any place you are required to provide a resource ID, you may instead use the AUTOID keyword. AUTOID automatically assigns a resource ID to an object, starting at 9000 and increasing sequentially. The AUTOID keyword is a quick way to identify objects that you do not need to refer to in your application's source code, as is usually the case with labels. The following example creates a label resource and assigns it a resource ID automatically:

```
LABEL "Enter your name below:" AUTOID AT (0 17) FONT 1
```

Like AT, there are other attributes that are common to more than one of the user interface objects. These attributes are described as follows:

- USABLE and NONUSABLE. These attributes are analogous to the Usable property in Constructor. Including the NONUSABLE attribute in an object definition prevents the operating system from drawing that object on the form. You can set the object back to USABLE status in the appropriate section of your application's code to display the object and allow the user to interact with it. If you omit both USABLE and NONUSABLE from an object's definition, PilRC assumes the object is USABLE and generates resources accordingly.



See Chapter 9, "Programming User Interface Elements," for more information about toggling an object's usability.

- FONT <font number>. For user interface elements that display text, this attribute controls the font in which that text appears. Use the font numbers specified in Table 7-1, earlier in this chapter, to choose an appropriate font for the object's text. If you omit the FONT attribute from an object, PilRC defaults to font number 0, the Standard font.

The following sections describe the commands PilRC looks for between the BEGIN and END lines of a form definition to create objects on a form.

Buttons

A button definition looks like this:

```
BUTTON <label> ID <resourceID> AT (<Left> <Top> <Width>
    <Height>) [USABLE|NONUSABLE] [FRAME|NOFRAME]
    [BOLDFRAME] [FONT <font number>]
```

The FRAME and NOFRAME attributes of a button are mutually exclusive. If FRAME is set, the Palm OS draws the button with a rounded rectangular border. NOFRAME creates a borderless button, suitable for use as an increment arrow; more details on increment arrows are available in the sidebar "Creating Arrow and Scrolling Buttons," earlier in this chapter. If you omit FRAME and NOFRAME entirely, PilRC assumes FRAME as a default and creates the button with a border.

Setting the BOLDFRAME attribute creates a button with a bold, 2-pixel border. This attribute has no effect if the button also contains the NOFRAME attribute.

The following example creates a button in the lower-left corner of a full-screen form, with the label "New" in the Standard font:

```
BUTTON "New" ID 1000 AT (1 BOTTOM@159 AUTO AUTO) FONT 0
```

Notice that the left of the button starts at a coordinate of 1 instead of 0. This ensures that the edge of the screen does not clip the left border of the button.

As of PilRC version 2.5c, it is not possible to create graphic buttons using PilRC.

Check boxes

A check box definition looks like this:

```
CHECKBOX <label> ID <resourceID> AT (<Left> <Top> <Width>
    <Height>) [USABLE|NONUSABLE] [FONT <font number>]
    [GROUP <group ID>] [CHECKED]
```

You may assign a check box to a mutually exclusive group by setting the GROUP attribute. Of the check boxes in a form that share the same GROUP number, only one may be checked at a time.

The CHECKED attribute controls the default behavior of the check box. If CHECKED is present, and the application code does not explicitly change the state of the check box, the system draws the box in its checked state.

The following example creates a check box with the label "Show Completed Items," which starts out checked and displays in the Bold font:

```
CHECKBOX "Show Completed Items" ID 1001 AT (0 80 AUTO AUTO)
    FONT 1 CHECKED
```

Fields

A text field definition looks like this:

```
FIELD ID <resourceID> AT (<Left> <Top> <Width> <Height>)
    [USABLE|NONUSABLE] [LEFTALIGN|RIGHTALIGN]
    [FONT <font number>] [EDITABLE|NONEDITABLE] [UNDERLINED]
    [SINGLELINE|MULTIPLELINES] [DYNAMICSIZE]
    <MAXCHARS <number>> [AUTOSHIFT] [NUMERIC]
```

The LEFTALIGN and RIGHTALIGN attributes control how text is justified in the field. LEFTALIGN justifies text to the left of the field, and RIGHTALIGN justifies to the right edge of the field. Note that RIGHTALIGN works only in a non-editable field; if EDITABLE is set, a field is automatically left-justified.

EDITABLE and NONEDITABLE control whether or not the field accepts user input. If EDITABLE, the user may enter text into the field via Graffiti or the on-screen keyboard. PilRC assumes the field is editable if you omit both of these attributes.

When you include UNDERLINED in a field's definition, the system draws the field with a gray underline. Editable text fields should always be underlined; an empty field with no underline is effectively invisible to the user.

The SINGLELINE and MULTIPLELINES attributes control whether a field is a single-line field or a multiline field, respectively. See the section on creating fields in Constructor, earlier in this chapter, for the differences between single-line and multiline fields.

Including the DYNAMICSIZE attribute causes the field to put a `fldHeightChanged Event` onto the queue whenever user input causes the field to scroll. You can intercept the `fldHeightChangedEvent` in your code to resize the field to fit its contents.

You must include a value for MAXCHARS for a field to work properly. MAXCHARS specifies the maximum number of characters the field may contain.

The AUTOSHIFT attribute tells the system to perform Graffiti autoshifting in this field on Palm OS version 2.0 or later.

Include the NUMERIC attribute to restrict a field to numbers and the location-specific decimal point specified in the Palm device's global format preferences.

The AUTO keyword does not work for a field because AUTO takes its cues from the text contained in a resource, and field resources do not contain any text at design time. Always explicitly declare the width and height of a text field.

The following example creates an editable multiline field that can hold 1KB (1024 characters) of text. This field covers most of the screen, leaving just enough room on the right side for a scroll bar control. The height of this field allows it to display eleven lines of text in the Standard font.

```
FIELD ID 1003 AT (0 16 153 121) EDITABLE UNDERLINED
    MULTIPLELINES MAXCHARS 1024 AUTOSHIFT
```

Form bitmaps

A form bitmap definition looks like this:

```
FORMBITMAP AT (<Left> <Top>) BITMAP <bitmapID>
    [USABLE|NONUSABLE]
```

To include a bitmap resource on a form, you must anchor it to that form with a form bitmap resource. The `BITMAP` attribute should specify the resource ID of a bitmap resource, defined elsewhere in the same `.rcp` file.

The following example places a bitmap with resource ID 100 in the upper-left corner of a form, just below the title bar:

```
FORMBITMAP AT (0 16) BITMAP 100
```

Gadgets

A gadget definition looks like this:

```
GADGET ID <resourceID> AT (<Left> <Top> <Width> <Height>)
    [USABLE|NONUSABLE]
```

The following example creates a gadget resource that occupies most of the screen, leaving just enough room at the bottom of the form for some buttons:

```
GADGET ID 1004 AT (0 16 160 120)
```

Graffiti shift indicator

A Graffiti shift indicator definition looks like this:

```
GRAFFITISHIFTINDICATOR AT (<Left> <Top>)
```

The following example places a Graffiti shift indicator in the lower-right corner of the form, which just happens to be the conventional place to put the indicator:

```
GRAFFITISHIFTINDICATOR AT (140 147)
```

Labels

A label definition looks like this:

```
LABEL <label text> ID <resourceID> AT (<Left> <Top>)
    [USABLE|NONUSABLE] [FONT <font number>]
```

Notice that a label does not have `Height` and `Width` attributes. The size of a label is entirely dependent on its text contents. You may insert newline characters into the string with the character sequence `\n`, and for readability, you may continue long lines of text by appending a backslash character (`\`).

The following example creates a label in the Bold font at the top of the form:

```
LABEL "Please enter your name\n"\
      "below:" AUTOID AT (0 16) FONT 1
```

Lists

A list definition looks like this:

```
LIST <Item 1> <Item 2> ... ID <resourceID>
    AT (<Left> <Top> <Width> <Height>) [USABLE|NONUSABLE]
    [VISIBLEITEMS <number>] [FONT <font number>]
```

You can add items to a list in PilRC by simply listing each one in double quotes, separated by spaces, at the beginning of the `LIST` directive.

The `VISIBLEITEMS` attribute specifies the number of items in the list that are visible at once. The user may scroll the list to view items in excess of the `VISIBLEITEMS` number.

Unlike the Visible Items property in Constructor, setting `VISIBLEITEMS` to 0 in PilRC does not produce a list that displays all of its items at once. Rather, it results in an unusable list, as does setting the attribute to a value of 1. Always set `VISIBLEITEMS` to a value greater than 1.

Using the `AUTO` keyword for a list's `Width` can produce undesired results because it makes the list only as wide as the first item in the list's definition. If your list's first item is shorter than another item in the list, the result list will not be wide enough to display the longer list item properly.

On the other hand, using `AUTO` for the `Height` attribute is much easier than setting the `Height` manually, provided that you have also specified a non-zero value for `VISIBLEITEMS`. The Palm OS is perfectly happy to draw a list exactly the height you ask for, regardless of whether or not that height matches the number of visible items. Using `AUTO` to specify `Height` can save you hours of frustration caused by trying to match the height of the list in pixels to its height in visible rows.

To create a pop-up list, you must make both a list resource and a pop-up trigger. The list should have the `NONUSABLE` attribute to prevent the system from drawing it until the user taps its attached pop-up trigger. See the next section, which discusses pop-up triggers, for more details.

The following example creates a list containing six items that can display only three of those items at a time. Notice the left coordinate is at 1 instead of 0. Like buttons, lists drawn at the left edge of the screen must be moved a pixel to the right to keep the edge of the form from chopping off the left side of the list.

```
LIST "Anchovies" "Cheese" "Pepperoni" "Olives" "Mushrooms"
    "Sausage" ID 1005 AT (1 50 80 AUTO) VISIBLEITEMS 3
```

Pop-up triggers

A pop-up trigger definition looks like this:

```
POPUPTRIGGER <label> ID <resourceID> AT (<Left> <Top>
```

```
    <Width> <Height>) [USABLE|NONUSABLE]
    [LEFTANCHOR|RIGHTANCHOR] [FONT <font number>]

POPUPLIST <trigger ID> <list ID>
```

A pop-up trigger requires two definitions — one for the trigger itself, and another to tie that trigger to a list resource. Define the pop-up trigger with a `POPUPTRIGGER` directive, and then use `POPUPLIST` to connect the resource ID of the pop-up trigger with the resource ID of a list resource.

Be sure to set a pop-up trigger's `label` to a string at least as long as the longest value the trigger's attached list will contain. If the label is too short, the application can crash the device.

The `LEFTANCHOR` and `RIGHTANCHOR` attributes determine which side of a pop-up list remains in place when the trigger's label changes. `LEFTANCHOR` nails down the left end of the trigger, and `RIGHTANCHOR` keeps the right side from moving.

The following example creates a pop-up list in the upper-right corner of the screen, similar to the category trigger used in some of the built-in applications. The width setting for the trigger is 0, working from the assumption that the application will initialize the trigger's value when the form opens.

```
POPUPTRIGGER "..............." ID 2000 AT (160 0 0 13)
    RIGHTANCHOR
LIST "Business" "Personal" "Unfiled" ID 3000 AT (86 1 72 AUTO)
    NONUSABLE VISIBLEITEMS 3
POPUPLIST 2000 3000
```

Push buttons

A push button definition looks like this:

```
PUSHBUTTON <label> ID <resourceID> AT (<Left> <Top> <Width>
    <Height>) [USABLE|NONUSABLE] [FONT <font number>]
    [GROUP <group ID>]
```

Push buttons may be grouped so that only one push button in the group may be selected at a time. All push buttons sharing the same `GROUP` number are part of such an exclusive grouping.

The following example creates a row of push buttons for selection of an integer value between one and five, similar to the buttons used in the To Do application's Details dialog box to set the priority for a task:

```
PUSHBUTTON "1" ID 5001 AT (1 80 AUTO AUTO) GROUP 1
PUSHBUTTON "2" ID 5002 AT (PREVRIGHT+1 PREVTOP AUTO AUTO)
    GROUP 1
PUSHBUTTON "3" ID 5003 AT (PREVRIGHT+1 PREVTOP AUTO AUTO)
    GROUP 1
PUSHBUTTON "4" ID 5004 AT (PREVRIGHT+1 PREVTOP AUTO AUTO)
```

```
    GROUP 1
PUSHBUTTON "5" ID 5005 AT (PREVRIGHT+1 PREVTOP AUTO AUTO)
    GROUP 1
```

As of PilRC version 2.5c, it is not possible to create graphic push buttons using PilRC.

Repeating buttons

A repeating button definition looks like this:

```
REPEATBUTTON <label> ID <resourceID> AT (<Left> <Top> <Width>
    <Height>) [USABLE|NONUSABLE] [FRAME|NOFRAME]
    [BOLDFRAME] [FONT <font number>]
```

The settings for repeating buttons do the same things as the settings for normal buttons; they differ only in how they behave at run time. Repeat buttons are ideal for scroll arrows because the user may hold the stylus on one to move through large amounts of data.

The following example creates a pair of vertical scroll arrows in the lower-right corner of a form, much like the scroll arrows in the built-in applications. Notice the use of octal numbers in the buttons' labels to specify special characters from the Palm OS Symbol 7 font.

```
REPEATBUTTON "\001" ID 2000 AT (147 144 13 8) FONT 5 NOFRAME
REPEATBUTTON "\002" ID 2001 AT (147 152 13 8) FONT 5 NOFRAME
```

As of PilRC version 2.5c, it is not possible to create graphic repeating buttons using PilRC.

Scroll bars

A scroll bar definition looks like this:

```
SCROLLBAR ID <resourceID> AT (<Left> <Top> <Width> <Height>)
    [USABLE|NONUSABLE] VALUE <number> MIN <number> MAX <number>
    PAGESIZE <number>
```

A scroll bar should always have a width of 7 pixels. The `AUTO` keyword does not work for this purpose because `AUTO` is intended for controls that have text labels.

The `VALUE` attribute sets the initial location of the scroll car. `MIN` is the value of the scroll bar when the car is all the way at the top, and `MAX` is the value at the bottom of the scroll bar. `PAGESIZE` controls how many units the scroll car moves when the user taps in the gray area above or below the car.

Unlike many attributes in PilRC, which are optional, you must set the `VALUE`, `MIN`, `MAX`, and `PAGESIZE` attributes in your `.rcp` file. Otherwise, the scroll bar will not appear on the form.

The following example creates a scroll bar suitable for use beside the multiline text field example from earlier in the chapter. Its values range between 0 and 100, its scroll car starts at position 40, and tapping the gray areas of the bar moves the car 10 units at a time.

```
SCROLLBAR ID 7000 AT (153 18 7 121) VALUE 40 MIN 0 MAX 100
    PAGESIZE 10
```

Selector triggers

A selector trigger definition looks like this:

```
SELECTORTRIGGER <label> ID <resourceID> AT (<Left> <Top>
    <Width> <Height>) [USABLE|NONUSABLE]
    [LEFTANCHOR|RIGHTANCHOR] [FONT <font number>]
```

Just like a pop-up trigger, the `LEFTANCHOR` and `RIGHTANCHOR` attributes of a selector trigger set which end of the trigger stays put when alterations to the trigger's label change its width.

Be sure that a selector trigger's label is long enough to contain the longest string you expect the trigger to have to display. The label not only sets the initial appearance of the selector trigger, it also allocates enough memory to hold whatever text the label may display during execution, and a label that is too short can crash the Palm OS.

The following example creates a selector trigger suitable for use as a date selector with the standard Palm OS date picker:

```
SELECTORTRIGGER "DAY MM/DD/YY" ID 1009 AT (56 34 AUTO AUTO)
    LEFTANCHOR
```

Tables

A table definition looks like this:

```
TABLE ID <resourceID> AT (<Left> <Top> <Width> <Height>)
    ROWS <number> COLUMNS <number> COLUMNWIDTHS <width 1>
    <width 2> ...
```

The `ROWS` and `COLUMNS` attributes of a table contain the number of rows and number of columns in the table, respectively. To set the widths of individual columns, list the widths separated by spaces after the `COLUMNWIDTHS` attribute.

The following example creates a table with five columns, similar to the table used as the mail list in the To Do application. Its height is sized to allow for 11 rows, each row 11 pixels tall.

```
TABLE ID 1010 AT (0 18 160 121) ROWS 11 COLUMNS 5
    COLUMNWIDTHS 12 10 96 27 24
```

Summary

In this chapter, you learned how to create forms and the user interface elements that they contain. After reading this chapter, you should understand the following:

- You use the *form layout window*, the *hierarchy window*, and the *catalog window* to create and edit resources in Metrowerks Constructor.
- *Increment arrows*, which are standard button or repeating button resources without frames, contain a single character from one of the Palm OS symbol fonts.
- PilRC provides several keywords to simplify arranging and sizing objects on a form.

✦ ✦ ✦

CHAPTER 8

Building Menus

In This Chapter

Building menus with Constructor

Building menus with Rez

Building menus with PilRC

Introduction to the Librarian sample application

Menus provide a way for users to access application commands that see less frequent use. Because they remain tucked away out of sight until the user needs them, menus also allow your application to perform more functions without your having to make use of valuable screen real estate.

The Metrowerks tools offer two different ways to create menus. You can visually create menu resources in Constructor, or you can compile menu resources from text files with the Rez compiler. The GNU PRC-Tools compile menu resources using PilRC. This chapter shows you how to construct menu resources using all three tools.

Building Menus with Constructor

The system of menus for a particular form consists of two kinds of resources: a *menu bar*, and one or more *menu* resources. A menu bar serves as a container for menus, holding the title of each individual menu. Tapping the title of a menu in the menu bar displays that menu's contents, just as in most of the graphical interfaces used on desktop computers.

To create a new menu bar, select the gray Menu Bar line in the project window, and then choose Edit➪New Menu Bar Resource or press Ctrl+K. Double-clicking the name of a menu bar resource in the project window, or selecting one and pressing Enter, opens the *menu bar editor window*. When first opened, the menu bar editor window is completely blank. Figure 8-1 shows the menu bar editor window, both before and after adding menus to a menu bar.

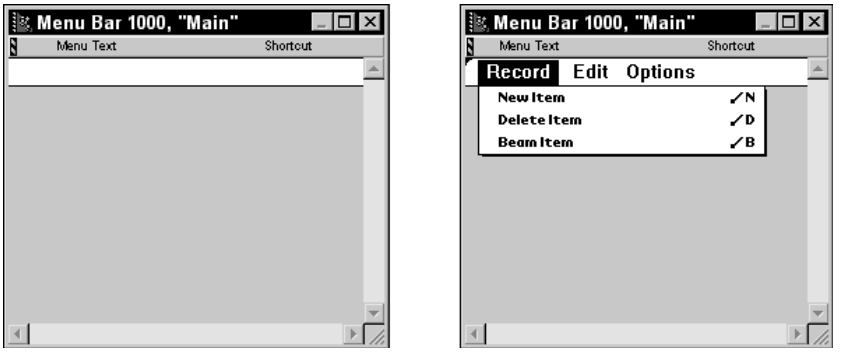


Figure 8-1: The menu bar editor window, as it appears when it first opens (left) and after adding menus (right)

You may add a new menu to a menu bar by selecting Edit➪New Menu or pressing Ctrl+M. Constructor creates a menu called "untitled." To change the menu title, click it once, and then edit the highlighted text. If you find you have created a menu in error, you may remove it by selecting the menu title, and then choosing Edit➪Remove Menu. You can also change the sequence in which menus appear in the menu bar by dragging them to the appropriate location with the mouse.

In general, clicking any item in the menu bar editor window, whether it is a menu title or a menu item, puts that text into *edit mode*. While in edit mode, typing changes the text of the item, and Constructor's Edit menu displays different commands that are associated with text editing. Edit mode pops up by default when you click an item, which can be a nuisance if you want only to select the item instead of edit its contents. To depart from edit mode and leave the item selected, simply press Enter.

Once you have a new menu, you need to add items to it. Make sure the appropriate menu is selected, and then choose Edit➪New Menu Item or press Ctrl+K to create a new item. When the new menu item appears, type the text that should appear in that item.

You may also enter a Graffiti command shortcut for the menu item. While the item's text is in edit mode, press the Tab key, and a highlighted area appears to the right of the menu item. Type the character you wish to use for this menu item's command shortcut.

If a menu item is selected but is not currently in edit mode, pressing Tab causes the item to enter edit mode. From there, you can simply press Tab a second time to add a command shortcut.

Besides normal menu items, you may also add *separator lines* to visually group menu items. To add a separator line, select Edit➪New Separator Item or press Ctrl+-. You may also turn any normal menu item into a separator by changing its text caption to a single hyphen (-).

To reorder a menu, you may drag menu items with the mouse to new locations within the current menu, or by dragging an item over the menu titles, you may move an item from one menu to another. You may also delete a selected menu item by choosing Edit➪Clear Menu Item or pressing the Backspace or Delete keys.

If an item is in edit mode, the Backspace and Delete keys edit the item's text instead of removing the item from the menu, and the Clear Menu Item command does not appear in the Edit menu. Press Enter to leave edit mode, and then try removing the menu item.

Adding Command Shortcuts to Menus

Command shortcuts are a great way to make menu commands quickly accessible to users who are adept at Graffiti. A short command stroke, followed by a single Graffiti character, activates the appropriate menu command without having to open and navigate the menu itself.

You should include command shortcuts for any menu item that might see frequent use. The standard text-editing commands of the Edit menu are a good example; they allow the user to quickly cut, copy, and paste text without moving the stylus from the Graffiti entry area of the screen. Commands that create new records or delete existing records are also good candidates for command shortcuts. However, anything that the user is less likely to use on a regular basis should not have a command shortcut. The best example of a command that does not require a shortcut is a menu item to display an application's about box.

The Palm OS built-in applications make use of several common shortcuts. Consider adding these shortcuts to your own application to make its interface consistent with the Palm applications.

Menu	***Command***	***Shortcut***
Record	New <item>	N
	Delete <item>	D
	Beam <item>	B
Edit	Undo	U
	Cut	X
	Copy	C
	Paste	P
	Select All	S
	Keyboard	K
	Graffiti Help	G
Options	Preferences	R

When you add a menu in the menu bar editor window, Constructor automatically creates a separate menu resource, gives it its own resource ID, and associates it with its menu bar. When you close the menu bar editor window, the project window updates to display any newly created menus in its Menus category. You may edit each menu individually by double-clicking its name in the project window, or by selecting it and pressing Enter, which opens the *menu editor window*. The menu editor window operates just like the menu bar editor, except that it can display only a single menu at a time. Because you can perform all the necessary menu-editing functions from the menu bar editor window, editing menus individually is never necessary.

Manually changing the resource ID of a menu resource can corrupt the menu bar resources that Constructor generates. Leave assignment of menu resource IDs to Constructor.

If you use the Auto Generate Header File option in the project window to create constant definitions for your resource IDs, be careful to name menu resources differently. Constructor uses the names displayed in the project window to create the constants, and if two menus share the same name in the project window, the automatically generated header will define the same constant twice. The best way to avoid this is to change the name of each menu so it starts with the name of the form it appears in. For example, if an application has two menus titled "Record," appearing in both the List and Edit views of an application, name the menus "List Record" and "Edit Record." When Constructor makes the header file, it then creates two constants called `ListRecordMenu` and `EditRecordMenu`.

Sharing Menus between Menu Bars

Though the process is not intuitive, you can share a menu between two or more menu bars. Menus that are common to more than one form in an application, such as the Edit menu for editing text in fields, are good candidates for sharing. Sharing a menu prevents you from having to create it twice in Constructor, and your application code can deal with the common menu code in one place instead of your having to repeat similar code in multiple locations throughout your program's source.

To share a menu between multiple menu bars, follow these steps:

1. Create the menu bar resources, without adding any menus to them.
2. Open one of the menu bars and create the shared menu.
3. Close the menu bar containing the shared menu. Take note of the new menu resource that appears in the project window; this is the resource you will share with other menu bars.
4. Open another menu bar that should contain the shared menu. Position the menu bar editor window so that both it and the project window are visible.

5. Drag the menu resource from the project window into the menu bar editor window.
6. Close the menu bar editor window.
7. To share the menu with more menu bars, repeat Steps 4 through 6 for each menu bar that should contain the shared menu.

Now, any changes to the shared menu will be reflected in all the menu bars containing that menu. If you have the Auto Generate Header File option checked in the project window, Constructor defines a single constant to represent the shared menu, which you can use throughout your code to refer to that menu, no matter which menu bar is currently displayed in the application.

If you find that you have shared a menu with a menu bar by mistake, you may remove the shared menu by selecting it in the menu bar editor window and choosing Edit➪Remove Menu. Removing a menu in this fashion, whether it is shared or not, removes only a reference to that menu resource from the menu bar; the actual menu resource remains in the Menus category of the project window. To permanently delete a menu, select it in the project window and choose Edit➪Clear Resource, or press the Delete or Backspace keys.

Building Menus with Rez

Creating menus with Constructor can be somewhat frustrating. The menu editing windows switch into edit mode at inopportune moments, and all the commands for creating and editing menus are tucked away in Constructor's menu system or hidden in obscure keyboard shortcuts. Constructor's critics often point to its quirky menu-editing interface as proof of its inferiority as a resource creation tool.

Fortunately, the Rez compiler built into CodeWarrior provides an alternate method of creating menus. Rez is a tool from the Macintosh Programmer's Workshop (MPW), which generates resources for Mac OS programs. With a little bit of hacking, you can use Rez to compile text files into Palm OS menu resources. Besides avoiding the difficult Constructor interface, a textual method of defining resources can have some other advantages over creating resources graphically, such as easier localization into other languages. Creating menus with Rez also allows you to assign your own resource IDs for menus, which Constructor's automatic resource ID creation does not allow, which means you do not have to second-guess what resource IDs Constructor will create for you.

The Rez tool compiles text files with an extension of `.r`. Before you can start making menu resources, though, you need to define what menu bar and menu resources look like. Listing 8-1 shows the contents of `MenuDefs.r`, a file that tells Rez how to make menu resources.

Listing 8-1: **The MenuDefs.r file**

```
type 'MENU'
{
    integer SYS_EDIT_MENU = 10000;  // Base menu ID
    fill byte[12];
    pstring;                        // Menu title
    array
    {
        pstring SEPARATOR = "-";    // Item text
        fill byte;
        char NONE = "\$00";         // Graffiti shortcut
        fill byte[2];
    };
    byte = 0;                       // Terminator
};

type 'MBAR'
{
    integer = $$CountOf(menus);
    array menus
    {
        integer;                    // Menu ID
    };
};
```

If you include `MenuDefs.r` in the `.r` file that describes your application's menus, Rez will understand how to make menu bars and menus. Use a standard C/C++ `#include` statement at the head of an `.r` file to include `MenuDefs.r`:

```
#include "MenuDefs.r"
```

You may also include standard C/C++ header files (`.h` extension) to define constants for the resource IDs of your menus.

To define a menu resource, use the following syntax:

```
resource 'MENU' (resource ID) {
    base ID,
    "Menu Title",
    {
        "First Menu Item", "Shortcut";
        "Second Menu Item", "Shortcut";
        ...
    }
};
```

The `resource ID` parameter is either the actual resource ID of the menu, or a constant representing the resource ID. Every menu also needs a `base ID`. The resource ID of the first item in the menu is equal to the base ID value; the resource ID of each menu item after that increases by one. For example, if the base ID for a menu is `2001`, the first item in the list has a resource ID of `2001`, the second item `2002`, the third `2003`, and so on.

A menu has a `Menu Title`, which is the string that appears in the menu bar to allow selection of that particular menu. Each menu item consists of a text caption that will appear in the menu, followed by a single character for use as a Graffiti command shortcut. You may omit the command shortcut character for menu items that do not have a shortcut. To make a menu item with no command shortcut, substitute `NONE` for the shortcut. You may also add a separator line to a menu by replacing a menu item's caption with `SEPARATOR`.

As an example, the following code creates a standard Edit menu:

```
resource 'MENU' (1001) {
    1001,
    "Edit",
    {
        "Undo",           "U";
        "Cut",            "X";
        "Copy",           "C";
        "Paste",          "P";
        "Select All",     "S";
        SEPARATOR,        NONE;
        "Keyboard",       "K";
        "Graffiti Help", "G";
    }
};
```

In the preceding example, the Undo menu item has a resource ID of `1001`, equal to the base ID of `1001` specified at the top of the menu's definition. Cut has a resource ID of `1002`, Copy has a resource ID of `1003`, and so on down the list. Note that the separator line counts in the sequential listing of resource IDs, so the Keyboard item in the preceding example has a resource ID of `1007`.

Once you have defined the menus for a menu bar, you need to define the menu bar itself. Menu bar syntax looks like this:

```
resource 'MBAR' (resource ID) {
    {first menu, second menu, ...}
};
```

Here, the `resource ID` is the resource ID of the menu bar itself. Enter the resource IDs of the menus to include in the menu bar in the comma-separated list that makes

up the body of the menu bar definition. The following example creates a menu bar resource containing three menus, whose resource IDs are 1001, 1011, and 1021:

```
resource 'MBAR' (1000) {
    {1001, 1011, 1021}
}
```

Integrating Rez Menus with Your Project

Once you have assembled your menus textually, you need to add to your project the .r file containing their definitions. You may add the file as you would any other source file in the CodeWarrior IDE, with the Project➪Add Files command. If you use the settings included in a Palm OS stationery project, CodeWarrior automatically compiles .r files using Rez.

If you create your menus textually, Constructor cannot automatically generate constant definitions for your menu resource IDs. Instead, make your own header file full of constant definitions for the menus in your project, and then include that file in both your .r menu definition file and your .c source file. Listing 8-2 shows a header file, Menus.h, containing constant definitions for a simple project.

Listing 8-2: **Menus.h, a header file for menu resource constants**

```
#define MainMenuBar         1000

#define MainRecordMenu      1001
#define MainEditMenu        1011
#define MainOptionsMenu     1021

#define RecordBase          2001
#define RecordNewItem       2001
#define RecordDeleteItem    2002
#define RecordBeamItem      2003

#define EditBase            2101
#define EditUndo            2101
#define EditCut             2102
#define EditCopy            2103
#define EditPaste           2104
#define EditSelectAll       2105
// Placeholder for separator line
#define EditKeyboard        2107
#define EditGraffitiHelp    2108

#define OptionsBase         2201
#define OptionsAbout        2201
```

A sample menu definition file, `Menus.r`, that makes use of the constants defined in `Menus.h`, appears in Listing 8-3. The `Menus.r` file contains definitions for only a single menu bar containing three menus, but when cross-referenced with its header file, `Menus.r` should give you a good example of one way to number your menu resources.

Listing 8-3: **Menus.r, a sample menu definition file**

```
#include "MenuDefs.r"
#include "Menus.h"

resource 'MENU' (MainRecordMenu) {
    RecordBase,
    "Record",
    {
        "New Item",    "N";
        "Delete Item", "D";
        "Beam Item",   "B";
    }
};

resource 'MENU' (MainEditMenu) {
    EditBase,
    "Edit",
    {
        "Undo",          "U";
        "Cut",           "X";
        "Copy",          "C";
        "Paste",         "P";
        "Select All",    "S";
        SEPARATOR,       NONE;
        "Keyboard",      "K";
        "Graffiti Help", "G";
    }
};

resource 'MENU' (MainOptionsMenu) {
    OptionsBase,
    "Options",
    {
        "About This App", NONE;
    }
};

resource 'MBAR' (MainMenuBar) {
    {MainRecordMenu, MainEditMenu, MainOptionsMenu}
};
```

Figure 8-2 shows what the menu bar described in `Menus.r` looks like at run time.

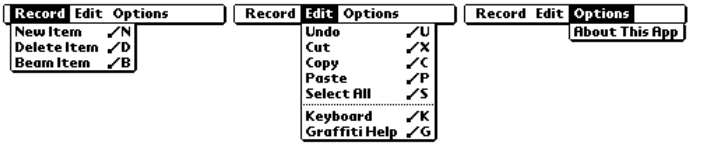


Figure 8-2: The menu bar generated from Menus.r

Building Menus with PilRC

Creating menus in PilRC is similar to using Rez in a CodeWarrior project; you define the resources in a text file, and then compile them. PilRC simply uses a different format from Rez to describe menu resources. Also, you can include menu definitions in the same `.rcp` file as the rest of your project's resources, instead of separating them as you must do in a CodeWarrior project.

PilRC creates a menu bar and the menus it contains within the same statement. The syntax for creating menus in PilRC looks like this:

```
MENU ID <resourceID>
BEGIN
    PULLDOWN <menu title>
    BEGIN
        MENUITEM <item text> ID <resourceID> [shortcut]
        MENUITEM SEPARATOR
        ...
    END
    PULLDOWN <another menu title>
    ...
END
```

The `resourceID` in the first line of a menu definition is the resource ID for the entire menu bar. Resource IDs specified on `MENUITEM` lines are the resource IDs of individual menu items.

Each menu may contain one or more `PULLDOWN` sections. Each `PULLDOWN` represents a single menu full of menu items. The `menu title` parameter specifies the text that appears within the menu bar at the top of a particular menu.

A `MENUITEM` consists of the text that will appear in that item's line of the menu (`item text`), a resource ID for that menu item (`resourceID`), and an optional Graffiti shortcut character (`shortcut`). Alternatively, you may use the line `MENUITEM SEPARATOR` to put a separator line into the menu at that location.

As yet another alternative, you may create a separator line by supplying a single hyphen (`-`) for a menu item's `item text` parameter; you might see the hyphen technique in older source code because earlier versions of PilRC did not have the `SEPARATOR` keyword.

Notice that PilRC doesn't assign resource IDs to menus, only to menu bars and individual menu items. However, this is not a problem, because menu-handling routines in the Palm OS need to deal only with the resource IDs of menu items.

Listing 8-4 shows the file `Menus.rcp`, which uses PilRC syntax to describe the same menu bar shown in Figure 8-2. The `Menus.rcp` file uses the constant definitions for resource IDs in the `Menus.h` file, shown earlier in Listing 8-2.

Listing 8-4: **Menus.rcp, a menu bar definition in PilRC**

```
#include "Menus.h"

MENU ID MainMenuBar
BEGIN
    PULLDOWN "Record"
    BEGIN
        MENUITEM "New Item"    ID RecordNewItem    "N"
        MENUITEM "Delete Item" ID RecordDeleteItem "D"
        MENUITEM "Beam Item"   ID RecordBeamItem   "B"
    END
    PULLDOWN "Edit"
    BEGIN
        MENUITEM "Undo"         ID EditUndo         "U"
        MENUITEM "Cut"          ID EditCut          "X"
        MENUITEM "Copy"         ID EditCopy         "C"
        MENUITEM "Paste"        ID EditPaste        "P"
        MENUITEM "Select All"   ID EditSelectAll    "S"
        MENUITEM SEPARATOR
        MENUITEM "Keyboard"     ID EditKeyboard     "K"
        MENUITEM "Graffiti Help" ID EditGraffitiHelp "G"
    END
    PULLDOWN "Options"
    BEGIN
        MENUITEM "About This App" ID OptionsAbout
    END
END
```

Introducing Librarian, a Sample Application

After reading this chapter, and the two that precede it (Chapter 6, "Creating and Understanding Resources," and Chapter 7, "Building Forms"), you should have a good grasp of how to create all the resources for a Palm OS application. At this point, I will introduce Librarian, a sample application I wrote that appears in this and later chapters to illustrate various points of developing for the Palm OS.

The Librarian source code, for both CodeWarrior and GCC, is available on this book's CD-ROM.

Librarian is a database for book collectors. Not only does it store the vital information about books in a collection, such as titles and authors but, because it runs on a handheld device, Librarian also allows a user to enter new books into the handheld while looking at them on the shelves, instead of dragging a pile of them to a desktop computer. The application can also serve as a wish list of books not yet purchased, and it keeps track of books loaned to or borrowed from other people.

Trying to fit every nuance of every part of the Palm OS into a single application would produce a bloated monster of an application that would be difficult to use and would require hideous amounts of storage space on a Palm OS device. Instead, Librarian focuses more on providing the right interfaces for given tasks, following Palm Computing user interface and programming guidelines as closely as possible. In fact, you will notice that Librarian operates in similar fashion to the built-in Address Book application, from which Librarian draws much of its user interface.

That said, the Librarian program uses functions from throughout the Palm OS, so it gives a good general picture of how the different parts of the OS work together in an application. Elements of the Palm OS not included in Librarian, such as serial communications, have their own, smaller examples in this book.

Displaying Multiple Records in List View

The primary display in Librarian is the List view, shown in Figure 8-3. The List view displays multiple books in Librarian's database at once, providing a lot of information at a glance. When Librarian starts, this is the form it first displays.

Figure 8-3: Librarian's List view

A large table resource occupies most of the screen space in the List view. Scroll arrows in the lower right corner of the screen allow the user to scroll the table to display more records; alternatively, pressing the scroll up and scroll down hardware buttons on the device serves the same function. Tapping a book's row in the table shows an expanded view of that particular book.

The pop-up list in the upper-right corner controls which category the List view displays. Users may assign books to different categories to organize them; selecting a category from the pop-up list restricts display in the List view to books belonging to the selected category.

Two buttons across the bottom of the screen provide access to commonly used functions. The New button opens Librarian's Edit view, described later in the "Editing a Record in Edit View" section, with a blank book record ready for entry.

The Show... button opens the Librarian Preferences dialog box, pictured in Figure 8-4. This dialog box controls what information appears in the List view. The Show In List push buttons control both the information displayed for each book and the order in which Librarian sorts the books in the List view table. By default, Librarian displays the last name of the author, followed by the author's first name and the title of the book, sorted alphabetically by the author's last name. Three check boxes toggle the display of other properties associated with each book.

Figure 8-4: The Librarian Preferences dialog box

Displaying an Individual Book in Record View

The Record view in Librarian, displayed when the user taps a book in the List view's table, shows all the information for a single book record. Figure 8-5 illustrates this form.

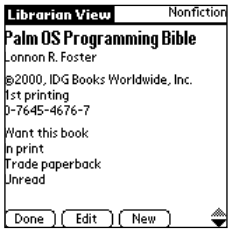
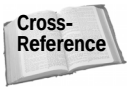


Figure 8-5: Librarian's Record view

Because the information displayed in this form varies considerably from record to record, a structured user interface element, such as a table or field, does not work. Instead, Librarian uses custom drawing routines to draw the information straight onto the screen. However, it would be nice to allow the user quick access to the contents of the displayed record by tapping anywhere within the display, so the central portion of the Record view contains a large gadget control. The gadget serves to capture a tap from the user, as well as providing a defined rectangular screen area for the custom drawing routines to work with.



The drawing routines for Librarian's Record view are described in Chapter 9, "Programming User Interface Elements."

Scroll arrows appear in the lower right of the Record view if the currently displayed book contains more information than the Record view can display at once. The Done button returns to the List view, the Edit button opens the displayed record in Edit view, and the New button opens the Edit view with a blank record, ready for the user to enter data.

Editing a Record in Edit View

Like the List view, the primary component of the Edit view is a table. The table has two columns, the left column displaying labels describing the contents of the items in the right column. Figure 8-6 shows the Edit view in Librarian.

Figure 8-6: Librarian's Edit view

The pop-up list in the upper-right corner is a standard category picker. Picking an item from the list changes the category of the currently displayed record.

Text fields in the Edit view are expandable, increasing or decreasing in height as necessary to accommodate whatever text the user enters. Besides fields, the table also contains a pair of pop-up lists and a check box to control other properties of the book.

The Done button returns to the application's List view. Tapping Details... opens the Book Details dialog box, pictured in Figure 8-7. From this dialog box, the user may

set the category of the current book record with the Category pop-up list. Checking the Private check box marks this record as private. If the user has assigned a password in the system's Security application and has chosen to hide private records, any book marked private in Librarian does not show up in the List view, or anywhere else in the application.

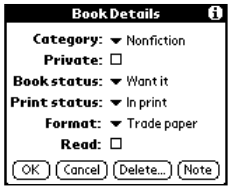


Figure 8-7: The Book Details dialog box

The Book Details dialog box's Delete... button allows the user to delete the current book from Librarian's database. When tapped, the Delete Book dialog box, shown in Figure 8-8, appears. The Delete Book dialog box not only confirms that the user really wishes to delete the book, but it also offers the user the option to save an archive copy of the book's record on the desktop via Librarian's companion conduit application.

Figure 8-8: The Delete Book dialog box

Tapping the Note button in the Book Details dialog box, or the Note button in the Edit form, opens Librarian's Note view. The Note view, shown in Figure 8-9, allows the user to enter any information about the book that does not correspond to any of the regular record fields.

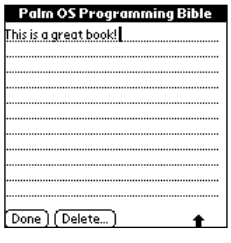
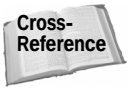


Figure 8-9: The Note view

From a programming perspective, the Note view is particularly interesting, because its resources are not part of the Librarian project itself. Instead, Librarian borrows the shared Note form used by the built-in applications. This form's resources are embedded in the ROM along with the standard Palm applications, and the Palm OS header files define various constants for use with the Note form.



Using the Note form in an application is covered in Chapter 9, "Programming User Interface Elements."

Examining Librarian's Menus

The List view has a menu bar, providing access to other less frequently accessed functions. Menus in the List view menu bar are shown in Figure 8-10.

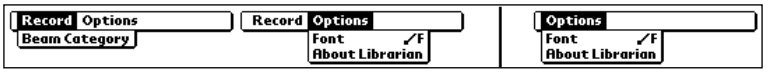
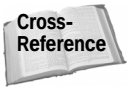


Figure 8-10: Menus in Librarian's List view

The Record menu on the left side of Figure 8-10 appears only when Librarian is running on a Palm OS device that supports infrared beaming, because the menu's only item beams the currently displayed category to another device that also has Librarian installed. To accomplish the feat of displaying different menus on different devices, Librarian actually contains two menu bar resources for the List view, one with the Record menu (left side of Figure 8-10), and one without (right side of Figure 8-10). Code within the application determines whether the device on which Librarian is running is capable of beaming, and displays the appropriate menu bar resource.



Chapter 9, "Programming User Interface Elements," details dynamically displaying menus, and Chapter 10, "Programming System Elements," shows how to determine the features supported by the device on which an application is running.

The user may choose Font... from the Options menu to open the Select Font dialog box, pictured in Figure 8-11. Like the Note view, the Select Font dialog box is another form resource stored in ROM, so Librarian does not need to define this form in its own source code. Changing the font in this dialog box causes Librarian to change the font it uses to display information in the current view, allowing a user to customize the program for easier viewing.

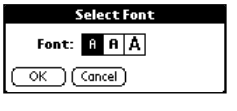


Figure 8-11: The Select Font dialog box

Also available from the Options menu is an About Librarian menu command, which displays the About Librarian dialog box pictured in Figure 8-12. This dialog box is a simple about box, showing information about the Librarian application, such as its version number.

Figure 8-12: The About Librarian dialog box

The Find Book dialog box, available from the Find... button in the List view, has a standard Edit menu to provide text-editing support for its one field.

Librarian's Record view has a menu bar, whose menus are pictured in Figure 8-13. Like the Record menu in the List view, the Record view's Record menu contains an option related to beaming records. Unlike the List view menu, the Edit view's Record menu also contains items that are not related to beaming, so Librarian has two Record menus, one with the Beam Book option, and one without. The application displays the appropriate menu for the device on which Librarian is running. The menu bar on the top of Figure 8-13 appears on devices that support IR beaming; Librarian displays the menu bar in the bottom of the figure on other devices.

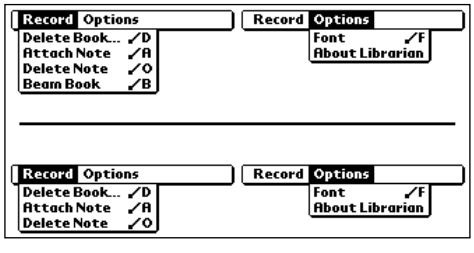


Figure 8-13: Menus in Librarian's Record view

Users can use the Record menu to delete the currently displayed book, attach a text note to it, or remove an existing note. If the device supports it, the user may also beam the current book record to another Palm OS device that has the Librarian application installed, adding the record to the Librarian database on the other device.

The Record view's menu bar also has the same Options menu used in the List view's menu bar.

In the Edit view, Librarian has similar menus to those in the Record view, with the addition of an Edit menu to provide text-editing functions. Figure 8-14 shows the Edit view's menu bar. Again, there are two different Record menus in this menu bar, one for devices with IR beaming (top) and one for devices without IR capability.

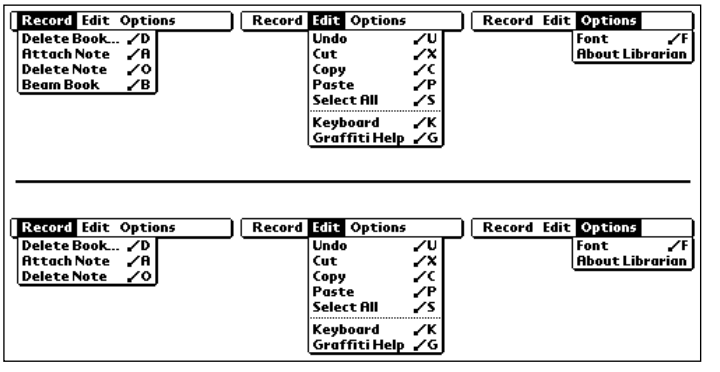
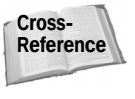


Figure 8-14: Menus in Librarian's Edit view

Librarian makes economical use of menu resources by sharing a number of common menus between different forms. The Options menu in the List view also appears in the program's two other major views, Record and Edit. Librarian shares the same Record menu between the Record and Edit views, and the Edit menu in the Edit view is the same as the Edit menu displayed from the Find Book dialog box.

Sharing menu resources in this way removes duplicate code from the project, because the application handles common menus with common routines. Such sharing reduces the complexity of the application, making it easier to debug and removing precious bytes from the size of the compiled executable.



Chapter 9, "Programming User Interface Elements," contains more information about handling common menu items.

Summary

This chapter taught you how to create menu resources using CodeWarrior and GNU PRC-Tools. After reading this chapter, you should understand the following:

- ✦ CodeWarrior allows you to make menu resources in two ways: graphically with Constructor, or textually by using CodeWarrior's built-in Rez compiler.

- Menus built in Constructor may be shared between more than one menu bar.
- You must include a `MenuDefs.r` file or its equivalent when using Rez to create resources, because Rez does not know how to make Palm OS menu resources unless you describe those resources to the compiler.
- PilRC, like Rez, compiles resources from textual descriptions.

✦ ✦ ✦

CHAPTER 9

Programming User Interface Elements

In This Chapter

Programming alerts

Programming forms and form objects

Programming grouped check boxes and push buttons

Programming date and time picker dialog boxes

Programming fields

Programming gadgets

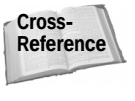
Programming lists and pop-up lists

Programming menus

Drawing graphics and text

The Palm OS involves a lot of user interaction. Most Palm OS applications serve to collect and display data, both of which functions require thoughtful user interface design and implementation to perform these functions effectively on such a small device. Earlier chapters showed you how to design and build the resources that make up a Palm OS application's user interface. This chapter shows you how to connect that interface with your application code, forming the vital link between the user and your program.

As you read through this chapter, you may notice a couple of what seem to be major omissions in the chapter's coverage of form objects: tables and scroll bars. Tables are the most complex user interface elements in the Palm OS, and as such, they warrant an entire chapter to themselves. Scrolling, while also linked to fields and lists, is most often associated with tables. Scroll bars and their cousins, repeating arrow buttons, are therefore covered more fully in the chapter on tables.



For the complete story on programming tables and scrolling behavior, see Chapter 11, "Programming Tables and Scrolling."

Programming Alerts

Alerts are the simplest way to present a dialog box, either to prompt the user for input or simply to display important information. Displaying an alert in your application is very straightforward; simply call the **FrmAlert** function, passing it the resource ID of the alert you wish to display. The **FrmAlert** function returns the number of the alert button pressed by the user. The left-most button in the alert is number 0, with each button's number increasing sequentially from left to right.

The following code displays an alert dialog box, capturing the button tapped by the user in the variable `tappedButton`.

```
UInt16 tappedButton;

tappedButton = FrmAlert(MyAlertID);
```

Depending on which button the user taps in the alert, your code can perform different actions. If the alert contains only one button, as is often the case in a purely informational alert, you may safely ignore the return value from **FrmAlert**:

```
FrmAlert(MyAlertID);
```

Otherwise, you will want to respond to the alert differently, depending on which button the user selected. The following PilRC alert definition, and the code snippet that follows, shows how to react to user input from an alert. Figure 9-1 shows the alert dialog box in action.

```
ALERT ID RPSAlert
CONFIRMATION
BEGIN
    TITLE "Mortal Combat"
    MESSAGE "Which implement of mass destruction "\
            "do you wish to wield?"
    BUTTONS "Rock" "Paper" "Scissors"
END
```

This code determines which button the user tapped and responds accordingly:

```
switch (FrmAlert(RPSAlert)) {
    case 0:
        // User tapped the "Rock" button
        break;
    case 1:
        // User tapped the "Paper" button
        break;
    case 2:
        // User tapped the "Scissors" button
        break;
    default:
        // Error
}
```

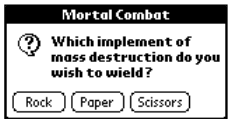


Figure 9-1: An alert dialog box with three buttons

You can also customize an alert's message text at run time. When creating the alert resource, you may insert three placeholders, `^1`, `^2`, and `^3`, which the **FrmCustomAlert** function can replace with whatever text you like. **FrmCustomAlert** takes four arguments. The first is the resource ID of the alert to display, and the second, third, and fourth arguments are character pointers to the strings that should replace `^1`, `^2`, and `^3`, respectively. If one of the placeholders is missing from the alert, you may pass `NULL` as its corresponding string argument.

Pass `NULL` as one of the string arguments only if that argument has no corresponding placeholder in the alert. For example, if the alert's message text contains `^1` and you pass `NULL` as the second argument to FrmCustomAlert, your application will crash. If you want to ignore an existing placeholder, pass the empty string (`""`) instead. On Palm OS 2.0 and earlier, the empty string also causes a crash. For those versions of the OS, pass a string containing a single space (`" "`) instead.

Also keep in mind that **FrmCustomAlert** does not substitute for placeholder strings in the title or in the button labels of an alert resource, only in the alert's message text. For example, a title string containing `^1` will still display "^1" at run time.

To illustrate **FrmCustomAlert** in action, consider the following alert resource, defined in PilRC syntax:

```
ALERT ID DeleteAlert
WARNING
BEGIN
    TITLE "Confirm Multiple Deletion"
    MESSAGE "You are about to delete ^1 records from the "\
            "^2 category. Do you wish to proceed?"
    BUTTONS "OK" "Cancel"
END
```

When the following call to **FrmCustomAlert** displays the `DeleteAlert` resource, substituting the strings `40` and `Business` for the alert's placeholders, the resulting alert dialog box is pictured in Figure 9-2.

```
tappedButton = FrmCustomAlert(DeleteAlert, "40", "Business",
    NULL)
```

Figure 9-2: A custom alert displayed with the FrmCustomAlert function

Prior to Palm OS 3.1, calling `FrmAlert(MyAlertID)` is equivalent to calling `FrmCustomAlert(MyAlertID, NULL, NULL, NULL)`. If your application is intended to run on Palm OS 3.0 or earlier, make sure any alert you call with FrmAlert does not contain the special placeholder strings (`^1`, `^2`, or `^3`). Calling FrmAlert with an alert containing placeholders on these earlier versions of the OS causes a crash.

Programming Forms

Depending on the purpose different forms in an application serve, the Palm OS offers you a number of options when it comes to displaying forms. The following scenarios outline the three most common ways your application might need to display a form:

- **Switching to a new form.** In this scenario, the application erases the current form and displays a new one in its place. This most commonly occurs when switching between major, full-screen views in an application. An example of this is changing between the List and Edit modes of the Address Book application when the user taps the **New** button on the List view's form. This situation calls for the **FrmGotoForm** function.
- **Displaying a complex modal dialog box.** Most dialog boxes have enough controls to warrant their own event handlers. If a dialog box contains more than a check box or two, you should display it using the **FrmPopupForm** function.
- **Displaying a simple modal dialog box.** Sometimes you need to display a quick prompt for user input, but you need a dialog box with a little more interactivity than an alert resource can provide. A good example of this is the delete confirmation dialog box displayed by the built-in applications when you delete a record (see Figure 8-8 in the previous chapter). Instead of simply asking you if it is okay to delete the record, the dialog box also provides a check box, giving you the option to save an archive copy of the record on the desktop. The **FrmDoDialog** function handles this situation.

Switching to a New Form

To allow users to switch between two full-screen views in an application, use the **FrmGotoForm** function. **FrmGotoForm** takes a single argument, the resource ID of a form to display. The following example calls **FrmGotoForm** to open and display a form called `EditForm`.

```
FrmGotoForm(EditForm);
```

FrmGotoForm sends a `frmCloseEvent` to the current form and then sends a `frmLoadEvent` and a `frmOpenEvent` to the form you want to display. Located at the end of the application's **EventLoop** function, the system's default form-handling function, **FrmHandleEvent**, automatically takes care of the `frmCloseEvent` by erasing and disposing of the current form. You do not need to do anything special to get rid of the current form. Your application's **ApplicationHandleEvent** function should take care of the `frmLoadEvent` by initializing the form with the **FrmInitForm** function, setting that form as the active form with **FrmSetActiveForm**, and associating an event handling function with that form by calling **FrmSetEventHandler**. Now that the form has been initialized in **ApplicationHandleEvent**, the form's event handler needs only to respond to the `frmOpenEvent` by drawing the form with **FrmDrawForm**.

Displaying a Complex Modal Dialog Box

If a modal dialog box contains controls that exhibit behavior the system cannot provide by itself, such as tables, selector triggers, or menus, you must display the dialog box using **FrmPopupForm**. Like a full-screen form, a complex modal dialog box needs its own event handling function to properly deal with user input.

Like **FrmGotoForm**, the **FrmPopupForm** function takes the resource ID of a form as an argument. Unlike **FrmGotoForm**, which sends a `frmCloseEvent` to the current form, **FrmPopupForm** just sends a `frmLoadEvent` and a `frmOpenEvent` to the form specified in the argument to **FrmPopupForm**. The following line of code pops up a form with a resource ID constant of `DetailsForm`:

```
FrmPopupForm(DetailsForm);
```

To exit the dialog box, use the **FrmReturnToForm** function. As an argument, **FrmReturnToForm** takes the resource ID of the form to return to. **FrmReturnToForm** erases the current form from the screen, deletes the form from memory, and then sets the active form to the form specified in its argument. The function assumes that the form you pass to it is already loaded into memory.

You may also pass in zero (0) as the argument to **FrmReturnToForm**, in which case the function returns to the last form that was loaded—in this case the form displayed by your application before calling **FrmPopupForm**. The following line of code, then, is all that is required to return from a modal dialog box to its parent form:

```
FrmReturnToForm (0);
```

Typically, the call to **FrmReturnToForm** should occur in those parts of the modal dialog box's event handler devoted to handling buttons on the dialog form. The following code from a dialog form's event handler closes the form when the user taps that form's OK or Cancel buttons:

```
if (event->eType == ctlSelectEvent) {
```

```
        switch (event->data.ctlSelect.controlID) {
            case OkButton:
                FrmReturnToForm (0);
                handled = true;
                break;

            case CancelButton:
                FrmReturnToForm (0);
                handled = true;
                break;

            default:
                break;
        }
    }
```

Because FrmReturnToForm does not send a `frmCloseEvent` to the current form, be sure you clean up any variables associated with that form before calling FrmReturnToForm. Failure to manually clean up such variables could result in a memory leak.

Displaying a Simple Modal Dialog Box

If a part of your application requires only a small amount of prompting from the user, such as confirmation to delete a record, an alert will usually suffice. However, the program may require a little more input than the simple buttons in an alert can provide, but not so much input that you should devote an entire event handling function to the form. In this case, the **FrmDoDialog** function can quickly display a simple dialog box.

A form for use with **FrmDoDialog** must contain only user interface elements that the system can handle without intervention from your own code. Check boxes and push buttons are ideal for this purpose, because they can store meaningful information and handle user input without any extra code. Tables and selector triggers are good examples of controls that will not work well with **FrmDoDialog,** because they will not work at all without a fair amount of help from your application's code.

In much the same way that **FrmAlert** or **FrmCustomAlert** returns the button tapped to dismiss the alert, **FrmDoDialog** returns the resource ID of the button tapped to dismiss its form. Your application's code can then react accordingly, depending on which button the user tapped.

To use **FrmDoDialog** in your application, follow these steps:

1. Initialize the form with **FrmInitForm**.
2. If necessary, set the values of the form's controls with **FrmSetControlValue**.
3. Display the form with **FrmDoDialog**.
4. Retrieve the values of the dialog box's controls using **FrmGetControlValue**.

5. Remove the form from memory with **FrmDeleteForm**.

As an example, the following PilRC resource definition from the Librarian sample application creates a delete confirmation dialog box, pictured in Figure 9-3. To provide online help for the dialog box, `DeleteBookDialog` also includes a string resource, `DeleteBookHelpStr`, which the user may display by tapping the "i" icon in the dialog box's title bar. The only thing that differentiates the dialog box from an alert is the inclusion of a check box control.

```
STRING ID DeleteBookHelpStr "The Save Archive Copy option "\
    "will store deleted records in an archive file on your "\
    "Desktop at the next HotSync operation."

FORM ID DeleteBookDialog AT (2 61 156 97)
MODAL
HELPID DeleteBookHelpStr
DEFAULTBTNID DeleteBookCancel
BEGIN
    TITLE "Delete Book"
    FORMBITMAP AT (8 19) BITMAP 10005
    LABEL "Delete selected book\nrecord?" AUTOID
        AT (32 19) FONT 1
    CHECKBOX "Save archive copy on PC" ID DeleteBookSaveBackup
        AT (12 54 AUTO AUTO) FONT 1 CHECKED
    BUTTON "OK" ID DeleteBookOK AT (5 80 AUTO AUTO)
    BUTTON "Cancel" ID DeleteBookCancel AT (PREVRIGHT+5 PREVTOP
        AUTO AUTO)
END
```

The form bitmap resource ID provided in the following example (`10005`) is the internal resource ID of the confirmation alert icon stored in the Palm OS ROM. In the headers included by `Pilot.h` you may find the following definitions for alert dialog box icon bitmaps:

```
#define InformationAlertBitmap    10004
#define ConfirmationAlertBitmap   10005
#define WarningAlertBitmap        10006
#define ErrorAlertBitmap          10007
```

Normally, it would be best to use the defined constant value instead of the raw resource ID, to prevent the application from breaking should Palm Computing change these numbers in a future release of the Palm OS. However, PilRC is somewhat limited in its ability to import header files. The resource compiler does not know how to parse the `#include` C/C++ directive, and can grab `#define` statements from only a single included header at a time. For the sake of illustration, I have used the resource ID number in the previous example instead of redefining `ConfirmationAlertBitmap` in Librarian's `librarianRsc.h` file.

Metrowerks Constructor has a somewhat worse problem, in that it cannot import header definitions at all. Constructor's only option for including resources embedded in the Palm OS ROM is to use the bare resource ID.

Figure 9-3: A simple dialog box, suitable for use with FrmDoDialog

The **DetailsDeleteRecord** function from Librarian (a sample application introduced in Chapter 8), listed in the following example, displays the dialog box shown in Figure 9-3 to prompt the user about whether the application should really delete a record from its database. Librarian calls this function when the user taps the **Delete** button while viewing a book's details, or when the user selects the **Delete Book** menu item.

```
static Boolean DetailsDeleteRecord (void)
{
    UInt16   ctlIndex;
    UInt16   buttonHit;
    FormType *form;
    Boolean  archive;

    // Initialize the dialog form.
    form = FrmInitForm(DeleteBookDialog);

    // Set the "Save archive copy on PC" check box to its
    // previous setting.
    ctlIndex = FrmGetObjectIndex(form, DeleteBookSaveBackup);
    FrmSetControlValue(form, ctlIndex, gSaveBackup);

    // Display the form and determine which button the user
    // tapped.
    buttonHit = FrmDoDialog(form);

    // Retrieve data from the dialog before deleting the form.
    archive = FrmGetControlValue(form, ctlIndex);

    // Release the form from memory.
    FrmDeleteForm(form);

    if (buttonHit == DeleteBookCancel)
        return (false);

    // Remember the value of the check box for later.
    SaveBackup = archive;

    // Code to actually delete the record omitted

    return (true);
}
```

The first thing **DetailsDeleteRecord** does is to initialize the dialog box form in memory with the **FrmInitForm** function. **FrmInitForm** takes the resource ID of a form as an argument, and it returns a pointer to the initialized form resource in memory.

Before displaying the dialog box, **DetailsDeleteRecord** needs to set the status of the dialog box's check box. In Librarian, the global variable `gSaveBackup` keeps track of whether deleted records should be archived on the PC or not. **DetailsDeleteRecord** first requires the index of the check box, which it retrieves with the **FrmGetObject Index** function. Then **DetailsDeleteRecord** sets the value of the check box via **FrmSetControlValue**.

Now **DetailsDeleteRecord** is ready to display the dialog box. The **FrmDoDialog** function displays a form, given a pointer to a form resource, and then returns the resource ID of the button the user tapped to close that form.

DetailsDeleteRecord grabs the value of the check box with **FrmGetControlValue**, and then removes the form from memory with **FrmDeleteForm**. This order of events is important; you must retrieve information from a form's controls before deleting it, because releasing the form's memory also releases the memory that holds its controls' values.

The **DetailsDeleteRecord** function returns `true` if the user tapped the dialog box's OK button, or `false` if the user tapped its Cancel button. If the user tapped OK, **DetailsDeleteRecord** also saves the dialog box's check box value to the global variable `gSaveBackup`. Because the `DeleteBookForm` defines its Cancel button as the form's default button, the operating system simulates tapping the Cancel button if the user switches applications while the dialog box is open. Therefore, switching applications also causes **DetailsDeleteRecord** to return `false`, ignoring whatever setting is in the dialog box's check box and leaving the contents of `gSaveBackup` alone.

Programming Objects on Forms

Though the objects a form may contain vary in form and function, all objects share some common ground when it comes to handling them in code. The default form handling provided by the **FrmHandleEvent** function not only takes care of visibly reacting to user input, such as inverting a button when tapped, but **FrmHandleEvent** also makes sure that the system posts events to the queue to which your application can respond.

Internally, the Palm OS treats a number of user interface elements as *controls*. The system deals with all controls using similar data structures, events, and functions. Controls consist of the following objects:

- Buttons and graphic buttons
- Push buttons and graphic push buttons

- ✦ Check boxes
- ✦ Pop-up triggers
- ✦ Selector triggers
- ✦ Repeating buttons and graphic repeating buttons
- ✦ Sliders and feedback sliders

For default processing of user input from controls, **FrmHandleEvent** actually hands control over to the **CtlHandleEvent** function, which takes care of all the specifics of user interaction with the control. **FrmHandleEvent** defers to different functions for other user interface objects: **FldHandleEvent** for fields, **LstHandleEvent** for lists, **MenuHandleEvent** for menus, **SclHandleEvent** for scroll bars, and **TblHandleEvent** for tables.

Handling Form Object Events

Most form objects respond to taps from the user in a similar fashion. When the user first taps within the borders of an object, the control queues an *enter* event. If the user then lifts the stylus while still within the object's screen boundaries, the control posts a *select* event. However, if the user drags the stylus outside the boundaries of the object before lifting, the control posts an *exit* event, instead. The repeating button and scroll bar controls generate another type of event, a *repeat* event, when the user holds the stylus down within the boundaries of those objects.

Select events are usually the most interesting to your application, because they indicate successful selection of the object. The following select events are possible:

- ✦ `ctlSelectEvent`
- ✦ `frmTitleSelectEvent`
- ✦ `lstSelectEvent`
- ✦ `popSelectEvent`
- ✦ `tblSelectEvent`

Handling an enter event allows your application to respond to a tap on an object before the user lifts the stylus from the screen. A good use for this style of event handling is to populate the contents of a pop-up list with dynamic data, as the user taps the pop-up trigger to display that list (see the "Programming Dynamic Lists" section, later in this chapter, for an example of how to dynamically populate a list). The Palm OS provides the following enter events:

- ✦ `ctlEnterEvent`
- ✦ `fldEnterEvent`

- `frmTitleEnterEvent`
- `lstEnterEvent`
- `sclEnterEvent`
- `tblEnterEvent`

Exit events allow for special processing when the user starts to select an object but decides against it and moves the stylus off the object before lifting it from the screen. The exit event is most useful in conjunction with an enter event, particularly if the enter event allocates memory for a variable or two. The exit event can then release that memory, preventing a leak. The exit events are:

- `ctlExitEvent`
- `lstExitEvent`
- `sclExitEvent`
- `tblExitEvent`

The scroll bar and repeating button objects continually post repeat events to the queue while the user holds down the stylus within their boundaries, allowing your application to scroll data dynamically until the user lifts the stylus. The two repeat events are:

- `ctlRepeatEvent`
- `sclRepeatEvent`

To respond to any of these events, your application should check for them in the appropriate form event handler. The following event handler captures select events from two buttons (`OKButton` and `CancelButton`) and a list:

```
static Boolean MyFormHandleEvent (EventType *event)
{
    Boolean  handled = false;

    if (event->eType == ctlSelectEvent) {
        switch (event->data.ctlSelect.controlID) {
            case OKButton:
                // Do something in response to the OK button
                // being tapped.
                handled = true;
                break;

            case CancelButton:
                // Do something in response to the Cancel
                // button being tapped.
                handled = true;
```

```
                break;

            default:
                break;
        }
    }

    else if (event->eType == lstSelectEvent) {
        // Do something in response to the list selection; this
        // code assumes only one list on the form.
        handled = true;
    }

    return (handled);
}
```

Because the form contains two buttons, each of which is capable of generating a `ctlSelectEvent`, the event handler in the previous example must determine which button generated the event. **MyFormHandleEvent** does this by looking at the `data` member of the event structure passed to the event handling function.

The event structure's `data` member is actually a union of many other structures, each of which holds different data depending on the type of event represented by the event structure. Each of the different structures stores the resource ID of the object that generated the event in a slightly different place. Table 9-1 shows you where to look for the resource ID of an object in an event structure, depending on the type of event involved.

Notice that **MyFormHandleEvent** assumes there is only one list in the form. If a form contains only one control capable of generating a particular event, you may forgo checking the event data for the resource ID of the object involved. Simply check for the occurrence of the event and respond to the event accordingly.

Table 9-1
Finding Resource IDs in the Event Structure

Event Type	*Where to Find the Resource ID*
`ctlEnterEvent`	`event->data.ctlEnter.controlID`
`ctlExitEvent`	`event->data.ctlExit.controlID`
`ctlRepeatEvent`	`event->data.ctlRepeat.controlID`
`ctlSelectEvent`	`event->data.ctlSelect.controlID`
`fldEnterEvent`	`event->data.fldEnter.fieldID`
`lstEnterEvent`	`event->data.lstEnter.listID`
`lstExitEvent`	`event->data.lstExit.listID`

Event Type	*Where to Find the Resource ID*
lstSelectEvent	event->data.lstSelect.listID
popSelectEvent	event->data.popSelect.controlID (ID of the pop-up trigger) event->data.popSelect.listID (ID of the attached list resource)
sclEnterEvent	event->data.sclEnter.scrollBarID
sclExitEvent	event->data.sclExit.scrollBarID
sclRepeatEvent	event->data.sclRepeat.scrollBarID
tblEnterEvent	event->data.tblEnter.tableID
tblExitEvent	event->data.tblExit.tableID
tblSelectEvent	event->data.tblSelect.tableID

Retrieving an Object Pointer

Before you can do something with an object on a form, you must retrieve a pointer to that object. The **FrmGetObjectPtr** function performs this function, given a pointer to a form and the index of the desired object. Note that the index of an object on a form is different from the object's resource ID. You will usually need to get the index number of an object with **FrmGetObjectIndex** before you can get a pointer to the object with **FrmGetObjectPtr**.

Both **FrmGetObjectIndex** and **FrmGetObjectPtr** also require a pointer to the form containing the object you want. Usually, you can simply use the **FrmGetActiveForm** function, which returns a pointer to the currently active form.

The following lines of code retrieve a pointer to the active form, and then a pointer to the button whose resource ID is `MainOKButton`:

```
FormType     *form;
ControlType  *ctl;

form = FrmGetActiveForm();
ctl = FrmGetObjectPtr(form, FrmGetObjectIndex(form,
                      MainOKButton));
```

The `ctl` variable now contains a pointer to the `MainOKButton` control, which may be manipulated using other functions.

Calling both **FrmGetObjectPtr** and **FrmGetObjectIndex** is the only way to get an object pointer from a resource ID; the Palm OS does not offer a function to directly turn a resource ID into an object pointer. Because retrieving object pointers is a very common action in programming for the Palm OS, you may wish to avoid repeating

the cumbersome **FrmGetObjectPtr** and **FrmGetObjectIndex** combination throughout your code. The following function is a wrapper for the process of converting a resource ID into a usable object pointer:

```
VoidPtr GetObjectPtr (UInt16 objectID)
{
    FormType  *form;

    form = FrmGetActiveForm();
    return (FrmGetObjectPtr(form, FrmGetObjectIndex(form,
            objectID)));
}
```

GetObjectPtr takes the resource ID of an object as its only argument and returns a pointer to that object, assuming the desired object is on the current active form. Using the **GetObjectPtr** function, you can shorten the first example in this section to the following single line of code:

```
ControlType *ctl = GetObjectPtr(MainOKButton);
```

Because it is such a useful function, you will see **GetObjectPtr** used elsewhere in this book, as well as in the built-in applications provided by Palm Computing. In fact, this function already exists in any project created from stationery in Metrowerks CodeWarrior.

The examples in this section assume you are coding in C, which does not require an explicit cast from a void pointer. In C++, you need to cast the return value of FrmGetObjectPtr and GetObjectPtr to the proper type, as in the following example:

```
ControlPtr ctl = (ControlPtr)GetObjectPtr(MainOKButton);
```

Hiding and Showing Form Objects

Sometimes, you may wish to hide user interface elements from view, or cause them to appear again, in response to user input. When hidden, an object does not display on the screen or accept any taps from the user. The Palm OS provides two functions for this purpose, **FrmHideObject** and **FrmShowObject**. As arguments, both functions take a pointer to the object's form and the object index. Note that the second argument is the object index, not its resource ID or an object pointer; you may retrieve the object index with the **FrmGetObjectIndex** function.

The following form event handler hides the `MyButton` control from view when the user selects the form's `HideCheckbox` control:

```
static Boolean MyFormHandleEvent (EventType *event)
{
    Boolean     handled = false;
    FormType    *form;
```

```
    UInt16      buttonIndex;
    ControlType *ctl;

    if (event->eType == ctlSelectEvent) {
        switch (event->data.ctlSelect.controlID) {
            case HideCheckbox:
                form = FrmGetActiveForm();
                buttonIndex = FrmGetObjectIndex(form,
                                MyButton);
                ctl = GetObjectPtr(HideCheckbox);
                if (CtlGetValue(ctl)) {
                    FrmHideObject(form, buttonIndex);
                } else {
                    FrmShowObject(form, buttonIndex);
                }
                handled = true;
                break;

            case MyButton:
                // Do something when the user taps MyButton
                handled = true;
                break;

            default:
                break;
        }
    }

    return (handled);
}
```

Notice the use of the **CtlGetValue** function to determine whether the check box is checked or not. **CtlGetValue** returns the value associated with a check box or push button control, either `1` for on or `0` for off. In the preceding code, if the value of the check box is `1` (checked), the event handler hides the `MyButton` control with **FrmHideObject**; if the check box is unchecked, **FrmShowObject** reveals the button again.

Programming Check Boxes and Push Buttons

As shown in the previous section, the **CtlGetValue** function returns the value of a check box or push button control, given a pointer to the control. A return value of `1` from **CtlGetValue** indicates a checked check box or a selected push button, whereas `0` represents an unchecked box or an unselected push button.

Besides looking at the value of a check box or push button, you may also set the value using the **CtlSetValue** function. The following lines of code select the `MyPushButton` push button control:

```
ControlType *ctl = GetObjectPtr(MyPushButton);
CtlSetValue(ctl, 1);
```

Although both CtlGetValue and CtlSetValue take a pointer to any control type, they work only with check boxes and push buttons. CtlGetValue returns an undefined value with other control types, and the system simply ignores calls to CtlSetValue for other controls.

Handling Control Groups

Check boxes and push buttons may be assigned to mutually exclusive *control groups*. Within a control group, only one check box or push button may be selected at a time. The Palm OS provides functions for setting or determining which check box or push button in a group is currently selected.

FrmSetControlGroupSelection takes three arguments: a pointer to the form containing the group of controls, the number assigned to the control group, and the resource ID of the control in that group to select. The **FrmSetControlGroupSelection** function selects the specified control and deselects all the other controls that share the same control group, saving you from having to make multiple calls to **CtlSetValue** to manually turn off all the other controls in the group.

FrmGetControlGroupSelection returns the index number of the selected control in a control group, given a pointer to the form containing the group and the group's number.

FrmGetControlGroupSelection returns the index of the selected control, but FrmSetControlGroupSelection sets the selection based on the control's resource ID. Keep this in mind as you use these functions, because passing the wrong number is a common error when handling control groups.

As an example, here is a PilRC resource definition for a form containing a group of push buttons:

```
FORM ID MainForm AT (0 0 160 160)
BEGIN
    PUSHBUTTON "Rock" ID MainRockPushButton
        AT (CENTER 60 40 12) GROUP 1
    PUSHBUTTON "Paper" ID MainPaperPushButton
        AT (PREVLEFT PREVBOTTOM+1 PREVWIDTH PREVHEIGHT) GROUP 1
    PUSHBUTTON "Scissors" ID MainScissorsPushButton
        AT (PREVLEFT PREVBOTTOM+1 PREVWIDTH PREVHEIGHT) GROUP 1
END
```

If you are using Constructor to create your application's resources, pay careful attention to the group names that Constructor automatically generates for you. Constructor assembles most resource constants from the name of the form, the word `GroupID`, and the ID you have assigned to the resource itself. When Constructor generates resource constants for group numbers, it uses the form's title instead of its name, followed by `FormGroupID` and some number that Constructor seems to pull from thin air. For example, in a form named `Details` with a title of `Record Details`, Constructor might generate a resource constant called `RecordDetailsFormGroupID2` instead of `DetailsGroupID1`. Also keep in mind that any changes you make to your resources may cause Constructor to create new names for group number constants, which can be quite a headache when debugging.

The following code selects the `MainRockPushButton` if it is not already selected:

```
FormType  *form = FrmGetActiveForm();
UInt8     rockIndex = FrmGetObjectIndex(form,
                                        MainRockPushButton)

if (! (FrmGetControlGroupSelection(form, 1) == rockIndex) )
    FrmSetControlGroupSelection(form, 1, MainRockPushButton);
```

Programming Selector Triggers

Without any extra programming, a selector trigger is merely a button surrounded by a gray (dotted) box; the Palm OS does not provide most of the behavior users expect from a selector trigger, such as displaying a date or time picker dialog box. Your application is responsible for calling up a dialog box to pick a value and then displaying that value in the selector trigger.

Fortunately, implementing a selector trigger is not a difficult process. Tapping the selector trigger queues a standard `ctlSelectEvent` that you can respond to in a form event handler. Use the **FrmPopupForm** or **FrmDoDialog** functions to display a dialog box form, from which the user may pick a value. After returning from the dialog box, change the selector trigger's label with the **CtlSetLabel** function. **CtlSetLabel** takes a pointer to a control and a character pointer containing the string to display in the control's label. The following line of code changes the label on a selector trigger to a date string:

```
ControlType *ctl = GetObjectPtr(MySelectorTrigger);
CtlSetLabel(ctl, "Mon 10/18/99");
```

The **CtlSetLabel** function also works with other controls, not just selector triggers. A companion function, **CtlGetLabel**, works in reverse, retrieving the current value of a control's label and returning that value as pointer to a null-terminated string.

Writing a string with CtlSetLabel that is longer than the string originally contained by the control's resource can cause a crash. Be sure when you create a control to give it a label at least as long as the longest string you plan to assign using CtlSetLabel. Also, the pointer you pass to CtlSetLabel must be valid if the form containing that control receives a `frmUpdate` event. This means that if you change a control's label at run time, the text must come from a global variable, a constant, or a memory chunk that remains locked for the entire life of the control.

The Palm OS contains two built-in dialog boxes for selecting dates and times, which are perfect candidates for use with selector triggers. The date and time pickers are accessible through the **SelectDay** and **SelectTime** functions. Figure 9-4 shows the date and time pickers in action.

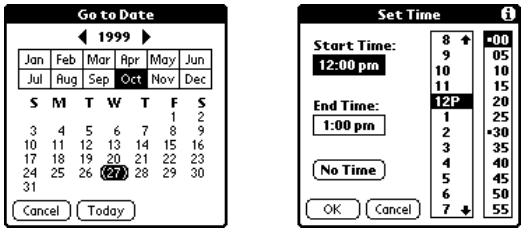


Figure 9-4: The Palm OS built-in date (left) and time (right) picker dialog boxes

The prototype for the **SelectDay** function looks like this:

```
Boolean SelectDay (const SelectDayType selectDayBy,
    Int16 *month, Int16 *day, Int16 *year, const Char *title)
```

The first argument to **SelectDay** allows you to specify the granularity of selection allowed by the date picker. `SelectDayType` is defined in the Palm OS header files, which also define these constant values appropriate for the first argument to **SelectDay**:

- `selectDayByDay`. This option tells the date picker to allow selection of a single day.
- `selectDayByWeek`. **SelectDay** restricts selection in the date picker to picking a week at a time when passed this value.
- `selectDayByMonth`. This option restricts selection to a particular month.

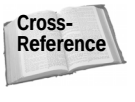
Not only do the `month`, `day`, and `year` arguments specify the initial date displayed by the date picker, but these arguments also specify where the date picker returns the date selected by the user. The `title` argument allows you to customize the title bar of the date picker.

SelectDay returns `true` if the user selected a date from the picker, or `false` if the user tapped the picker's Cancel button.

The following form event handler displays a date picker in response to the user's tapping a selector trigger, and then changes the label in the selector to the date picked by the user. This event handler assumes that the selector trigger is the only control object on the form.

```
static Boolean MyFormHandleEvent (EventType *event)
{
    Boolean       handled = false;
    ControlType   *ctl;
    Char          *label;
    Int16         month, day, year;
    DateTimeType  now;

    if (event->eType == ctlSelectEvent) {
        TimSecondsToDateTime(TimGetSeconds(), &now);
        year = now.year;
        month = now.month;
        day = now.day;

        if (SelectDay(selectDayByDay, &month, &day, &year,
                      "Select a Day")) {
            ctl = GetObjectPtr(MySelectorTrigger);
            label = CtlGetLabel(ctl);
            DateToDOWDMFormat(month, day, year,
                              dfDMYLongWithComma, label);
            CtlSetLabel(ctl, label);
        }
        handled = true;
    }

    return (handled);
}
```

Before displaying the date picker with **SelectDay**, the preceding example initializes the date first displayed in the dialog box to today's date with the **TimSecondsToDateTime** and **TimGetSeconds** functions. Another handy Palm OS date and time function, **DateToDOWDMFormat**, converts the raw month, day, and year values from **SelectDay** into a nicely formatted date string.



Chapter 10, "Programming System Elements," covers these and other date and time functions, along with the `DateTimeType` structure they use, in more detail.

The previous example also initializes the `label` variable, a character pointer to the string displayed in the selector trigger's label, with **CtlGetLabel** to ensure that `label` has enough space to hold the date string produced by **DateToDOWDMFormat**.

The SelectDay function is available only in Palm OS version 2.0 and later. If your application must run on Palm OS 1.0, use the function SelectDayV10, which allows selection of dates by day only, not by week or month.

Using **SelectTime** to present the user with a time picker dialog box is similar to using **SelectDay**. The prototype for **SelectTime** looks like this:

```
Boolean SelectTime (TimeType *startTimeP, TimeType *EndTimeP,
    Boolean untimed, const Char *titleP, Int16 startOfDay,
    Int16 endOfDay, Int16 startOfDisplay)
```

The first two arguments to **SelectTime**, `startTimeP` and `EndTimeP`, are pointers to `TimeType` structures. `TimeType` is defined in the standard Palm OS include file `DateTime.h` as follows:

```
typedef struct {
    UInt8  hours;
    UInt8  minutes;
} TimeType;
```

SelectTime initializes the time picker to display the start and end times given in `startTimeP` and `EndTimeP`. The **SelectTime** function also returns the times the user selects in these two variables. You may also initialize the dialog box to display no selected time by passing `true` for the value of the `untimed` parameter. If the user chooses no time in the picker dialog box, **SelectTime** sets `startTimeP` and `EndTimeP` to the constant `noTime`, which has a value of `-1`.

The `titleP` argument to **SelectTime** contains the text displayed in the time picker's title bar, and `startOfDay` specifies what hour of the day the picker displays at the top of its hour list. The `startOfDay` argument must be a value from `0` to `12`, inclusive.

The `endOfDay` argument specifies what hour of the day will be returned if the user taps the All Day button in the time selector dialog box, and `startOfDisplay` defines what hour will be displayed at the top of the dialog box when it first opens.

Prior to version 3.5 of the Palm OS, the SelectTime function lacks the `endOfDay` and `startOfDisplay` arguments. If you must maintain backwards compatibility with older code, you can use the SelectTimeV33 function instead of SelectTime.

SelectTime returns `false` if the user tapped the dialog box's Cancel button; otherwise, **SelectTime** returns `true` to indicate that the user changed the time in the picker.

If you wish to prompt the user for a single time, instead of start and end times, use the **SelectOneTime** function instead of **SelectTime**. **SelectOneTime** displays a different picker dialog box, shown in Figure 9-5.

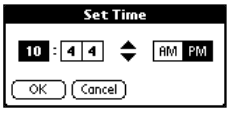


Figure 9-5: The SelectOneTime picker dialog box

The **SelectOneTime** function takes only three arguments, as shown in the following example:

```
Boolean SelectOneTime (Int16 *hour, Int16 *minute,
    const Char *title)
```

The first two arguments to **SelectOneTime** determine the time initially displayed in the picker dialog box, as well as provide variables for **SelectOneTime** to return the time selected by the user. The `title` argument provides the text displayed in the picker's title bar. **SelectOneTime** returns `true` if the user changed the time in the dialog box, or `false` if the user tapped the Cancel button.

Programming Fields

To the user, text fields seem like very simple user interface objects. However, making a text field simple for the user requires a fair amount of effort on your part. Specifically, fields are rather particular about memory management, because the addition or removal of text by the user causes a field object to dynamically change the amount of memory required to store the field's contents.

The Palm OS deals with the difficulty of resizable field text by keeping track of the field's storage with a handle instead of a pointer. Each field has its own handle to store the field's text contents. The system automatically allocates a new handle for a field if the user enters any text into the field.

Setting a Handle for a Text Field

To programmatically set the text in a field, you simply point the field at a different handle that contains the new text for the field. The **FldSetTextHandle** function allows you to change the handle that stores the text for a particular field. In the following example, the **FldSetTextHandle** function sets the handle for a text field, given a pointer to the field and the handle containing the field's new contents.

```
static void SetFieldHandle (FieldType *field, MemHandle textH)
{
    MemHandle  oldTextH;

    // Retrieve the field's old text handle.
    oldTextH = FldGetTextHandle(field);
```

```
    // Set the new text handle for the field.
    FldSetTextHandle(field, textH);
    FldDrawField(field);

    // Free the old text handle to prevent a memory leak.
    if (oldTextH)
        MemHandleFree(oldTextH);
}
```

Notice that the preceding code retrieves the field's original handle with the **FldGetTextHandle** function, and then later explicitly frees the handle with **MemHandleFree**. The Palm OS does not free the handle containing the field's original contents, so the application code must take care of this housekeeping task to prevent a substantial memory leak.

The previous example also calls **FldDrawField** after setting the field's new handle to ensure that the system draws the new contents of the field to the screen. **FldSetTextHandle** updates only the handle; your application must manually redraw the field to display the field's new contents.

When a form closes, the system automatically frees the handle associated with each of its fields. To keep the handle, you must remove the association between the field and the handle. To do this, call **FldSetTextHandle** with `NULL` as its second argument, as in the following example:

```
FldSetTextHandle(field, NULL);
```

Specifying `NULL` for a text field's handle disconnects the field from its handle, allowing the handle to persist after the system has disposed of the field.

Modifying a Text Field

The Palm OS keeps track of a good deal of data about each field, such as the position of the insertion point and where line breaks occur in the text. Directly modifying the text in a handle associated with a field can confuse the system, resulting in garbled text on the screen. To avoid this kind of mess, you must disconnect the handle from the field, modify the text in the handle, and then reattach the handle. The **ReverseField** function in the following example reverses a string contained in a field:

```
static void ReverseField (FieldType *field)
{
    MemHandle  textH;

    if (FldGetTextLength(field) > 0) {
        textH = FldGetTextHandle(field);
        if (textH) {
            char  *str, *p, *q;
            char  temp;
            int   n;
```

```
                FldSetTextHandle(field, NULL);
                str = MemHandleLock(textH);
                n = StrLen(str);
                q = (n > 0) ? str + n - 1 : str;
                for (p = str; p < q; ++p, --q) {
                    temp = *p;
                    *p = *q;
                    *q = temp;
                }

                MemHandleUnlock(textH);
                FldSetTextHandle(field, textH);
                FldDrawField(field);
            }
        }
    }
```

The **ReverseField** function copies the field's text handle into the variable `textH`, and then disconnects the handle from the field with **FldSetTextHandle**. Then **Reverse Field** calls **MemHandleLock** on the handle to lock it and obtain a character pointer to its contents, which is suitable for a little pointer arithmetic to reverse the order of the string. **MemHandleUnlock** unlocks the handle after the text has been reversed, **FldSetTextHandle** reattaches the modified handle to the field, and then **FldDraw Field** redraws the field so the results of the string reversal are visible to the user.

If you need to modify only a bit of a field's text at a time, instead of the extensive changes performed by the preceding example, note that the Palm OS provides a few functions that safely change the field's contents without your having to mess with **FldSetTextHandle**. The functions, and their uses, are described as follows:

- **FldSetSelection.** Sets the start and end of the highlighted selection in a field. **FldSetSelection** takes three arguments: a pointer to a field, the offset in bytes of the start of the text selection, and the offset of the end of the selection. If the start and end offsets are equal, **FldSetSelection** does not highlight any text, instead moving the insertion point to the indicated offset.
- **FldInsert.** Replaces the current selection in a field, if any, with a new string. This function takes three arguments: a pointer to a field, a character pointer containing the string to insert, and the length in bytes of the string to be inserted, not including a trailing null character.
- **FldDelete.** Deletes a specified range of text from a field. **FldDelete** takes three arguments: a field pointer, the byte offset at the beginning of the text that should be deleted, and the byte offset of the end of the text to delete. This function does not delete the character at the ending offset, just everything up to that point in the field.

You should use these functions to make only small, relatively infrequent changes to a field. Both FldInsert and FldDelete redraw the entire field, and if you call them too often, distracting flickering may occur on the screen as the system rapidly and repeatedly redraws the field. Worse yet, both functions post a `fldChangedEvent` to the queue each time you call them. It is possible to overflow the queue with these events if you call FldInsert and FldDelete too often.

Retrieving Text from a Field

You can retrieve the text from a field by using **FldGetTextHandle** to get the field's text handle, and then locking that handle with **MemHandleLock** to obtain a pointer to the string held in that handle; the last **ReverseField** example in the previous section does this. However, using **FldGetTextHandle** in this fashion can be a bit cumbersome if you want only to read the text from a field without modifying it. The Palm OS also provides the **FldGetTextPtr** function for this purpose. **FldGetTextPtr** returns a locked pointer to a field's text string, or `NULL` if the field is empty.

The pointer returned by FldGetTextPtr becomes invalid as soon as the user edits the text field. Also, any changes you make to the contents of the pointer can muddle the field's internal data for keeping track of text length and word wrapping information, taking the field out of sync with its actual contents. Use FldGetTextPtr to take a snapshot of a field's value at only one point in time.

Also, **FldGetTextHandle** and **FldGetTextPtr** return `NULL` if the field has never contained any text, but if the user entered text into the field and deleted all of it, these two functions return a non-`NULL` pointer that points to a `NULL` character (`\0`). You can use the **FldGetTextLength** function to determine how long the text in a field is.

UInt16 length = FldGetTextLength(field);

Setting Field Focus

In a form containing multiple text fields, only one field displays an insertion point and allows text entry at a time. This field has the *focus*. When the user taps in a field, the system automatically shifts the focus to that field. To set the focus within application code, use the **FrmSetFocus** function. **FrmSetFocus** has two arguments: a pointer to a form, and the index number of the field that should receive the focus.

Notice that FrmSetFocus uses the index number of a field, not an object pointer or a resource ID. Recall that you may use FrmGetObjectIndex to retrieve an object's index number, given its resource ID.

You should set which field in a form initially has the focus when you handle the `frmOpenEvent`, right after drawing the form with **FrmDrawForm**. Setting a default field in this way ensures that the user may begin data entry immediately, without having to tap on a field to "wake" it.

The **FrmGetFocus** function is a useful companion to **FrmSetFocus**. **FrmGetFocus** returns the index number of the field that currently has the focus, or the constant `noFocus` if no field has the focus.

Any form with multiple text fields should also support the `prevFieldChr` and `nextFieldChr` Graffiti characters, which the user may enter to quickly switch between fields instead of having to move the stylus from the Graffiti entry area to tap on another field. You can deal with these two characters in a form event handler by handling the `keyDownEvent`. The following event handler moves the focus between two fields in a form in response to the `prevFieldChr` and `nextFieldChr` characters:

```
static Boolean MyFormHandleEvent (EventType *event)
{
    Boolean  handled = false;

    if (event->eType == keyDownEvent) {
        FormType *form = FrmGetActiveForm();
        UInt16 fieldIndex1 = FrmGetObjectIndex(form, Field1ID);
        UInt16 fieldIndex2 = FrmGetObjectIndex(form, Field2ID);

        switch (event->data.keyDown.chr) {
            case nextFieldChr:
                if (FrmGetFocus(form) == fieldIndex1)
                    FrmSetFocus(form, fieldIndex2);
                handled = true;
                break;

            case prevFieldChr:
                if (FrmGetFocus(form) == fieldIndex2)
                    FrmSetFocus(form, fieldIndex1);
                handled = true;
                break;
        }
    }

    return (handled);
}
```

Setting Field Attributes

Internally, a field stores its attributes in a `FieldAttrType` structure, which the Palm OS header file `Field.h` defines as follows:

```
typedef struct {
    UInt16 usable         :1;
    UInt16 visible        :1;
    UInt16 editable       :1;
```

```
    UInt16 singleLine     :1;
    UInt16 hasFocus       :1;
    UInt16 dynamicSize    :1;
    UInt16 insPtVisible   :1;
    UInt16 dirty          :1;
    UInt16 underlined     :2;
    UInt16 justification  :2;
    UInt16 autoShift      :1;
    UInt16 hasScrollBar   :1;
    UInt16 numeric        :1;
} FieldAttrType;
```

Most of the time, you will never need to change these attributes at run time. For those rare occasions where you want to hand-tweak a field's attribute, the Palm OS offers the **FldGetAttributes** and **FldSetAttributes** functions. The values of many of these attributes may also be affected by using other field functions. For example, **FldSetFocus** changes the status of the `hasFocus` field in a field's `FieldAttrType` structure.

FldGetAttributes has two arguments: a pointer to the field, and a pointer to a `FieldAttrType` structure to receive a copy of the field's attributes. **FldSetAttributes** takes the same two arguments, only it copies the `FieldAttrType` structure it receives into the actual attributes of the specified field. Note that setting field attributes has no immediate physical effect on the screen; in order to see the changes, you must redraw the field with **FldDrawField**.

One common use of **FldSetAttributes** is to manually set the `underlined` attribute. A bug in the Windows CodeWarrior R5 version of Constructor causes underlined fields to have solid underlining when displayed in Palm OS version 3.1 and later. A workaround for this bug is to set the `underlined` attribute of each field to the constant `grayUnderline`. The following code does just that:

```
FieldAttrType  attr;
FieldType      *field = GetObjectPtr(MyTextField);

FldGetAttributes(field, &attr);
attr.underlined = grayUnderline;
FldSetAttributes(field, &attr);
```

If you call this code while initializing the form containing the offending field, before calling **FrmDrawForm** to draw that form, you do not need to call **FldDrawField** separately to make the `underlined` attribute change stick.

Programming Gadgets

A gadget object allows you to define your own user interface element. The standard objects included in the Palm OS should be sufficient, and you should stick with them if at all possible to maintain a consistent look and feel with other Palm OS

applications. However, if you have need for an object that simply cannot be implemented using the normal Palm OS interface elements, you can make your own object using a gadget.

Your application code must handle all taps on the gadget, as well as drawing the gadget to the screen. How you accomplish these tasks depends entirely on what the gadget's function is and what it looks like. The simple gadget pictured in Figure 9-6 illustrates the mechanics of interacting with a gadget. This particular gadget represents a single square from a tic-tac-toe board. Tapping the gadget toggles it from a blank square to an "X," from an "X" to an "O," and from an "O" back to a blank square again.

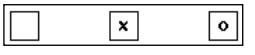


Figure 9-6: Three views of a simple gadget object

The gadget stores its state in a `UInt8` variable. The value `0` represents a blank square, `1` indicates an "X," and `2` is an "O." To provide storage space for this variable, the application must allocate a variable for it when it initializes the form containing the gadget:

```
static void MainFormInit(FormType *form)
{
    UInt8   *data;

    data = MemPtrNew(sizeof(UInt8));
    if (data) {
        *data = 0;
        FrmSetGadgetData(form, FrmGetObjectIndex(form,
                         MainTicTacToeGadget), data);
    }
}
```

MainFormInit uses the **MemPtrNew** function to obtain memory space in which to store the gadget's data. Once it has a pointer to this new memory, **MainFormInit** initializes the gadget data to `0` (representing a blank square). To associate the newly allocated handle with the gadget, the **MainFormInit** function calls **FrmSetGadgetData**, passing a pointer to the form containing the gadget, the index number of the gadget object, and the pointer to the gadget's data.

An event handler for the **Main** form calls **MainFormInit** while handling the `frmOpenEvent`. The form event handler takes care of other details related to the gadget when it handles `frmCloseEvent` and `penDownEvent`. You will find details about the latter two events later in this section; for now, take a look at the `frmOpenEvent` part of the event handler:

```
static Boolean MainFormHandleEvent(EventType *event)
{
    Boolean   handled = false;
    FormType  *form = FrmGetActiveForm();
```

```
    if (event->eType == frmOpenEvent) {
        MainFormInit(form);
        FrmDrawForm(form);
        TicTacToeDraw(form, MainTicTacToeGadget);
        handled = true;
    }

    else if (event->eType == frmCloseEvent) {
        // Handle the frmCloseEvent.
    }

    else if (event->eType == penDownEvent) {
        // Handle the penDownEvent.
    }

    return (handled);
}
```

After calling **MainFormInit**, the event handler draws the form with **FrmDrawForm**, and then draws the gadget with **TicTacToeDraw**:

```
static void TicTacToeDraw(FormType *form, UInt16 gadgetID)
{
    RectangleType  bounds;
    UInt16  gadgetIndex = FrmGetObjectIndex(form, gadgetID);
    MemHandle  dataH = FrmGetGadgetData(form, gadgetIndex);

    if (dataH) {
        FontID   originalFont = FntSetFont(boldFont);

        UInt8  *data = MemHandleLock(dataH);

        FrmGetObjectBounds(form, gadgetIndex, &bounds);
        // Draw a border around the gadget.
        WinEraseRectangle(&bounds, 0);
        WinDrawRectangleFrame(rectangleFrame, &bounds);

        // Draw the contents of the tic tac toe square.
        switch(*data) {
            case 1:
                WinDrawChars("X", 1, bounds.topLeft.x + 6,
                             bounds.topLeft.y + 4);
                break;
            case 2:
                WinDrawChars("O", 1, bounds.topLeft.x + 6,
                             bounds.topLeft.y + 4);
                break;
            default:
                break;
        }
```

```
            MemHandleUnlock(dataH);
            FntSetFont(originalFont);
        }
    }
```

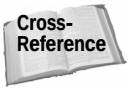
The **TicTacToeDraw** function first determines the current value of the gadget using **FrmGetGadgetData**, which requires a pointer to the form containing the gadget and the index number of the gadget object. **TicTacToeDraw** locks the handle returned by **FrmGetGadgetData** to obtain a pointer, from which the application may read the gadget's stored value.

Once it has the gadget's value, **TicTacToeDraw** can get down to the business of actually drawing the gadget on the screen. **TicTacToeDraw** retrieves the boundaries of the gadget object, which are stored in a `RectangleType` structure. The Palm OS header file `Rect.h` defines `RectangleType`, and the `PointType` structures contained by `RectangleType`, as follows:

```
typedef struct {
    Coord  x;
    Coord  y;
} PointType;

typedef struct {
    PointType  topLeft;
    PointType  extent;
} RectangleType;
```

The `Coord` data type used in `PointType` is simply a `typedef` for `Int16`, defined in the Palm OS header `PalmTypes.h`.

Armed with the gadget's boundaries, **TicTacToeDraw** begins by erasing the gadget's interior with the **WinEraseRectangle** function, and then draws a box around the gadget with **WinDrawRectangleFrame**. Then, depending on the value of the gadget, **TicTacToeDraw** draws an "X" or an "O" in the square with **WinDrawChars**, or leaves the newly erased square alone if the gadget's value indicates a blank square.



WinEraseRectangle, WinDrawRectangleFrame, WinDrawChars, and other drawing functions are detailed later in this chapter, under "Drawing Graphics."

Returning to the **Main** form event handler, the application must free the handle holding the gadget's data when the form no longer has use of it. The `frmCloseEvent` is a perfect place to do just that:

```
else if (event->eType == frmCloseEvent) {
    MemHandle  dataH = FrmGetGadgetData(form,
        FrmGetObjectIndex(form, MainTicTacToeGadget));

    if (dataH)
```

```
        MemHandleFree(dataH);
    handled = true;
}
```

The event handler must also capture the `penDownEvent`, which occurs when the user begins to tap the screen:

```
else if (event->eType == penDownEvent) {
    UInt16  gadgetIndex = FrmGetObjectIndex(form,
                                            MainTicTacToeGadget);
    RectangleType  bounds;

    FrmGetObjectBounds(form, gadgetIndex, &bounds);
    if (RctPtInRectangle(eventP->screenX, eventP->screenY,
                         &bounds)) {
        TicTacToeTap(form, MainTicTacToeGadget, eventP);
        handled = true;
    }
}
```

MainFormHandleEvent compares the screen coordinates of the tap with the boundaries of the tic-tac-toe gadget, and if the tap is within the gadget, the event handler calls the **TicTacToeTap** function to handle updating the gadget:

```
static void TicTacToeTap(FormType *form, UInt16 gadgetID,
                         EventType *event)
{
    UInt16    gadgetIndex = FrmGetObjectIndex(form, gadgetID);
    MemHandle dataH = FrmGetGadgetData(form, gadgetIndex);
    Int16     x, y;
    Boolean   penDown;
    RectangleType  bounds;
    Boolean   wasInBounds = true;

    if (dataH) {
        FrmGetObjectBounds(form, gadgetIndex, &bounds);
        WinInvertRectangle(&bounds, 0);

        do {
            Boolean  nowInBounds;

            PenGetPoint(&x, &y, &penDown);
            nowInBounds = RctPtInRectangle(x, y, &bounds);
            if (nowInBounds != wasInBounds) {
                WinInvertRectangle(&bounds, 0);
                wasInBounds = nowInBounds;
            }
        } while (penDown);

        if (wasInBounds) {
            BytePtr  data = MemHandleLock(dataH);
```

```
                if (++(*data) > 2)
                    *data = 0;
                MemHandleUnlock(dataH);
                TicTacToeDraw(form, gadgetID);
            }
        }
    }
```

The tic-tac-toe gadget reacts to taps in much the same way as a standard button. The gadget highlights when the user taps it (via the **WinInvertRectangle** function) and remains highlighted for as long as the user holds the stylus on the gadget. Moving the stylus outside the bounds of the gadget returns the gadget's appearance to normal, and releasing the stylus outside the gadget's borders has no lasting effect on the gadget.

If the user taps and releases within the gadget, **TicTacToeTap** rotates the gadget's attached data to the next value, and then redraws the gadget by calling **TicTac ToeDraw**.

Programming Lists and Pop-up Lists

List objects, depending on how you use them, can be very simple to implement, or very difficult. The main determining factor in the complexity of a list is whether its elements need to change at run time or not. Defining all of a list's items at design time, as part of the list resource, results in an easy-to-program static list. If you need to generate the list's items while the program is running, though, programming the list requires more effort.

Retrieving List Data

If the user taps the screen and releases within a list, the system queues a `lstSelectEvent`, which you can choose to handle in a form event handler. To determine which list item the user selects, you need to use the **LstGetSelection** function. **LstGet Selection** takes a list pointer as its only argument, returning the number of the list item currently selected. List items are numbered sequentially, starting from `0`. If no item is selected in the list, **LstGetSelection** returns the constant `noListSelection`, which is defined in the Palm OS headers.

Retrieving the text of a particular item in the list requires the **LstGetSelectionText** function. **LstGetSelectionText** is somewhat misnamed; instead of returning the text of the selected list item, the function actually returns *any* list item's text, given a pointer to the list and the item's number. For example, the following code retrieves the text of the first item in a list:

```
Char *text = LstGetSelectionText(list, 0);
```

You can retrieve the currently selected list item's text by calling **LstGetSelection** within the call to **LstGetSelectionText**. The following example retrieves the text of the currently selected list item:

```
Char *text = LstGetSelectionText(list,
                                 LstGetSelection(list));
```

The character pointer returned by LstGetSelectionText is actually a pointer into the list object's internal data, not a copy of that data. Modifying the pointer returned from LstGetSelectionText will actually change text in the list itself, causing unexpected results. For example, the following code directly sets the label of a control from the text of the second item in a list, and then changes the control's label again:

```
CtlSetLabel(ctl, LstGetSelectionText(list, 1));
CtlSetLabel(ctl, "This is a bad idea");
```

This code does have the desired effect of changing the control's label to the text of the appropriate list item. However, the second call to CtlSetLabel overwrites some of the list's contents with the text "This is a bad idea". This spillover happens because the control's label and the list share a pointer. A safer way to set the label from a list item is to copy the list item's text to a new variable, and then set the label from the variable:

```
Char *label = CtlGetLabel(ctl);
StrCopy(label, LstGetSelectionText(list, 1));
CtlSetLabel(ctl, label);
```

You can also retrieve the total number of items in a list or the number of visible items, using the **LstGetNumberOfItems** and **LstGetVisibleItems** functions, respectively.

LstGetVisibleItems is available only on Palm OS version 2.0 and later.

Manipulating Lists

You can directly set which list item is selected with the **LstSetSelection** function. **LstSetSelection** takes two arguments: a pointer to the list object, and the number of the item to select. Like other list functions, `0` represents the first item in the list. You may also pass `-1` as the second argument to **LstSetSelection**, which clears the selection entirely.

To make a particular list item visible, use the **LstMakeItemVisible** function. **LstMake ItemVisible** takes a pointer to a list and the number of the list item to make visible. The **LstMakeItemVisible** function changes the top item of the list to make the requested item visible. If the item is already visible, **LstMakeItemVisible** leaves the list alone.

Note that **LstMakeItemVisible** does not update the list display, only the list's internal data. You need to call **LstDrawList** to refresh the list display after a call to **LstMakeItemVisible**. However, even this is not enough to properly redraw the list, because **LstDrawList** does not always erase the highlight from the selected list item before redrawing the list in the new position mandated by **LstMakeItemVisible**. Calling **LstEraseList** before **LstDrawList** removes the highlight, resulting in a clean redraw of the list. Here is an example that scrolls a list to make its fifth item visible, and then refreshes the list display:

```
LstMakeItemVisible(list, 4);
LstEraseList(list);
LstDrawList(list);
```

If you are initializing a list before the form containing the list has been drawn, you can safely avoid calling **LstEraseList** and **LstDrawList**, because **FrmDrawForm** handles drawing the list for you.

Programming Dynamic Lists

The Palm OS does not provide any functions to insert items into a list or remove them from it. Instead, you have two choices for dynamically altering the contents of a list:

- Set the entire contents of the list at once by passing an array of strings to **LstSetListChoices**.
- Attach a callback function to the list with **LstSetDrawFunction** that the system calls to draw each row of the list.

Both methods of modifying a list are perfectly valid, but using **LstSetDrawFunction** is more flexible and requires less memory overhead. The two methods are described in the next section.

Populating a list with LstSetListChoices

The **LstSetListChoices** function requires a pointer to a list object, a pointer to an array of strings, and the total number of items in the list. Each member of the array of strings becomes a single item in the list. The prototype for **LstSetListChoices** looks like this:

```
void LstSetListChoices (ListType *listP, Char **itemsText,
                        UInt16 numItems);
```

The arguments to **LstSetListChoices** are `list`, a pointer to a list object, `itemsText`, a pointer to an array of strings, and `numItems`, the total number of items in the list. If you have ever dealt with pointers to arrays of strings in C/C++, you should immediately realize that trying to pass all of a list's text at once with **LstSetListChoices** is

cumbersome at best. Your application must handle the allocation, initialization, and maintenance of the array of strings without any help from the operating system, a process which can quickly become a nightmare to code and debug if you need to change the list's contents often during program execution. Also, if the list is long enough, this technique can be a drain on the limited amount of dynamic memory available, because you must keep the entire array of strings in memory while the form containing the list is open.

Filling a list with LstSetDrawFunction

Fortunately, the Palm OS offers a simpler solution to the problem of changing a list's contents at run time. Instead of explicitly setting the text of all the list's items at once, you can set a callback function to take care of drawing each individual row in the list using **LstSetDrawFunction**. Once the callback function is in place, whenever the system needs to draw a row in the list, it calls your custom routine to perform the drawing operation. The one drawback to this approach is that you cannot use the **LstGetSelectionText** function with the list, because none of the list's items are populated with text.

For the sake of simplicity, the example that follows uses this globally defined array of strings:

```
char ListElements[6][10] =
{
    { "Gold" },
    { "Silver" },
    { "Hydrogen" },
    { "Oxygen" },
    { "Argon" },
    { "Plutonium" }
};
```

The event handler for the form containing the list sets a custom drawing function for the list with **LstSetDrawFunction** while handling the `frmOpenEvent`. Using a callback list drawing function still requires a call to **LstSetListChoices** to tell the list how many items it should contain. **MainFormHandleEvent** passes `NULL` as the second argument to **LstSetListChoices** instead of a pointer to an array of strings, along with the total number of items in the list as the third argument to **LstSetListChoices**.

```
static Boolean MainFormHandleEvent(EventType *event)
{
    Boolean   handled = false;

    if (event->eType == frmOpenEvent) {
        FormType  *form = FrmGetActiveForm();
        ListType  *list = GetObjectPtr(MainList);
        int        numChoices = 0;

        // Set custom list drawing callback function.
        LstSetDrawFunction(list, MainListDraw);
```

```
        // Determine the number of items in the string array,
        // then fill the list with that many items.
        numChoices = sizeof(ListAnimals) /
                     sizeof(ListAnimals[0]);
        LstSetListChoices(list, NULL, numChoices);

        // Draw the form, which also draws the list.
        FrmDrawForm(form);
        handled = true;
    }

    return (handled);
}
```

MainListDraw is the callback function that handles drawing each item in the list:

```
static void MainListDraw(UInt16 itemNum,
                         RectangleType *bounds,
                         Char **itemsText)
{
#ifdef __GNUC__
    CALLBACK_PROLOGUE;
#endif

    WinDrawChars(ListElements[itemNum],
                 StrLen(ListElements[itemNum],
                 bounds->topLeft.x,
                 bounds->topLeft.y);

#ifdef __GNUC__
    CALLBACK_EPILOGUE;
#endif
}
```

The system passes the number of the item in the list to draw, along with a rectangle defining the screen area occupied by that item. With these two pieces of information, a simple call to **WinDrawChars** can take care of drawing the appropriate list item, retrieving the correct string by using `itemNum` as an index into the `ListElements` array. For example, when drawing the third list item, the system passes the value `2` in the `itemNum` argument, which causes **MainListDraw** to draw the string stored in `ListElements[2]`, or "Hydrogen". In a full-fledged application, you can use `itemNum` as an index into a stored array of string resources, the application's database, or any other appropriate data structure.

This example draws only the text of each list item, but it is simple to add code in the callback for drawing bitmaps if you wish to further customize the list display. The ability to add extra drawing code to the callback function makes this method of filling a list much more flexible than calling **LstSetListChoices** by itself.

Handling Pop-up Lists

When the user selects an item from a pop-up list, a pop-up trigger sends two events to the queue, a `ctlSelectEvent` and a `popSelectEvent`. The `popSelectEvent` is more useful, because it contains data about both the trigger and its attached list, whereas the `ctlSelectEvent` sparked by tapping a pop-up trigger contains only information about the trigger itself. A `popSelectEvent` also contains other useful bits of information, as outlined in Table 9-2.

Table 9-2
Data in a popSelectEvent

Data	*Description*
`data.popSelect.controlID`	Resource ID of the pop-up trigger
`data.popSelect.controlP`	`ControlPtr` to the pop-up trigger resource control structure
`data.popSelect.listID`	ResourceID of the list attached to a pop-up trigger
`data.popSelect.listP`	`ListPtr` to the attached list structure
`data.popSelect.selection`	Item number of the newly selected list item
`data.popSelect.priorSelection`	Item number of the list item selected before the new selection

When the user selects an item from the pop-up list, the text of that item becomes the new label for the pop-up trigger. However, this behavior works only if you leave the `popSelectEvent` in the queue so **FrmHandleEvent** has a chance to process it. You can leave the event on the queue by not setting `handled = true` in the code that reacts to the `popSelectEvent`. For example, the following form event handler reacts to the `popSelectEvent` but leaves it on the queue so that **FrmHandleEvent** will be able to automatically set the pop-up trigger's label to the text of the item selected from the pop-up list:

```
static Boolean MainFormHandleEvent(EventType *event)
{
    Boolean  handled = false;

    if (event->eType == popSelectEvent) {
        // Do something in response to pop-up list selection.
        // Leave handled = false so the event stays in the
        // queue.
    }
}
```

If a pop-up trigger is attached to a list that the application fills dynamically at run time, you need to initialize the trigger's label to an appropriate value before displaying the form that contains that pop-up trigger. The following code uses the **CtlSetLabel** function to set a pop-up trigger's label to the text of the selected item in its attached list:

```
ListType    *list = GetObjectPtr(PopupList);
ControlType *popTrig = GetObjectPtr(PopupTrigger);
Char        *label = CtlGetLabel(popTrig);

StrCopy(label, LstGetSelectionText(list,
                                   LstGetSelection(list)));
CtlSetLabel(popTrig, label);
```

Note that the code listed above would only work with either a static list populated at design time, or in a list populated using **LstSetListChoices**, because **LstGet SelectionText** can only retrieve text that actually exists within the list itself. If you were to use the **LstSetDrawFunction** method described earlier, you would need to look up the text values yourself, as in the following example:

```
StrCopy(label, ListElements[LstGetSelection(list)]);
CtlSetLabel(popTrig, label);
```

Programming Menus

Dealing with menus in code is mostly straightforward; the "Hello World" application in Chapter 4 covers the basics of handling menu events. Briefly, here are some things to remember when implementing menus in an application:

- In the event handler for the form containing a menu, check for an event of type `menuEvent`.
- Look in the event data structure, under `event->data.menu.itemID`, for the resource ID of the menu item the user selected.
- If a form contains more than two or three menu items, it is a good idea to write a separate menu handling function that actually executes the menu commands, and call that function from your event handler. The menu handling function should take the resource ID of the selected menu as an argument. This technique prevents clutter in the form event handler, making it easier to read.

This information should get you through most menu programming. The rest of this section deals with one potential pitfall of using menu Graffiti shortcuts, along with some methods for doing fancier things with menus in your application.

Using MenuEraseStatus

When the user invokes a menu command using a Graffiti command shortcut, the system briefly displays the name of that menu command in the lower-left corner of the screen, or on Palm OS 3.5, displays a command bar that covers the entire bottom of the screen. Normally, this status message or command bar disappears after a few seconds, just long enough to alert the user that the system has responded to the user's command stroke.

Right before the Palm OS first prints the status message, the system saves the screen area behind the message. After the system's internal timer counts out a few seconds, the system restores that saved screen region. Normally, this behavior is perfectly acceptable. Unfortunately, if your code changes the contents of the lower part of the screen in response to a menu command, the system will still overwrite the status message or command bar area with the bits saved from the old screen, probably resulting in an ugly mess in the bottom of the screen.

Switching between forms in response to a menu command is a common way to cause this error. Since version 2.0, the Palm OS usually erases the menu status automatically when switching forms, but another problem can occur. If the system saves the bits from the old form before displaying a new form, it might save the status message as part of that screen. When returning to the original form, the system then writes those old bits onto the screen, status message and all.

To prevent these errors from happening, the Palm OS provides the **MenuEraseStatus** function, which you may call to immediately erase the status message or command bar from the bottom of the screen. Call **MenuEraseStatus** immediately before performing any action that changes the lower part of the screen. The following code erases the status message or command bar, and then switches to a new form:

```
MenuEraseStatus(NULL);
FrmGotoForm(AnotherForm);
```

MenuEraseStatus takes a single argument, a pointer to a menu bar structure. Normally, you can just pass the value `NULL`, which tells **MenuEraseStatus** to use the current menu.

Removing Menu Items

The Palm OS does not offer any way to gray out unusable menu items, as is often the case with desktop GUIs. For the most part, if a menu command is invalid in a certain context in your application, you can simply display an alert dialog box letting the user know why that command cannot be used. The built-in applications use this strategy for a number of commands. Figure 9-7 shows the dialog box that appears when the user tries to invoke Record➪Delete Item in the To Do application without first selecting a to do item from the list.

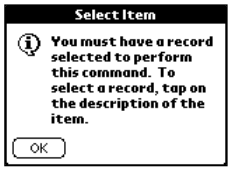


Figure 9-7: The built-in To Do application displays this dialog box when the user tries to use the Record➪Delete Item menu command without selecting an item.

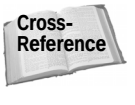
This strategy of displaying a dialog box to inform the user why the command does not work is fine for most context-dependent menu items. However, you may wish to hide a menu item altogether. For example, if an application supports beaming, but can also run on devices that do not have an infrared port, removing the beaming-related menu item entirely on non-IR devices removes clutter from the interface. Hiding completely unavailable commands is also more polite, because it doesn't tempt the user with menu options that display only a disappointing dialog box when invoked.

Because adding and removing menu items are not standard functions in the Palm OS, you must use a brute force approach to hide menu items. Simply create two menu bars, one that contains the menu item you wish to hide and one without that menu item. You may then programmatically switch with the **FrmSetMenu** function which menu is displayed. **FrmSetMenu** takes two arguments: a pointer to a form, and the resource ID of a menu bar to associate with that form.

Palm OS version 1.0 does not support the FrmSetMenu function. If your application needs to run on a 1.0 device, you will not be able to dynamically set menus at run time.

To implement this dual menu bar strategy, assign one of the two menu bars to the form resource at design time, just as you would do for a single menu bar. Then, just before displaying the form containing the switchable menu bar, you can call **FrmSetMenu**, if necessary, to switch to the second menu. The Librarian sample application uses this strategy to hide the Record➪Beam Book command in its Edit view. The following code checks to see if the device running Librarian supports beaming, and if not, sets the menu bar associated with the Edit view to a version that does not contain a Beam Book command:

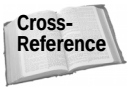
```
UInt32  romVersion;

FtrGet(sysFtrCreator, sysFtrNumROMVersion, &romVersion);
if (romVersion < gRequiredVersion)
    FrmSetMenu(FrmGetActiveForm(), RecordNoIRMenuBar);
```



Using the FtrGet function to retrieve information about supported features is covered in Chapter 10, "Programming System Elements."

Drawing Graphics and Text

Most of the time, you can rely on the standard user interface elements to handle the display of whatever data your application needs to communicate to the user, but sometimes you might need to draw something different on the screen. The Palm OS offers a number of functions for directly drawing graphics and text.



This section discusses drawing black and white graphics only. Chapter 10, "Programming System Elements," delves into the black magic of Palm OS grayscale programming.

Understanding Windows

Palm OS drawing functions center on the concept of the *window*. A window is a rectangular drawing area, either on screen or in memory. You are already familiar with one type of window: forms. A form is simply a window with added features for handling various user interface elements. Every form is a window, but not every window is a form.

The system keeps track of two special windows. At any given time, there may only be one *draw window* and one *active window*. Usually, the operating system treats the same window as both draw window and active window.

The draw window is where the system renders all output from graphics functions. All coordinates used in the drawing functions are relative to the current draw window. The coordinate `0, 0` refers to the upper-leftmost pixel of the draw window. The system clips output from graphics functions to the edges of the draw window. Normally, the system takes care of setting the draw window automatically; the current active form is usually also the draw window. You can manually set the current draw window with the **WinSetDrawWindow** function. **WinSetDrawWindow** takes a single argument, the handle of the window to set as the new draw window, and returns the handle of the old draw window.

The active window is the only region of the screen that accepts user input. If the user taps outside the active window, the system discards that input. The system automatically sets the active form as the active window. If you need to explicitly set a different window to be active, you can do so with the **WinSetActiveWindow** function, which also sets the draw window to be the same as the active window.

By default, whichever form is currently active is both the active window and the draw window. All of the various graphics functions with a **Win** prefix automatically write to the active form if you take no action to manually set the draw window.

You can create a new window with the **WinCreateWindow** function, whose prototype looks like this:

```
WinHandle WinCreateWindow (RectangleType *bounds,
    FrameType frame, Boolean modal, Boolean focusable,
    UInt16 *error)
```

The `bounds` argument is a pointer to a `RectangleType` structure defining the boundaries of the new window. You can define the type of frame that surrounds the window by setting the `frame` argument, which accepts a `FrameType` value describing the type of frame to draw. The Palm OS header file `Window.h` defines a number of constants for window frames; these constants are described in Table 9-3.

Table 9-3
FrameType Constants

Constant	*Value*	*Description*
`noFrame`	`0`	No frame.
`simpleFrame`	`1`	One pixel wide rectangular frame.
`rectangleFrame`	`1`	Same as `simpleFrame`.
`roundFrame`	`0x0401`	One pixel wide frame with rounded corners. The corners have a diameter of 7 pixels.
`boldRoundFrame`	`0x0702`	Two pixel wide frame with rounded corners. The corners have a diameter of 7 pixels.
`popupFrame`	`0x0205`	One pixel wide frame with rounded corners and a shadow effect. The corners have a diameter of 2 pixels. The Palm OS draws menus using this style of frame.
`dialogFrame`	`0x0302`	Two pixel wide frame with rounded corners. The corners have a diameter of 3 pixels. The system draws modal dialog forms and alerts using this style of frame.
`menuFrame`	`0x0205`	Same as `popupFrame`.

The system draws a window's frame outside the actual borders of the window, so take this into account if you need to later remove the window from the screen. The **WinGetWindowFrameRect** function can help with this, because it retrieves a `RectangleType` defining the rectangular area occupied by a window and its frame.

Calling WinCreateWindow does not draw a window's frame; it merely defines what the frame should look like. You must first set the new window as the draw window with WinSetDrawWindow, and then call the WinDrawWindowFrame function to actually draw the window's frame.

Two `Boolean` arguments follow `frame` in a **WinCreateWindow** call. The `modal` argument, if `true`, indicates that the window should be modal, which is to say it prevents user input outside its bounds. Passing `true` as the `focusable` argument `true` allows the window to become the active window.

Unlike most functions in the Palm OS, which indicate an error of some kind in their return value, **WinCreateWindow** returns possible error values via the `error` argument, which is a pointer to a `UInt16` value representing the error encountered. A value of `0` in the `error` argument indicates that **WinCreateWindow** completed successfully.

WinCreateWindow does not clear the area occupied by a new window. If you create a new window over an existing window, you must manually clear the area occupied by the new window with the **WinEraseWindow** function. It may also be important to restore the screen area under the window, particularly for a pop-up-style window. You can use the **WinSaveBits** function to save a screen area to an off-screen window, and then restore that area with **WinRestoreBits** after removing the pop-up window.

Once you are finished with a window, you may remove it from memory, and from the screen, using the **WinDeleteWindow** function. **WinDeleteWindow** takes two arguments. The first is the handle of the window you wish to delete. The second argument is a `Boolean` value to indicate whether the window should be erased before deleting it. Passing `true` as this second argument erases the area occupied by the window, and its frame, before removing the window from memory.

WinDeleteWindow releases the memory occupied by the window, but it does not reset the address of the window handle. After a call to WinDeleteWindow, the window handle of the deleted window becomes invalid. If you plan to use the same window handle later in your code, setting the window handle to `NULL` is a good idea to prevent crashing the system by reading from an unallocated chunk of memory.

The following example creates and displays a pop-up window in the middle of the screen, surrounded by a rounded bold border:

```
WinHandle      newWindow, originalWindow, savedWindow;
RectangleType  newBounds, savedBounds;
UInt16         err;

// Set the bounds of the new window.
newBounds.topLeft.x = 20;
newBounds.topLeft.y = 20;
```

```
newBounds.extent.x = 120;
newBounds.extent.y = 120;

// Create the window.
newWindow = WinCreateWindow(&newBounds, boldRoundFrame, false,
                            false, &err);

// Save the bits from the form beneath the new window.
WinGetWindowFrameRect(newWindow, &savedBounds);
savedWindow = WinSaveBits(&savedBounds, &err);

// Draw the new window.
originalWindow = WinSetDrawWindow(newWindow);
WinEraseWindow();
WinDrawWindowFrame();
WinSetDrawWindow(originalWindow);
```

This example starts by setting up the `newBounds` rectangle that defines the screen area for the new window. Once the rectangle structure is filled, a call to **WinCreate Window** creates the window in memory and assigns it to the `newWindow` handle. Because this window is not intended for user input, the `modal` and `focusable` arguments to **WinCreateWindow** are both `false`.

At this point in the code, a window exists in memory, but no visible change has happened on the screen. The code continues by saving the screen bits in the area occupied by the form, so they may be restored later. The **WinGetWindowFrame Rect** function ensures that the saved bits include the frame around `newWindow`; recall that the system draws a window's frame around the outside edge of the rectangle that makes up the frame, so the `newBounds` rectangle does not actually include the frame. Armed with a `savedBounds` rectangle containing the window and its frame, a call to **WinSaveBits** stores the original screen contents under `newWindow` in the handle `savedWindow`.

Next, the code sets `newWindow` as the draw window with **WinSetDrawWindow**, saving the current draw window in `originalWindow` so it may be restored later. **WinEraseWindow** clears the screen area occupied by the window, and **WinDraw WindowFrame** draws a border around the frame.

Be careful about calling erase functions in the corners of a window or form with a rounded frame. Rounded frame corners pass within the actual drawing area of a window, so functions such as WinEraseWindow, WinEraseLine, and WinErase Rectangle can overwrite the corners of a window with white space. The example code here calls WinDrawWindowFrame after it calls WinEraseWindow for exactly this reason. If WinEraseWindow were called last, it would clip the corners of the window's frame.

The window is now drawn on the screen. Another call to **WinSetDrawWindow** passes draw window status back to the form underneath the new window.

The following code erases the pop-up window drawn by the previous example:

```
WinDeleteWindow(newWindow, true);
WinRestoreBits(savedWindow, savedBounds.topLeft.x,
               savedBounds.topLeft.y);
newWindow = NULL;
```

WinDeleteWindow releases the window structure from memory, and the `true` value in its second argument tells the system to erase the window from the screen, frame and all. **WinRestoreBits** restores the contents of the screen to what they were before displaying the pop-up window, given the `savedWindow` handle, which was filled earlier by a call to **WinSaveBits**. As a precaution, the example also sets the `newWindow` handle to `NULL`, because it is now invalid, and accessing that handle could crash the system.

Drawing Lines

The most basic drawing function in the Palm OS is **WinDrawLine**, which simply draws black lines, one pixel wide, between two points on the screen. **WinDrawLine** takes four arguments. The first two arguments specify the x- and y-coordinates of the line's start point, and the last two arguments are the x- and y-coordinates of the line's end point. For example, the following line of code draws a line connecting the upper-left and lower- right corners of the screen, assuming the draw window occupies the entire display area of a standard Palm OS device:

```
WinDrawLine(0, 0, 159, 159);
```

Two companion functions, **WinDrawGrayLine** and **WinEraseLine**, operate in the same fashion. **WinDrawGrayLine** draws every other pixel of the specified line, resulting in a dotted, or gray, line. **WinEraseLine** draws blank pixels between two points; calling **WinEraseLine** with the same arguments as an earlier **WinDrawLine** call will remove a line from the screen entirely.

The **WinFillLine** function draws a line using the current fill pattern set by the **WinSetPattern** function. See the section on drawing rectangles for more information about using **WinSetPattern**.

Drawing Rectangles

Trying to fill a large area of the screen with the line functions requires many calls to **WinDrawLine** or its ilk, because they only draw a single pixel width at a time. It is much more convenient to call **WinDrawRectangle** to fill a rectangular region of the screen. **WinDrawRectangle** takes a pointer to a `RectangleType` structure and a `cornerDiam` value indicating the roundness of the rectangle's corners.

The `cornerDiam` argument is misnamed and, in some of the Palm OS documentation, described incorrectly. Unlike what its name seems to suggest, the `cornerDiam` argument does not specify the diameter of the corners but rather their radius. The system draws the corners as if an imaginary circle with radius equal to `cornerDiam` had been placed in each corner of the rectangle, tangent to two sides of the rectangle.

A larger `cornerDiam` argument results in a rounder corner. In fact, if `cornerDiam` is equal to half the width or height of a perfectly square rectangle structure, **WinDraw Rectangle** draws a solid circle with diameter equal to twice the `cornerDiam` argument. A `cornerDiam` value of `0` results in a rectangle with square corners. The following code draws a circle 80 pixels across in the middle of the screen:

```
RectangleType  rect;
UInt16  cornerDiam = 40;

rect.topLeft.x = 40;
rect.topLeft.y = 40;
rect.extent.x = 80;
rect.extent.y = 80;

WinDrawRectangle(&rect, cornerDiam);
```

The **WinEraseRectangle** is the opposite of the **WinDrawRectangle** function, requiring the same arguments that **WinDrawRectangle** requires, but erasing pixels instead of filling them.

Filling with patterns

Along with creating solid rectangles, the Palm OS also provides the **WinFillRectangle** function to create rectangles filled with a custom pattern of pixels. **WinFillRectangle** takes the same arguments as the other rectangle creation functions, but it draws a rectangle using a pattern defined by the **WinSetPattern** function. **WinSetPattern** also sets the pattern used by the **WinFillLine** function.

WinSetPattern takes an argument of type `CustomPatternType`, which the Palm OS header file `Window.h` defines as follows:

```
typedef UInt8 CustomPatternType [8];
```

This array of eight `UInt8` variables may not look like much, but it stores a bitmap pattern eight pixels high by eight pixels wide. Each byte in the array defines two rows in an 8×8 grid of pixels, one row per byte. For example, the following `CustomPatternType` array represents a 50 percent gray pattern:

```
CustomPatternType  gray50 = { 0xAA, 0x55, 0xAA, 0x55,
                              0xAA, 0x55, 0xAA, 0x55 };
```

In the preceding example, rows alternate between `AA`, which has a binary representation of `10101010`, and `55` (`01010101` in binary). Each binary `1` represents a pixel that is turned on, and each `0` a pixel that is turned off.

This pattern storage technique is extremely flexible, if a bit difficult to use. Unfortunately, using `CustomPatternType` requires that you perform a fair amount of binary arithmetic each time you want to create a new pattern, and the Palm OS headers do not have any pre-defined constants for commonly used patterns. Listing 9-1 has a few patterns to get you started.

Listing 9-1: **Common Palm OS fill patterns**

```
// Light gray dots (12.5%)
CustomPatternType  patternGray12 = { 0x11, 0x00, 0x44, 0x00,
                                     0x11, 0x00, 0x44, 0x00 };
// Medium gray dots (50%)
CustomPatternType  patternGray50 = { 0xAA, 0x55, 0xAA, 0x55,
                                     0xAA, 0x55, 0xAA, 0x55 };
// Dark gray dots (87.5%)
CustomPatternType  patternGray87 = {~0x11,~0x00,~0x44,~0x00,
                                    ~0x11,~0x00,~0x44,~0x00 };
// Light horizontal stripes (25%)
CustomPatternType  patternStripesHor25 ={ 0xFF,0x00,0x00,0x00,
                                          0xFF,0x00,0x00,0x00};
// Horizontal halftone (50%)
CustomPatternType  patternStripesHor50 ={0xFF,0x00,0xFF,0x00,
                                         0xFF,0x00,0xFF,0x00};
// Dark horizontal stripes (75%)
CustomPatternType  patternStripesHor75 =
    { ~0xFF, ~0x00, ~0x00, ~0x00,
      ~0xFF, ~0x00, ~0x00, ~0x00 };
// Light vertical stripes (25%)
CustomPatternType  patternStripesVer25 = {0x88,0x88,0x88,0x88,
                                          0x88,0x88,0x88,0x88};
// Vertical halftone (50%)
CustomPatternType  patternStripesVer50 = {0xAA,0xAA,0xAA,0xAA,
                                          0xAA,0xAA,0xAA,0xAA};
// Dark vertical stripes (75%)
CustomPatternType  patternStripesVer75 =
    { ~0x88, ~0x88, ~0x88, ~0x88,
      ~0x88, ~0x88, ~0x88, ~0x88 };
// Bold horizontal stripes
CustomPatternType  patternStripesHorBold={0xFF,0xFF,0x00,0x00,
                                          0xFF,0xFF,0x00,0x00};
// Bold vertical stripes
CustomPatternType  patternStripesVerBold={0xCC,0xCC,0xCC,0xCC,
                                          0xCC,0xCC,0xCC,0xCC};
// Diagonal stripes, upper left to lower right
```

```
CustomPatternType  patternStripesDiag1 = {0x88,0x44,0x22,0x11,
                                          0x88,0x44,0x22,0x11};
// Diagonal stripes, upper right to lower left
CustomPatternType  patternStripesDiag2 = {0x11,0x22,0x44,0x88,
                                          0x11,0x22,0x44,0x88};
// Polka dots
CustomPatternType  patternPolkaDots = {0x60, 0xF0, 0xF0, 0x60,
                                       0x06, 0x0F, 0x0F, 0x06};
// Inverse polka dots (white on black)
CustomPatternType  patternWhitePolkaDots =
    { ~0x60, ~0xF0, ~0xF0, ~0x60,
      ~0x06, ~0x0F, ~0x0F, ~0x06 };
// Checkerboard
CustomPatternType  patternCheckerboard = {0xF0,0xF0,0xF0,0xF0,
                                          0x0F,0x0F,0x0F,0x0F};
// Large grid
CustomPatternType  patternGridLarge = {0xFF, 0x80, 0x80, 0x80,
                                       0x80, 0x80, 0x80, 0x80};
// Small grid
CustomPatternType  patternGridSmall = {0xFF,0x88,0x88,0x88,
                                       0xFF,0x88,0x88,0x88};
// Crosshatch
CustomPatternType  patternCrossHatch = {0xFF,0xAA,0xFF,0xAA,
                                        0xFF,0xAA,0xFF,0xAA};
```

Before calling **WinSetPattern** to change the fill pattern, call **WinGetPattern** to retrieve the current pattern so you can restore it when you are done drawing with the new pattern. The following example draws a rectangle in a checkerboard pattern, and then restores the original fill pattern:

```
RectangleType      rect;
CustomPatternType  newPattern = { 0xF0, 0xF0, 0xF0, 0xF0,
                                  0x0F, 0x0F, 0x0F, 0x0F };
CustomPatternType  oldPattern;

rect.topLeft.x = 40;
rect.topLeft.y = 40;
rect.extent.x = 80;
rect.extent.y = 80;

WinGetPattern(oldPattern);
WinSetPattern(newPattern);
WinFillRectangle(&rect, 0);
WinSetPattern(oldPattern);
```

Drawing rectangular borders

The **WinDrawRectangleFrame** and **WinDrawGrayRectangleFrame** functions draw a hollow rectangular border around a given `RectangleType` structure. **WinDraw RectangleFrame** draws a solid frame, and **WinDrawGrayRectangleFrame** draws a gray border, starting with the upper-left pixel turned on and alternating black and white pixels.

Both rectangle frame functions draw the frame outside the edge of the rectangle you pass to the functions. Keep this in mind when you try to determine the screen area occupied by a rectangle frame; you can use the WinGetFramesRectangle function to obtain a `RectangleType` structure that comprises the area occupied by a rectangle and its surrounding frame.

WinDrawRectangleFrame and **WinDrawGrayRectangleFrame** each take two arguments: a `FrameType` value describing the type of border to draw, and a pointer to a `RectangleType` structure defining the rectangle around which the function draws a frame. The rectangle frame functions use the same `FrameType` constants described earlier in Table 9-3.

You can remove a frame from the screen by calling **WinEraseRectangleFrame**, which takes the same parameters as the other two rectangle frame functions, but erases pixels instead of drawing them.

Drawing Text

The **WinDrawChars** function draws a string of characters onto the screen at a specific location. **WinDrawChars** takes four arguments: a pointer to the characters to draw, the length of the characters in bytes, and the x- and y-coordinates where the characters should appear.

To control the font that **WinDrawChars** uses to draw the characters, call the **FntSetFont** function. Only one font can be the active font at a time, and **FntSetFont** changes the active font given the `FontID` of the new font. **FntSetFont** also returns the current active font before changing it, allowing you to store the old font value so you can restore it when you are done with the new font.

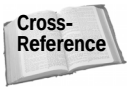


Table 7-1 from Chapter 7, "Building Forms," contains a complete list of constants you can use to specify a `FontID` value.

Another function that alters the behavior of **WinDrawChars** is **WinSetUnderline Mode**. The underline mode may be one of three values:

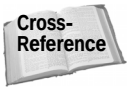
- `noUnderline`. Draws text without any underlining.
- `grayUnderline`. Draws a dotted underline beneath the text, much like the dotted underline used by a standard text field.
- `solidUnderline`. Draws a solid line under the text.

Like **FntSetFont**, **WinSetUnderlineMode** returns the value of the previous underline setting so you can restore it later.

The following example draws the string "To be, or not to be" near the top of the screen in the Palm OS "large" font, with a gray (dotted) underline:

```
FontID             oldFont;
UnderlineModeType  oldUnderline;

oldFont = FntSetFont(largeFont);
oldUnderline = WinSetUnderlineMode(grayUnderline);
WinDrawChars("To be, or not to be",
             StrLen("To be, or not to be"), 0, 20);

// Restore font and underline values
FntSetFont(oldFont);
WinSetUnderlineMode(oldUnderline);
```

Use the **WinEraseChars** function to erase the pixels occupied by characters in a particular string. **WinEraseChars** works particularly well for drawing white text over a black background. The **WinEraseChars** function takes the same arguments as **WinDrawChars**.

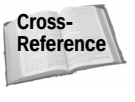


The Palm OS contains a plethora of string and font functions for manipulating text. See Chapter 10, "Programming System Elements," for more details.

Drawing Bitmaps

The **WinDrawBitmap** function takes a pointer to a bitmap resource, along with x- and y- coordinates, and draws the bitmap on the screen at the specified location. Coordinates passed to **WinDrawBitmap** represent the window-relative location of the upper-left corner of the bitmap.

Because bitmaps are defined as resources, you must perform a little resource magic to get a bitmap pointer suitable for passing to **WinDrawBitmap**. The **DmGetResource** function allows you to retrieve a handle to a bitmap resource, which you can then turn into a pointer using **MemHandleLock**.



More information about using resources, including a more complete description of DmGetResource, is available in Chapter 12, "Storing and Retrieving Data."

If you must draw a lot of bitmaps in an application, consider using a function such as **DrawBitmap** in the following example, which displays a bitmap resource given its resource ID and the screen coordinates where it should appear:

```
void DrawBitmap (UInt16 bitmapID, Int16 x, Int16 y)
{
    MemHandle  bitmapH;

    // Retrieve a handle to the bitmap resource.
```

```
    bitmapH = DmGetResource(bitmapRsc, bitmapID);

    if (bitmapH) {
        BitmapType  *bitmap;

        // Lock the bitmap handle to retrieve a pointer, then
        // draw the bitmap.
        bitmap = (BitmapPtr)MemHandleLock(bitmapH);
        WinDrawBitmap(bitmap, x, y);
        MemHandleUnlock(bitmapH);

        // Release the bitmap resource.
        DmReleaseResource(bitmapH);
    }
}
```

Summary

In this chapter, you learned how to program most of the user interface elements in the Palm OS, as well as how to directly draw graphics and text to the screen. After reading this chapter, you should understand the following:

- Alerts are an excellent way to quickly display information, or to prompt the user for simple input.
- You can display forms as stand-alone screens with **FrmGotoForm**, complex dialog boxes with **FrmPopupForm**, or simple dialog boxes with **FrmDoDialog**.
- Many user interface objects post an *enter event* when first tapped, an *exit event* if the user drags the stylus outside the object before lifting it, and a *select event* if the user lifts the stylus within the object's borders.
- Most actions you may perform with a form object require a pointer to that object or the object's index number, which you may retrieve with **FrmGetObjectPtr** and **FrmGetObjectIndex**, respectively.
- You can temporarily remove form objects from the user interface with the **FrmHideObject** function, and you can make them appear again using **FrmShowObject**.
- To set which check box or push button in a control group is selected, or to determine which object is selected, use the **FrmSetControlGroupSelection** and **FrmGetControlGroupSelection** functions.
- Selector triggers should be used to display a dialog box for changing the trigger's value; this dialog box may be a form that you program yourself, or the dialog box may be one of the Palm OS standard date and time pickers, which you may display with **SelectDay**, **SelectTime**, or **SelectOneTime**.

- Fields require attention to detail to program properly, mostly because you must handle much of the memory allocation details of fields with your own code.
- To implement a gadget, you must provide code to handle taps on the gadget and to handle drawing the gadget on the screen.
- You may populate a list at run time either all at once, using **LstSetListChoices**, or a line at a time, by assigning a list callback function with **LstSetDrawFunction**.
- The system takes care of most menu behavior, but you do have to do a little extra work if a menu command changes the screen contents, or if you want to show different menus in different situations.
- All Palm OS drawing occurs within the current *draw window*, which you may set with **WinSetDrawWindow**, and all user input goes to the current *active window*, which you may set with **WinSetActiveWindow**.
- To directly draw on the screen, you may take your pick from a number of drawing functions for creating lines, rectangles, patterns, and text.

✦ ✦ ✦

CHAPTER 10

Programming System Elements

In This Chapter

Checking for supported features

Manipulating text

Handling pen and key events

Implementing alarms

Playing sounds

Generating random numbers

Launching applications

Manipulating time values

Using the clipboard

The preceding chapter, "Programming User Interface Elements," covered the ins and outs of programming the interface of a Palm OS application, the part that the user interacts with directly. This chapter focuses on what runs "under the hood" of your application and the functions that the Palm OS provides to help you with many common programming tasks. The Palm OS supplies a very complete tool box of functions and features to help you with everything from making sounds to manipulating text to launching other applications to checking the time of day.

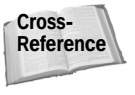
Checking for Supported Features

Many features of the Palm OS do not exist on all versions of the operating system. Infrared beaming, for example, was introduced in Palm OS version 3.0; 1.0 and 2.0 devices do not support beaming. Likewise, different devices running the same version of the Palm OS may not share the same hardware capabilities. For example, the Palm IIIxe and Palm VIIx share the same operating system, but the wireless hardware on the VIIx does not exist in a IIIxe. If your application must be compatible with a variety of Palm OS devices, and you plan to support features that are available on only some of those devices, you need to query the system about what features are available.

The Palm OS feature manager allows you to find out what features exist in the operating system and on the handheld. A *feature* is a 32-bit piece of data published by the operating system, or another program, to indicate the presence of a

particular software or hardware element. The value of a feature has a specific meaning in the context of the application that publishes the feature. You can identify a particular feature by its *feature creator* and *feature number*. The feature creator is a unique creator ID, usually the same as the creator ID of the application that publishes the feature. The feature number is simply a 16-bit value used to distinguish between different features that share a creator ID.

The system stores lists of registered features in *feature tables*. System-published features reside in a feature table in the device's ROM, while a separate feature table in RAM holds application-published features. On Palm OS 3.1 and later, the system copies the contents of the ROM feature table into the RAM feature table at startup, making both system and application features available from the same location.



Your applications can publish their own features for their own use or for use by other programs, which can be very handy if you need quick access to a small amount of data that should persist between closing and re-opening an application. This mechanism also comes in handy when an application must operate without access to global variables, allowing you to store small values without resorting to globals. Chapter 12, "Storing and Retrieving Data," explains how to publish your own features and use feature memory.

To retrieve the value of a feature, use the **FtrGet** function. **FtrGet** has three arguments: the creator ID of the application that owns the feature, the application-specific number assigned to the desired feature, and a pointer to a variable that will receive the feature value. **FtrGet** also returns an error value of `0` if the feature was retrieved without incident, or the constant value `ftrErrNoSuchFtr` if the requested feature does not exist.

Determining Operating System Version

One handy feature published by the system is the version number of the operating system. The Palm OS header file `SystemMgr.h` defines the constants `sysFtrCreator` and `sysFtrNumROMVersion` to assist in retrieving the version number. Calling **FtrGet** with the following code places the system version number value in the variable `romVersion`:

```
UInt32  romVersion;

FtrGet(sysFtrCreator, sysFtrNumROMVersion, &romVersion);
```

The operating system version number published by the system has a very specific, if nonintuitive, format. Here is what the format looks like, expressed in the form of a hexadecimal number:

```
0xMMmfsbbb
```

The individual parts of the format have the following meanings:

- `MM`. Major version number
- `m`. Minor version number
- `f`. Bug fix version number
- `s`. Release stage, wherein the value for `s` may be one of the following:
  - `0`. Development
  - `1`. Alpha
  - `2`. Beta
  - `3`. Release
- `bbb`. Build number for nonreleases

`SystemMgr.h` defines some useful macros for parsing the system version number. Given the feature value returned from a call to **FtrGet**, these macros mask out the appropriate bits of the version number and return a more focused subset of the information contained in the feature. The available macros are:

- **sysGetROMVerMajor**
- **sysGetROMVerMinor**
- **sysGetROMVerFix**
- **sysGetROMVerStage**
- **sysGetROMVerBuild**

For example, the following code retrieves the system version number, parses out the major version, and based on the version number, changes which code is executed:

```
DWord  romVersion;

FtrGet(sysFtrCreator, sysFtrNumROMVersion, &romVersion);
switch (sysGetROMVerMajor(romVersion)) {
    case 1:
        // Version 1.x
        break;
    case 2:
        // Version 2.x
        break;
    case 3:
        // Version 3.x
        break;
    default:
        // Not version 1, 2, or 3
        break;
}
```

Table 10-1 shows the system version numbers of all the Palm OS versions available when this book was written.

Table 10-1
Palm OS System Version Numbers

Number	*Palm OS Version (and Device)*
0x01003001	1.0 (Pilot 1000, Pilot 5000)
0x02003000	2.0 (PalmPilot Personal, PalmPilot Professional)
0x03003000	3.0 (Palm III)
0x03103000	3.1 (Palm IIIx, Palm IIIe, Palm V)
0x03203000	3.2 (Palm VII)
0x03303000	3.3 (Palm Vx; Palm III, Palm IIIx, or Palm V with software upgrade; Visor; original TRGPro)
0x03503001	3.5 (Palm IIIxe, Palm IIIc; newer TRGPro)
0x03513000	3.5.1 (Palm m100)

At the time of this writing, Palm Computing has a prerelease version of Palm OS 3.5 available for developers, with a system version number of 0x03501000. By the time this book goes to press, there is a good chance that a release version of 3.5 will be in circulation, in which case it should have a version number like 0x03503000. Be sure to check the Palm Computing Web site for the latest documentation, though, because it is entirely possible that version numbers may change before release.

Checking Individual Features

Because future versions of the Palm OS may not necessarily include all the features of earlier versions, it is safest to check for specific features before using them, instead of assuming that those features are present on a particular version of the operating system. To help discover what features are available, the Palm OS SystemMgr.h header defines a number of useful constants for checking the presence of individual elements, such as the processor the device uses or whether the device has a backlight.

Unfortunately, the system does not publish features for some fairly obvious things you might wish to check for, such as the presence or absence of an infrared port. For system and device elements that do not have constants defined in SystemMgr.h, checking the system version number is still an available option.

To find the processor used in a Palm OS device, use **FtrGet** to query the system for the `sysFtrNumProcessorID` feature. The processor ID value may be one of the following values for the Palm OS devices that exist at the time of this writing:

- `0x00010000`, or `sysFtrNumProcessor328`. Motorola 68328 DragonBall.
- `0x00020000`, or `sysFtrNumProcessorEZ`. Motorola 68EZ328 DragonBall EZ.

To determine whether the device has a backlight, query the system for the `sysFtrNumBacklight` feature. The backlight feature has a value of `0x00000001` if the device has a backlight, or `0x00000000` if the backlight is not present or not supported.

You may also retrieve a list of features available on the device by repeatedly calling the **FtrGetByIndex** function. **FtrGetByIndex** has the following prototype:

```
Err FtrGetByIndex (UInt16 index, Boolean romTable,
                   UInt32 *creatorP, UInt16 *numP,
                   UInt32 *valueP)
```

The **FtrGetByIndex** function's return value simply indicates whether an error occurred while you were attempting to retrieve the requested feature. **FtrGetByIndex** returns `0` if there was no error, or `ftrErrNoSuchFeature` if the `index` argument provided is out of range.

The `index` argument to **FtrGetByIndex** is simply a numerical index into the feature table, starting at `0` and incrementing by one for each feature in the table. Passing `true` for the `romTable` argument tells **FtrGetByIndex** to return a feature from the ROM feature table; a `false` value for `romTable` retrieves features from the table in RAM.

All three remaining arguments to **FtrGetByIndex** are where the function returns useful information. The `creatorP` argument holds the creator ID of the application that owns the feature, `numP` holds the application-specific feature number, and `valueP` holds the actual value of the feature in question.

The following example loops through the RAM feature table, retrieving each feature registered there:

```
UInt16  index = 0;
UInt32  creator;
UInt16  feature;
UInt32  value;

while (FtrGetByIndex(index, false, &creator, &feature, &value)
       != ftrErrNoSuchFeature) {
    // Do something with the values of creator, feature, and
    // value.
    index++;
}
```

Manipulating Text

The Palm OS provides several useful functions and macros for manipulating text. This section covers three groups of functions and macros: font functions, string functions, and character macros. Font functions deal with the actual on-screen representation of characters, string functions allow you to control strings within your program's code, and character macros are handy tools for determining information about individual characters.

Using Font Functions

Font functions in the Palm OS are dependent on the current font setting in an application. The system keeps track of one font at a time. All font functions, as well as all drawing functions that deal with text (such as **WinDrawChars**), use the current font. To set the current font, use the **FntSetFont** function:

```
FontID  oldFont;

oldFont = FntSetFont(largeFont);
```

The **FntSetFont** function takes a single argument of type `FontID`, specifying an ID for the new font. The function returns whatever font was previously the current font before calling **FntSetFont**, allowing you to store that value for later restoration. It is good coding practice to always save the value of the current font before changing it. Once you are done using the new font setting, restore the original font with another call to **FntSetFont**, like this:

```
FntSetFont(oldFont);
```

The Palm OS also provides the **FntGetFont** function, which takes no arguments and returns the `FontID` value of the current font.

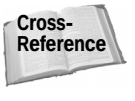


Table 7-1 from Chapter 7, "Building Forms," contains a complete list of constants you can use to specify a `FontID` value.

Many of the Palm OS font functions are purely informational in function, providing data about the current font. These information functions take no arguments. Table 10-2 briefly describes the return value from each of the informational font functions.

Table 10-2
Informational Font Functions

Function	*Return Value*
FntAverageCharWidth	Width in pixels of the average character in the current font.
FntBaseLine	Distance in pixels from the top of the character cell to the baseline for the current font.
FntCharHeight	Height in pixels of a character in the current font, including accents and descenders.
FntDescenderHeight	Distance in pixels from the baseline to the bottom of the character cell for the current font.
FntLineHeight	Height in pixels of a line of text in the current font. The height of a line is equal to the height of a character plus the distance between lines of text.

A few other simple functions, **FntCharWidth**, **FntCharsWidth**, and **FntLineWidth**, provide the width of individual characters or strings of characters. The **FntCharWidth** function takes a single character as an argument and returns its width in pixels for the current font. The **FntCharsWidth** function takes a string and the length of the string in bytes as arguments, and it returns the width of the entire string in pixels, substituting the font's missing character symbol for any character that does not exist in the current font. The **FntLineWidth** function takes the same arguments as **FntCharsWidth**, returning the width in pixels of the string for the current font, taking tab characters and missing characters into account.

The WinDrawChars function does not account for the width of tab characters, and neither does FntCharsWidth. If you want to know the exact width of a string to be drawn by WinDrawChars, use the FntCharsWidth function. The FntLineWidth function works better for determining the width of a string that will appear at the start of a line in a text field, because it properly handles the width of tab characters.

Fitting text to a specific screen width

Some applications require a method for drawing text within a certain amount of space on the screen. The Address List view in the built-in Address Book application is a good example. Each name must fit within a specific area on the screen. If the application called **WinDrawChars** to draw each name in the list, without checking first to see how much of the name would fit, the program would draw over the top of the phone numbers, resulting in an unreadable mess.

Fortunately, there is an easy way around the problem of keeping text within a specific area of the screen. The Palm OS provides the **FntCharsInWidth** function, which allows you to determine the number of bytes of a particular string that will fit within a given width using the current font. The **FntCharsInWidth** function can be tricky to use, however, because half of its arguments serve as both input parameters and return values. The prototype for **FntCharsInWidth** looks like this:

```
void FntCharsInWidth (const char *string, Int16 *stringWidthP,
    Int16 *stringLengthP, Boolean *fitWithinWidth)
```

The `string` argument is simply the string to test. As input parameters, `stringWidthP` tells **FntCharsInWidth** the maximum width in pixels that the string should occupy, and `stringLengthP` represents the maximum length of text to allow in bytes. Upon return, **FntCharsInWidth** sets `stringWidthP` to the actual width in pixels of the text that fits, and the function sets `stringLengthP` to the actual length in bytes of the text. The function sets the `fitWithinWidth` argument to `true` if the entire string fits within the specified width and length; **FntCharsInWidth** sets `fitWithinWidth` to `false` if the string must be truncated to fit the specified width and length.

The **FntCharsInWidth** function treats spaces and newlines at the end of a string specially. The function removes any spaces at the end of the string and ignores them, returning `true` in the `fitWithinWidth` argument. If there is a newline in the string, **FntCharsInWidth** ignores any characters after the newline and treats the string as truncated, returning `false` in the `fitWithinWidth` argument.

As an example, the list view in the Librarian sample application (introduced in Chapter 8, "Building Menus") must deal with similar space restrictions as the Address Book application. The following utility function from Librarian, **DrawCharsInWidth**, draws as much of a string as will fit a given width, appending an ellipsis (. . .) to the end of the string if it must be truncated to fit.

```
static void DrawCharsInWidth (Char *str, Int16 *width,
    Int16 *length, Int16 x, Int16 y, Boolean rightJustify)
{
    Int16    ellipsisWidth;
    Boolean  fitInWidth;
    Int16    newX;
    char     ellipsisChar;


    // Determine whether the string will fit within the maximum
    // width.
    FntCharsInWidth(str, width, length, &fitInWidth);

    // If the string fits within the maximum width, draw it.
    if (fitInWidth) {
        if (rightJustify)
            WinDrawChars(str, *length, x - *width, y);
        else
```

```
            WinDrawChars(str, *length, x, y);
        }

        // The string was truncated; append an ellipsis to the
        // end of the string, and recalculate the portion of the
        // string that can be drawn, because the ellipsis shortens
        // the width available.
        else {
            // Retrieve an ellipsis character and set its width.
            ChrHorizEllipsis(&ellipsisChar);
            ellipsisWidth = (FntCharWidth(ellipsisChar));

            *width -= ellipsisWidth;
            FntCharsInWidth(str, width, length, &fitInWidth);

            if (rightJustify)
                newX = x - *width - ellipsisWidth;
            else
                newX = x;

            WinDrawChars(str, *length, newX, y);
            newX += *width;
            WinDrawChars(&ellipsisChar, 1, newX, y);

            // Add the width of the ellipsis to return the actual
            // width used to draw the string.
            *width += ellipsisWidth;
        }
    }
```

The first three arguments to **DrawCharsInWidth** mirror the first three arguments of the **FntCharsInWidth** function; `str` is the string to draw, `width` is a pointer to the width in pixels that the string must fit into, and `length` is a pointer to the maximum length in bytes for the string. The **DrawCharsInWidth** function has `x` and `y` arguments to specify the window-relative coordinates of the upper-left corner of the space the string should occupy, and the `rightJustify` argument, if `true`, tells the function to draw the text right-justified within the given space. If `rightJustify` is false, **DrawCharsInWidth** simply starts at the `x` position when drawing the string.

After declaring variables, **DrawCharsInWidth** calls **FntCharsInWidth** to determine if `str` will fit within the constraints of `width` and `length`. If the return value from **FntCharsInWidth**, stored in `fitInWidth`, is `true`, the entire string fits within the given width and length, so **DrawCharsInWidth** calls **WinDrawChars** to draw the string to the screen, modifying the starting horizontal coordinate if the string should be drawn right-justified.

If the return value from **FntCharsInWidth** is `false`, the string must be truncated to fit within the allotted space. In this situation, **DrawCharsInWidth** attaches an ellipsis character to the end of the string before drawing. Because the ellipsis itself

takes up some space, it might be necessary to further truncate `str` to allow for the width of the ellipsis character. The **DrawCharsInWidth** function retrieves an ellipsis character with the **ChrHorizEllipsis** macro, determines the character's width with **FntCharWidth**, and subtracts the character's width from the total width available. Then **DrawCharsInWidth** calls **FntCharsInWidth** again with the new value of `width`, modified to accommodate the ellipsis character.

Librarian uses the ChrHorizEllipsis macro instead of hard-coding the value of the ellipsis character, because starting with Palm OS 3.1, the ellipsis has a different character code than in previous versions of the operating system. The ChrHorizEllipsis macro is not available in pre-3.1 header files (such as those that ship with CodeWarrior R5), so if you are building with headers earlier than version 3.1 and your application is not intended for use on 3.1 or later devices, you should use the `horizEllipsisChr` constant instead of ChrHorizEllipsis.

After the second call to **FntCharsInWidth**, `length` points to the number of bytes of `str` that will fit, followed by an ellipsis character, within the constraints originally set by `width` and `length`. Passing the value of `length` as the second argument to **WinDrawChars** draws only those characters in `str` that fit before the ellipsis. After drawing `str`, **DrawCharsInWidth** moves the drawing position, represented by `newX`, to the right of the text and draws an ellipsis character with another call to **WinDrawChars**.

The **DrawCharsInWidth** function treats its `width` and `length` arguments in much the same way as **FntCharsInWidth** treats its `stringWidthP` and `stringLengthP` arguments. Before returning, **DrawCharsInWidth** makes sure that `width` and `length` represent the actual width in pixels and length in bytes of the string that it drew, instead of the values originally passed to the function. For this reason, **DrawCharsInWidth** adds the width of the ellipsis character to `width` if the string was truncated, because the second call to **FntCharsInWidth** sets `width` to the length of the string without the ellipsis.

Using String Functions

Many of the string functions in the Palm OS are familiar to anyone with a reasonable amount of C/C++ programming experience, because the system provides its own versions of string functions from the C standard library. Table 10-3 lists the Palm OS string functions that mirror standard library calls, as well as each function's equivalent in the standard library.

As a general rule, use the Palm OS string functions instead of their standard library equivalents. Including functions from the standard library in your application unnecessarily increases the size of the compiled executable code. You can keep your application much smaller by using the functions that already exist in the operating system.

Table 10-3
Palm OS String Functions and Their Standard Library Equivalents

Palm OS Function	*Standard Library Equivalent*
StrAToI	atoi
StrCat	strcat
StrChr	strchr
StrCompare	strcmp
StrCopy	strcpy
StrLen	strlen
StrNCat	strncat
StrNCompare	strncmp
StrNCopy	strncpy
StrPrintF	sprintf
StrStr	strstr
StrVPrintF	vsprintf

The Palm OS includes **StrIToA** and **StrIToH** functions for converting integer values to strings containing the integer's ASCII or hexadecimal equivalent. The following code shows these two functions in action:

```
int   i = 42;
char  *asciiString, *hexString;

// If necessary, allocate memory for the two strings using
// MemHandleNew and MemHandleLock. This step has been omitted.

StrIToA(asciiString, i);
StrIToH(hexString, i);

// asciiString now contains "42".
// hexString now contains "0000002A".
```

In addition to the **StrCompare** and **StrNCompare** functions for comparing the values of two strings, the Palm OS also offers **StrCaselessCompare** and **StrNCaseless Compare**. The caseless versions ignore the case and accent of each character in the two strings to be compared. For example, **StrCaselessCompare** treats e, E, and é as the same character for purposes of comparison.

Use StrCompare and StrNCompare to compare strings for the purpose of sorting them alphabetically, because these two functions pay attention to case and accent. When comparing strings for the purpose of finding a particular string, use StrCaselessCompare and StrNCaselessCompare. The caseless comparison is particularly useful when trying to find a string based on user input in a text field, because it allows the user to enter a search string more quickly by not bothering with capitalization or accented characters, both of which take more time to enter using Graffiti.

Using Character Macros

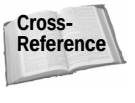
Like many of the string functions that mirror functions from the C standard library, the Palm OS also contains several character attribute macros that mimic the macros defined in the standard library. These macros take a character argument and return a Boolean response indicating whether the character belongs to a specific class of characters. For example, the **IsDigit** macro returns true if a character is one of the numeric digit characters (`0` through `9`).

Use the Palm OS versions of the character attribute macros instead of the C standard library versions to avoid compiling extra library code into your application.

There are two versions of each character macro in the Palm OS. The older macro of each pair has been in the Palm OS since version 1.0, whereas the new version of each macro was added more recently with the *International Feature Set*. The International Feature Set was introduced with Palm OS 3.1 to provide features to support localization of an application to different languages, particularly Asian languages that require double-byte character encoding. To check for the existence of the International Feature Set on a given device, use the following line of code:

```
error = FtrGet(sysFtrCreator, sysFtrNumIntlMgr, &value);
```

If the International Feature Set is present, `value` will be non-zero, and the value for `error` will be zero (to indicate no error). The older macros are not available if you are compiling an application using headers from a Palm OS version that includes the International Feature Set.



The International Feature Set contains much more than the small number of macros included in this chapter. See "Localizing Applications" in Chapter 20, "Odds and Ends," for more details.

Table 10-4 correlates the two Palm OS versions of each character macro with the equivalent macro from the C standard library. The older character macros are defined in the Palm OS header file `CharAttr.h`, and the new macros from the international manager are defined in `TextMgr.h`.

Table 10-4
Character Macros in the Palm OS

Old Palm OS Macro	*Macro from Palm OS with International Feature Set*	*Standard Library Equivalent*
IsAlNum	TxtCharIsAlNum	isalnum
IsAlpha	TxtCharIsAlpha	isalpha
IsAscii	TxtCharIsAscii	isascii
IsCntrl	TxtCharIsCntrl	iscntrl
IsDigit	TxtCharIsDigit	isdigit
IsGraph	TxtCharIsGraph	isgraph
IsHex	TxtCharIsHex	isxdigit
IsLower	TxtCharIsLower	islower
IsPrint	TxtCharIsPrint	isprint
IsPunct	TxtCharIsPunct	ispunct
IsSpace	TxtCharIsSpace	isspace
IsUpper	TxtCharIsUpper	isupper

Calling the old character macros is somewhat different from calling the macros from the International Feature Set or the macros in the C standard library. Except for **IsAscii**, which takes a single character argument, the old character macros require two arguments. The first argument is the character attribute block, followed by the character itself. The Palm OS provides the **GetCharAttr** function to retrieve the character attribute block, so testing a character for a particular attribute involves code similar to the following:

```
Boolean  itsADigit;
char     c = '2';

itsADigit = IsDigit(GetCharAttr(), c);
```

The new macros in the International Feature Set have longer, more cumbersome names, but they take only one argument, the character to be tested, so in practice they are easier to use:

```
itsADigit = TxtCharIsDigit(c);
```

Both of the examples above set `itsADigit` to `true`.

The Palm OS also has the macro **IsDelim** (defined as **TxtCharIsDelim** in the International Feature Set) to test whether a character is a text delimiter. A text delimiter in this case is any space or punctuation character.

Another macro not found in the C standard library, **ChrIsHardKey** (**TxtCharIs HardKey** in the International Feature Set), tests whether a character code represents one of the four hardware application buttons on the device. Testing an incoming character from a `keyDownEvent` with **ChrIsHardKey** allows you to copy the behavior of the built-in applications, which cycle through the categories as the user repeatedly presses, or holds, the button that activates the application. Your own application can take advantage of this behavior if the user redefines one of the buttons to launch your application instead of one of the default programs.

The Librarian sample application implements this category-cycling behavior if the user has defined a hardware button to launch Librarian. Note that, unlike most of the other character attribute macros, the older **ChrIsHardKey** takes only one argument (the character to test), but **TxtCharIsHardKey** requires two arguments (the modifiers to the `keyDownEvent` and the character itself). The following example is the section of Librarian's **ListFormHandleEvent** event handler that handles a `keyDownEvent` if it happens to be the result of the user's pressing an application button. This example assumes a version of the Palm OS that does not include the International Feature Set, such as Palm OS 3.0 or earlier:

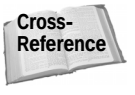
```
case keyDownEvent:
    if (ChrIsHardKey(event->data.keyDown.chr)) {
        if (! (event->data.keyDown.modifiers &
               poweredOnKeyMask)) {
            ListFormNextCategory();
            handled = true;
        }
    }
    else {
        // Other keyDownEvent handling omitted.
    }
    break;
```

Here is the same section of event handler, written for a Palm OS version that includes the International Feature Set, such as Palm OS 3.1:

```
case keyDownEvent:
    if (TxtCharIsHardKey(event->data.keyDown.modifiers,
                         event->data.keyDown.chr)) {
        if (! (event->data.keyDown.modifiers &
               poweredOnKeyMask)) {
            ListFormNextCategory();
            handled = true;
        }
    }
    else {
        // Other keyDownEvent handling omitted.
```

```
        }
        break;
```

The event handler checks that the `keyDownEvent` modifiers do not contain the `poweredOnKeyMask` value, which would indicate that the application button press was responsible for activating the application; the first press of the hardware button that activates the program should not cycle the categories. Satisfied that the incoming `keyDownEvent` represents the second or later hardware button press, **ListFormHandleEvent** calls **ListFormNextCategory**, another function internal to Librarian, to actually change the category.

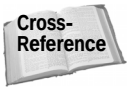


More details about implementing categories are available in Chapter 13, "Manipulating Records."

Handling Pen Events

Most of the time, handling user input is best left in the capable hands of user interface elements. However, if you want your application to directly respond to the stylus as the user drags it across the screen (for example, as in a drawing program), the Palm OS provides three events for direct handling of stylus input. The system generates three events in response to the user tapping, dragging, and releasing the stylus: `penDownEvent`, `penMoveEvent`, and `penUpEvent`.

The `penDownEvent` occurs when the user first taps the screen, and the event contains the window-relative coordinates of that tap in the event's `screenX` and `screenY` members.



See the "Drawing Graphics and Text" section of Chapter 9, "Programming User Interface Elements," for more information about window-relative coordinates.

As the user drags the stylus across the screen, the system queues `penMoveEvent` events. Like the `penDownEvent`, each `penMoveEvent` contains the current window coordinates of the stylus.

When the user finally lifts the stylus from the screen, the system queues a `penUpEvent`, which contains the window-relative coordinates of the place where the stylus left the screen. The `penUpEvent` also contains two `PointType` structures in the event's `data` member, called `start` and `end`. These two point structures contain the display-relative coordinates of the start and end points of the stylus stroke.

If you handle a `penDownEvent` completely, returning `true` in the form event handler, controls and other user interface elements on the form never get a chance to process a stylus tap. If you wish to capture stylus input and allow the user to manipulate user interface objects on the same form, you should check the screen coordinates of the `penDownEvent` and return `false` if the stylus tap occurs on an object.

The following form event handler implements a very simple doodling program. This example handles `penDownEvent`, `penMoveEvent`, and `penUpEvent` to draw on the screen wherever the user drags the stylus, except for small regions at the top and bottom of the screen, which prevents the scribbling from overwriting the application's title bar or any command buttons located at the bottom of the form. The drawing area itself is defined by a global `RectangleType` variable called `gDrawRect`. The **MainFormHandleEvent** function also looks in the global variables `gX` and `gY` for the current coordinates of the stylus, and in `gPenDown` for the current state of the stylus, either `true` for down or `false` for up.

```
static Boolean MainFormHandleEvent(EventType *event)
{
    Boolean  handled = false;

        case penDownEvent:
            if (RctPtInRectangle(event->screenX,
                event->screenY, &gDrawRect)) {
                gX = event->screenX;
                gY = event->screenY;
                gPenDown = true;
                handled = true;
            }
            break;

        case penMoveEvent:
            if (RctPtInRectangle(event->screenX,
                event->screenY, &gDrawRect) && gPenDown) {
                Int16 newX = event->screenX;
                Int16 newY = event->screenY;

                WinDrawLine(gX, gY, newX, newY);
                gX = newX;
                gY = newY;
                handled = true;
            }
            break;

        case penUpEvent:
            if (RctPtInRectangle(event->screenX,
                event->screenY, &gDrawRect) && gPenDown) {
                Int16 newX = event->screenX;
                Int16 newY = event->screenY;

                WinDrawLine(gX, gY, newX, newY);
                gX = gDrawRect.topLeft.x;
                gY = gDrawRect.topLeft.y;
                gPenDown = false;
                handled = true;
```

```
                }
                break;

            default:
                break;

        }

        return handled;
    }
```

When the form receives any of the pen-related events, the event handler first checks to see if the point passed by the event is within the drawing area. The **RctPtInRectangle** function is useful for this operation, because it returns `true` if a given point lies within a given rectangle. By comparing the `screenX` and `screenY` coordinates passed by the various pen events with the borders of the rectangle, the event handler can tell whether it should take care of the event, or hand it off to the system for default processing.

A `penDownEvent` within the drawing area merely sets the global `gX` and `gY` variables to the coordinate of the screen tap. It also sets the `gPenDown` variable to `true`, providing a point from which the `penMoveEvent` and `penUpEvent` handlers may draw a line and indicating to the rest of the application that the stylus is currently touching the screen. The **MainFormHandleEvent** function handles the `penMoveEvent` and `penUpEvent` only if the pen state is currently down (which is to say, `gPenDown` is `true`); otherwise, the program would draw some unexpected things on the screen if the user tapped outside the drawing rectangle and then released the stylus within the drawing area.

Handling Key Events

Whenever the user enters a text character via Graffiti, the system queues a `keyDownEvent`. The system also puts a `keyDownEvent` on the queue when the user presses a hardware button or taps one of the silk-screened buttons, such as the **Menu** button. Along with these tangible key events, the system also generates various virtual key events when certain actions take place, such as when the low battery display appears or when the user extends the antenna on a Palm VII.

The `keyDownEvent` stores the value of the character, hardware button, or virtual key in the event's `chr` member. You can find a large number of useful constants for various key event values in the header file `Chars.h`. Some of the more common character code constants are listed in Table 10-5.

Table 10-5
Character Code Constants for keyDownEvent

Constant	*Value*	*Description*
leftArrowChr	0x1C	Character left Graffiti stroke
rightArrowChr	0x1D	Character right Graffiti stroke
nextFieldChr	0x0103	Graffiti next field character
prevFieldChr	0x010C	Graffiti previous field character
pageUpChr	0x0B	Hardware scroll up button
pageDownChr	0x0C	Hardware scroll down button
hard1Chr	0x0204	Date Book hardware button
hard2Chr	0x0205	Address Book hardware button
hard3Chr	0x0206	To Do List hardware button
hard4Chr	0x0207	Memo Pad hardware button
hardPowerChr	0x0208	Hardware power button
hardCradleChr	0x0209	HotSync button on the cradle
launchChr	0x0108	Application launcher silk-screened button
menuChr	0x0105	Menu silk-screened button
calcChr	0x010B	Calculator silk-screened button
findChr	0x010A	Find silk-screened button
lowBatteryChr	0x0101	Queued when the low battery dialog appears
alarmChr	0x010D	Queued before displaying an alarm
ronamaticChr	0x010E	Queued upon a stroke from the Graffiti area to the upper half of the screen
backlightChr	0x0113	Toggles the state of the backlight
autoOffChr	0x0114	Queued when the power is about to shut off because of inactivity

The `keyDownEvent` also has a `modifiers` member, which is a bit field that stores a number of flags pertaining to the contents of the key event. The `Event.h` header file contains constant values that you can use as bit masks to test for the presence or absence of certain flags in the `modifiers` field. Table 10-6 shows some of these constants and their values.

Table 10-6
Bit Masks for keyDownEvent Modifiers

Constant	*Value*	*Description*
`shiftKeyMask`	`0x0001`	Graffiti is in case-shift mode.
`capsLockMask`	`0x0002`	Graffiti is in caps lock mode.
`commandKeyMask`	`0x0008`	This `keyDownEvent` is a menu event or a special virtual key code.
`autoRepeatKeyMask`	`0x0040`	This `keyDownEvent` is the result of an auto-repeat event. Auto-repeating usually occurs as a result of holding down one of the hardware buttons, such as the scrolling buttons.
`poweredOnKeyMask`	`0x0100`	This `keyDownEvent` powered the system on.

Virtual key events always have the `commandKeyMask` flag set. Checking for the presence of this flag is a good way to separate normal text entry from special system events.

As an example, the following form event handler displays an alert if the user presses the scroll down hardware button (represented by `pageDownChr`), but only if the user holds down the button long enough to generate an auto-repeat, which requires about half a second of pressing.

```
static Boolean MainFormHandleEvent(EventType *event)
{
    Boolean  handled = false;

    switch (event->eType) {
        case keyDownEvent:
            if ((event->data.keyDown.chr == pageDownChr) &&
                (event->data.keyDown.modifiers &
                 autoRepeatKeyMask)) {
                FrmAlert(MyAlert);
                handled = true;
            }

        default:
            break;

    }

    return handled;
}
```

Setting Alarms

The Palm OS provides facilities for setting alarms based on the device's real-time clock, which may be used to display reminders or perform periodic tasks. Alarms will actually wake up the handheld, bringing it out of sleep mode so it can perform some sort of processing, which may or may not include alerting the user with sound or a dialog box. Setting and responding to alarms requires some cooperation between the system and your application. Here are the steps required for setting and responding to an alarm, followed by sections that explain the process in greater detail:

1. The application sets the alarm with **AlmSetAlarm**. The **AlmSetAlarm** function places a new alarm into the Palm OS alarm manager's queue. If multiple applications request the same alarm time, the alarm manager processes alarms on a first-requested, first-serviced basis. The alarm manager keeps track of only a single alarm for each application.
2. When an alarm comes due, the system sends a `sysAppLaunchCmdAlarmTriggered` launch code to each application that has an alarm in the queue for the current time.
3. The application responds to the `sysAppLaunchCmdAlarmTriggered` launch code. This launch code gives applications the chance to perform some sort of quick action, such as setting another alarm, playing a short sound, or performing some quick maintenance activity. Anything the application does at this point should be very brief; otherwise, the application will delay other applications with alarms set at the same time from responding in a timely fashion.
4. Once all the applications with alarms set for the current time have dealt with the `sysAppLaunchCmdAlarmTriggered` launch code, the alarm manager sends a `sysAppLaunchCmdDisplayAlarm` code to each application with an alarm set for the current time.
5. The application responds to the `sysAppLaunchCmdDisplayAlarm` launch code. Now that all applications with pending alarms for the current time have had a chance to do something quick with the `sysAppLaunchCmdAlarmTriggered` launch code, the application may perform some lengthy operation in response to the alarm, such as displaying a dialog box.
6. If applications are still displaying dialog boxes in response to the `sysAppLaunchCmdDisplayAlarm` launch code when another alarm comes due, the alarm manager sends a new `sysAppLaunchCmdAlarmTriggered` launch code to each application with a pending alarm for the new time. However, the system waits until the last application dismisses its dialog box before sending out another batch of `sysAppLaunchCmdDisplayAlarm` codes.

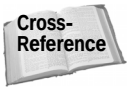
Setting an Alarm

Use the **AlmSetAlarm** function to set an alarm. The prototype for **AlmSetAlarm** looks like this:

```
Err AlmSetAlarm (UInt16 cardNo, LocalID dbID, UInt32 ref,
                 UInt32 alarmSeconds, Boolean quiet)
```

The `cardNo` argument specifies the number of the storage card on which the application resides, and `dbID` is the local ID of the application. These two items may be obtained using the **DmGetNextDatabaseByTypeCreator** function. See the next example for more details.

Because an application does not have access to its global variables when responding to the two alarm launch codes, the `ref` argument provides a space for you to store information that the application might require when displaying the alarm. If you find that you need to pass more information than you can comfortably fit in the 32-bit integer provided by the `ref` argument, you might wish to consider using feature memory to store the extra data.



See Chapter 12, "Storing and Retrieving Data," for more information about using feature memory.

The `alarmSeconds` parameter is where you specify the actual time for the alarm, in seconds since January 1, 1904. The Palm OS provides a number of useful functions for converting different time values to and from this seconds-since-1/1/1904 format. See the "Manipulating Time Values" section, later in this chapter, for descriptions of some of these functions.

The `quiet` argument is reserved for future use. For now, just pass the value `true` for this argument.

An example code snippet follows, which sets an alarm three hours ahead of the current time. The creator ID for this example is stored in the global variable `gAppCreatorID`.

```
UInt16   cardNo;
LocalID  dbID;
DmSearchStateType  searchInfo;
UInt32   alarmTime, nowTime;


alarmTime = nowTime = TimGetSeconds();
alarmTime += 10800;  // 10800 seconds in three hours

DmGetNextDatabaseByTypeCreator(true, &searchInfo,
    sysFileTApplication, gAppCreatorID, true, &cardNo, &dbID);
AlmSetAlarm(cardNo, dbID, nowTime, alarmTime, true)
```

The preceding example not only sets an alarm three hours in the future, but it also stores the current time in the alarm's `ref` parameter via the `nowTime` variable. As a result, the application will know what time the alarm was set when the alarm goes off and the system sends the two alarm-related launch codes.

Because the alarm manager keeps track of only a single alarm for each application, calling **AlmSetAlarm** a second time before the first alarm has gone off replaces the first alarm with a new one. You can use the **AlmGetAlarm** function to retrieve the alarm settings for a particular application before overwriting the application's current alarm. If your application must keep track of more than one alarm time, you need to store that information with the application's data in storage RAM.

Alarms may trigger a bit late when the handheld is "off," or in sleep mode. This happens because the main system clock (which keeps track of time down to the nearest hundredth of a second) is shut down in sleep mode, and the real-time clock on the device (which only keeps track of time to the nearest whole second), still active during sleep mode, does not have the fine granularity of the main system clock. As a result, alarms that trigger while the device is "on" occur at exactly the time you expect them to go off, but alarms that trigger in sleep mode may be almost a minute late.

Responding to Alarms

The first launch code your application needs to respond to when an alarm goes off is `sysAppLaunchCmdAlarmTriggered`. At this point, your application should perform only quick actions in response to the alarm to keep from delaying any alarms set by other applications for the same time. Such actions may include setting the application's next alarm or playing a short sound.

When your application handles `sysAppLaunchCmdDisplayAlarm`, it has a chance to perform lengthier actions in response to the alarm, such as displaying a pop-up dialog box to show the user whatever information the application needs to convey as a result of the alarm's going off.

Unless you need only a couple lines of code to respond to `sysAppLaunchCmd AlarmTriggered` and `sysAppLaunchCmdDisplayAlarm`, you should hand processing of the launch codes from your **PilotMain** routine to other functions. This technique makes the code in your application's **PilotMain** routine more readable and easier to maintain, and it is also easier to process the launch codes' parameter blocks if you need to retrieve any information the system passes with those launch codes. The following **PilotMain** routine passes handling of the two alarm launch codes to the functions **AlarmTriggered** and **DisplayAlarm**:

```
UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
    switch (cmd) {
```

```
            case sysAppLaunchCmdNormalLaunch:
                // Normal launch code handling omitted.

            case sysAppLaunchCmdAlarmTriggered:
                AlarmTriggered((SysAlarmTriggeredParamType *)
                               cmdPBP);
                break;

            case sysAppLaunchCmdDisplayAlarm:
                DisplayAlarm((SysDisplayAlarmParamType *) cmdPBP);
                break;

            default:
                break;
        }

        return 0;
    }
```

Because the `cmdPBP` variable that contains the parameter block for a launch code is actually just a pointer to type `char`, **PilotMain** must cast `cmdPBP` to the appropriate structure before passing the parameter block to another function. The Palm OS header file `AlarmMgr.h` defines the following structures for alarm-related launch codes:

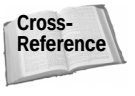
```
typedef struct SysAlarmTriggeredParamType {
    UInt32   ref;
    UInt32   alarmSeconds;
    Boolean  purgeAlarm;
} SysAlarmTriggeredParamType;

typedef struct SysDisplayAlarmParamType {
    UInt32   ref;
    UInt32   alarmSeconds;
    Boolean  soundAlarm;
} SysDisplayAlarmParamType;
```

In both `SysAlarmTriggeredParamType` and `SysDisplayAlarmParamType`, the `ref` member stores extra application-defined data related to the alarm, and `alarmSeconds` holds the actual time of the alarm, in seconds since January 1, 1904.

If application sets the `purgeAlarm` member of `SysAlarmTriggeredParamType` to `true`, the alarm manager does not send a `sysAppLaunchCmdDisplayAlarm` code to the application for that alarm. You should set `purgeAlarm` to `true` if your application completely handles incoming alarms when it takes care of the `sysAppLaunchCmdAlarmTriggered` launch code. By default, `purgeAlarm` is `false`, so if you do not fiddle with the value of `purgeAlarm`, the system will send a `sysAppLaunchCmdDisplayAlarm` code to your application.

According to Palm Computing documentation, the `soundAlarm` member of `SysDisplayAlarmParamType` should be `true` if the alarm is to be sounded, `false` otherwise. However, the system does not currently use `soundAlarm`, so you can safely ignore this value.

Typically, the **AlarmTriggered** function called from the **PilotMain** function in the previous example would contain a call to **SndPlaySystemSound** to trigger the system default alarm sound. If the application stores more than one alarm at a time in its permanent database, **AlarmTriggered** should then look through that database for the next alarm time and set that alarm with a call to **AlmSetAlarm**.



Playing sounds in the Palm OS is covered later in this chapter, under "Playing Sounds." Retrieving values stored in an application's database is covered in Chapter 12, "Storing and Retrieving Data."

Displaying a dialog box in response to `sysAppLaunchCmdAlarmTriggered` can be somewhat tricky, because your application might not be running at the time and must display a dialog box over the top of another application. The following example defines a **DisplayAlarm** function to take care of displaying a simple alarm dialog box to the user:

```
static void DisplayAlarm (SysDisplayAlarmParamType *cmdPBP)
{
    FormType  *form, *curForm;


    form = FrmInitForm(AlarmForm);
    curForm = FrmGetActiveForm();
    if (curForm)
        FrmSetActiveForm(form);
    FrmSetEventHandler(form, AlarmFormHandleEvent);
    FrmDrawForm(form);

    // Do something here with cmdPBP->alarmSeconds or
    // cmdPBP->ref, such as drawing those values to the
    // alarm display using WinDrawChars.

    FrmDoDialog(form);
    FrmDeleteForm(form);
    FrmSetActiveForm(curForm);
}
```

In the example above, **DisplayAlarm** first initializes a new form object with **FrmInitForm**. The **DisplayAlarm** function then saves the currently displayed form in the variable `curForm` so the application can return to whatever form is currently on the screen when the alarm goes off. Storing the active form in this way is important, because the application handling the alarm might not be active when the

alarm goes off, and suddenly changing to another application in the middle of what the user is doing can be disorienting at best, or even downright rude at worst.

After storing the current form for later use, **DisplayAlarm** sets an event handler for the alarm form. Depending on the needs of the application, the **AlarmForm HandleEvent** function can be as simple or as complex as the alarm dialog box requires. One thing that should be included in the alarm dialog box's event handler, though, is a trap for the `appStopEvent` to prevent switching to another application until the user dismisses the dialog box. Switching to another application while the alarm dialog box is displayed can cause crashing behavior because the alarm dialog box is probably displayed over the top of another application. Also, trapping `appStopEvent` serves a very practical purpose: If the user doesn't hear the alarm go off, or has the system alarm sound turned off, the dialog box will be the first thing the user sees when turning on the device again. Here is a short **AlarmForm HandleEvent** function that traps `appStopEvent`:

```
static Boolean AlarmFormHandleEvent(EventType *event)
{
    if (event->eType == appStopEvent)
        return true;
    else
        return false;
}
```

Moving back to the **DisplayAlarm** example, once an event handler has been set for the dialog box, **DisplayAlarm** then draws the alarm dialog box with **FrmDrawForm**. At this point, the application can perform any custom drawing required on the alarm dialog box, possibly using the values passed from the parameter block in the `sysAppLaunchCmdDisplayAlarm` launch code. The **DisplayAlarm** function then displays the dialog box using **FrmDoDialog**. Though the preceding example ignores the return value from **FrmDoDialog**, it could also respond differently depending on which dialog box button the user tapped.

After the user dismisses the alarm dialog box, **DisplayAlarm** removes the dialog box from memory with **FrmDeleteForm**, and then restores the active status of the form that was on the screen before the alarm dialog box appeared using **FrmSetActiveForm**.

Be sure to set the Save Behind attribute of the alarm dialog box when designing its form resource. Without this attribute, the screen quickly becomes a mess as the system tries to draw the alarm dialog box over the top of the current form without first clearing it from the screen.

Responding to Other Launch Codes

There are two other launch codes you should consider responding to in an alarm-enabled application: `sysAppLaunchCmdSystemReset` and `sysAppLaunchCmd TimeChange`.

The Palm OS alarm manager does not keep track of the alarm queue across system resets. This means that after any system reset, all alarms disappear from the system. Fortunately, the system sends a `sysAppLaunchCmdSystemReset` launch code to every application after a system reset. To make your application's alarms persist across system resets, respond to the `sysAppLaunchCmdSystemReset` launch code by retrieving the next appropriate alarm time from your application's stored data, and then setting that time in the system alarm queue with **AlmSetAlarm**.

When the user changes the system time, the OS sends a `sysAppLaunchCmdTimeChange` launch code to every application. Applications that handle alarms might be interested in this event. By default, if an application has an alarm in the queue and the user changes the system clock to a time beyond the set alarm, the alarm immediately goes off when the clock is changed. For most applications, this is desirable behavior, because it ensures that the user will not miss any alarms set between the old system time and the new system time. However, some applications may need to reset alarms in response to a change in system time, and the `sysAppLaunchCmdTimeChange` launch code allows your application to do just that.

Playing Sounds

The current generation of Palm OS devices is very limited in their ability to create sound. With the exception of the TRGPro, which has more advanced sound capabilities than other Palm OS hardware, only a single tone may be generated at any particular time through the device's simple piezoelectric speaker. This limitation makes the handheld unsuitable for music or accurate voice playback without the addition of extra hardware, and it also rules out using the device to generate phone-dialing tones for use as an auto-dialer. The current speaker in Palm devices does work well for simple alarm beeps and user input feedback, though.

The TRGPro's improved speaker design is capable of producing dual tone modulated frequency (DTMF) sounds, but you must use TRG's sound library functions to access this feature of the TRGPro. This book's CD-ROM includes the TRGPro SDK, which contains all the tools and documentation you need to make use of the special features of the TRGPro handheld.

The simplest way to make sounds in the Palm OS is the **SndPlaySystemSound** function, which allows you to generate one of the predefined system sounds. The **SndPlaySystemSound** function takes a single argument of type `SndSysBeepType`, which is an enum defined as follows in `SoundMgr.h`:

```
typedef enum SndSysBeepType {
    sndInfo = 1,
    sndWarning,
    sndError,
    sndStartUp,
```

```
    sndAlarm,
    sndConfirmation,
    sndClick
} SndSysBeepType;
```

Aside from the predefined system sounds, you may also manually generate your own tones using **SndDoCmd**. The **SndDoCmd** function has the following prototype:

```
Err SndDoCmd (MemPtr chanP, SndCommandType *cmdP,
              Boolean noWait)
```

The `chanP` argument to **SndDoCmd** specifies which audio channel should receive the sound output. Because current implementations of the Palm OS support only a single, default channel, you must pass `NULL` for this argument. Likewise, `noWait` is also not fully implemented. In the future, passing `0` for `noWait` will tell the system to play the sound synchronously, which is to say **SndDoCmd** will play the entire sound before returning. A non-zero value specifies asynchronous playback, which means that **SndDoCmd** returns immediately, allowing interruption of the sound playback by other processes. For now, you must pass `0` for this argument.

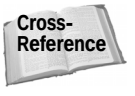
All of the real work of **SndDoCmd** is contained in its second argument, `cmdP`, which is a pointer to a `SndCommandType` structure. The `SndCommandType` structure may contain a number of completely different pieces of information, based on the value of its first member, an enum called `SndCmdIDType`. The Palm OS header file `SoundMgr.h` defines `SndCmdIDType` and `SndCommandType` as follows:

```
typedef enum SndCmdIDType {
    sndCmdFreqDurationAmp = 1,
    sndCmdNoteOn,
    sndCmdFrqOn,
    sndCmdQuiet
} SndCmdIDType;

typedef struct SndCommandType {
    SndCmdIDType  cmd;
    Int32         param1;
    UInt16        param2;
    UInt16        param3;
} SndCommandType;
```

The only option available in versions of the Palm OS prior to 3.0 is `sndCmdFreq DurationAmp`. When this value is passed for the `cmd` member, `param1` specifies the frequency of the sound to play (in Hertz), `param2` specifies the duration of the sound (in milliseconds), and `param3` specifies the amplitude of the sound. When using `sndCmdFreqDurationAmp`, the **SndDoCmd** function always plays the sound synchronously, unless `param3` is `sndMaxAmp` (a constant defined in `SoundMgr.h` that is equal to `0`), in which case **SndDoCmd** plays the sound asynchronously, returning immediately.

All values for `cmd` other than `sndCmdFreqDurationAmp` cause the SndDoCmd function to crash on versions of the Palm OS prior to 3.0. Be sure to check the version before making SndDoCmd calls using the other values in the `SndCmdIDType` enum if you intend for your application to run on earlier versions of the OS.

The `sndCmdNoteOn` function allows you to specify a tone to play using the MIDI (Musical Instrument Digital Interface) format. When using `sndCmdNoteOn`, `param1` specifies the MIDI key index (a value from `0` to `127`), `param2` specifies the duration in milliseconds, and `param3` specifies the sound's velocity (another value from `0` to `127`, which **SndDoCmd** interpolates into an amplitude for playback).



The MIDI format is far too complex a topic to cover completely in this book. For more information about the MIDI specification, point your browser at `http://www.midi.org`, the official Web site of the MIDI Manufacturers Association.

Using `sndCmdFreqOn` is similar to using `sndCmdFreqDurationAmp`, but it always plays asynchronously, allowing you to interrupt the sound playback with another call to **SndDoCmd**. When `cmd` is `sndCmdFreqOn`, `param1` specifies frequency in Hertz, `param2` specifies duration in milliseconds, and `param3` specifies amplitude (use the `sndMaxAmp` constant to set the sound at the maximum amplitude).

The final value in the `SndCmdIDType` enum is `sndCmdQuiet`, which stops the play of the current sound. All three `param` variables should be set to `0` when using `sndCmdQuiet`.

Looking Up Phone Numbers

In Palm OS version 2.0 and later, the system provides an easy way for the user to look up a phone number in the built-in Address Book application from any text field. It is almost as simple to add this feature in your own application. All that is required to implement a phone number lookup is a text field and the **PhoneNumberLookup** function.

The **PhoneNumberLookup** function takes a single argument: a pointer to a field object. When called, **PhoneNumberLookup** first searches the Address Book application for whatever text is currently selected in the specified field. If no text is selected in the field, **PhoneNumberLookup** tries to find whatever word is closest to the insertion point.

If **PhoneNumberLookup** finds a match, it replaces the current selection, if any, with the full name and phone number of the record the function found in the Address Book database. If **PhoneNumberLookup** cannot immediately find an unambiguous match, the function displays the Address Book's Phone Number Lookup form to allow the user to manually select a record.

To maintain consistency with the built-in applications, any program that implements phone number lookup should have an Options➪Phone Lookup menu item, with an "L" character for its Graffiti command shortcut. The Librarian sample application, like the built-in applications, implements a phone lookup in its Note view. The following section of Librarian's **NoteViewHandleEvent** function launches a phone number lookup when the user selects the Options➪Phone Lookup menu item or enters an "L" after a command stroke:

```
static Boolean NoteViewDoCommand(UInt16 command)
{
    FieldType  *field;
    Boolean    handled = true;

    switch (command) {
        case notePhoneLookupCmd:
            field = GetObjectPtr(NoteField);
            PhoneNumberLookup(field);
            break;

        // Other menu items omitted

        default:
            handled = false;
    }

    return (handled);
}
```

Launching Applications

Normally, launching an application is a simple matter of tapping the silk-screened Applications button and selecting a program from the Palm OS application launcher. Well-designed Palm OS applications make this a painless operation by saving whatever data the user is working on before switching to the launcher. The constant readiness of a Palm OS application to drop whatever it's doing and allow the user to switch to another application is part of what makes a Palm OS device convenient and fast to use.

However, some applications, particularly replacements for the system launcher application, may need to call up the launcher without waiting for the user to tap the Applications button. In other cases, an application may need to launch another program directly, without using the system launcher at all. Also, it can be useful for an application to send a specific launch code to another application to request that it perform some sort of action or modify its data in some way.

Calling the System Application Launcher

Starting with Palm OS version 3.0, the application launcher is an independent application stored in the handheld's ROM. Prior to 3.0, the launcher is a pop-up dialog box. Regardless of this difference, there is a method of displaying the launcher that works on any version of the Palm OS. Simply queue a `keyDownEvent` that contains the special `launchChr` character, and the system takes care of showing the launcher. The following code assembles the `keyDownEvent` and adds it to the event queue with **EvtAddEventToQueue**:

```
EventType  newEvent;

newEvent.eType = keyDownEvent;
newEvent.data.keyDown.chr = launchChr;
newEvent.data.keyDown.modifiers = commandKeyMask;
EvtAddEventToQueue(&newEvent);
```

To display the launcher pop-up on Palm OS 2.0 and earlier, you may also use the SysAppLauncherDialog function, which requires no arguments and has no return value. The SysAppLauncherDialog function still exists in post-2.0 versions of the Palm OS for backward compatibility. However, you should always queue a `keyDownEvent` containing a `launchChr` to bring up the system application launcher to ensure that your code will continue to work without modification on future versions of the operating system.

Launching Applications Directly

Two functions in the Palm OS are responsible for launching other applications: **SysAppLaunch** and **SysUIAppSwitch**. Both functions allow you to customize how you wish to launch an application, giving you control over the launch code, launch flags, and parameter block to send to the other application. The **SysAppLaunch** function is for making use of another program, and then returning to the original application, while **SysUIAppSwitch** quits the current application and starts another in its place.

Do not use SysAppLaunch or SysUIAppSwitch to call the system application launcher. If another application has replaced the default launcher, that application will open instead of the default system launcher. Instead, queue a `keyDownEvent` containing a `launchChr` to open the system launcher, as described earlier in this chapter.

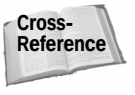
The most common use for **SysAppLaunch** is to send launch codes to other applications, which enables an application to make use of another program's features to perform a task. In effect, **SysAppLaunch** allows a program to call a specific subroutine in another application. As an example, the Palm OS **PhoneNumberLookup** function, discussed earlier in this chapter, uses **SysAppLaunch** to send a

`sysAppLaunchCmdLookup` launch code to the Address Book application, telling the Address Book to search its database for a particular name and return the phone number associated with that name.

The prototype for **SysAppLaunch** looks like this:

```
Err SysAppLaunch (UInt16 cardNo, LocalID dbID,
    UInt16 launchFlags, UInt16 cmd, MemPtr cmdPBP,
    UInt32 *resultP)
```

The `cardNo` and `dbID` arguments identify the application that **SysAppLaunch** should call, specifying the memory card where that application is located and its unique database ID. You can retrieve the values for `cardNo` and `dbID` with the **DmGetNextDatabaseByTypeCreator** function, which returns the card and database ID given the creator ID of the application.



Chapter 12, "Storing and Retrieving Data," contains more information about DmGetNextDatabaseByTypeCreator.

To send a launch code to another application, pass the value `0` for the `launchFlags` argument. Other launch flags exist, but they are usually needed only by the system; see Appendix A, "Palm OS API Quick Reference," for a list of available launch flags.

The `cmd` argument specifies the launch code to send to the other application, and `cmdPBP` points to a parameter block structure containing information the called application needs to process the launch code. When the other application is finished, **SysAppLaunch** uses `resultP` to return a pointer to the result of the called application's **PilotMain** routine.

The **SysUIAppSwitch** function takes the same parameters as **SysAppLaunch** but does not have a `resultP` parameter, because **SysUIAppSwitch** tells the current application to quit before launching the new program. However, an extra step may be necessary when sending anything other than a `sysAppLaunchCmdNormalLaunch` launch code. If you pass a parameter block to the new application via the `cmdPBP` argument, you should grant ownership of the parameter block to the system with the **MemPtrSetOwner** function. Otherwise, the system will free the memory allocated for the parameter block when the calling application quits. The **MemPtrSetOwner** function takes two parameters, the pointer itself and the ID of the new owning application. Passing `0` for the second parameter assigns ownership of the pointer to the system. For example, the following line of code changes ownership of `cmdPBP` to the system:

```
Err error = MemPtrSetOwner(cmdPBP, 0);
```

The `error` return value from **MemPtrSetOwner** contains `0` if there is no error while changing ownership, or `error` contains the constant `memErrInvalidParam` if an

error occurs. Another function, **MemHandleSetOwner**, exists for changing ownership of handles, which must also be done for any handles within the parameter block. The **MemHandleSetOwner** function works the same way as **MemPtrSetOwner**, substituting a handle for the pointer in the function's first argument.

To simplify calls to **SysAppLaunch** and **SysUIAppSwitch**, the Palm OS provides two macros, **AppCallWithCommand** and **AppLaunchWithCommand**, which incorporate a call to **DmGetNextDatabaseByTypeCreator**. These macros allow you to send launch codes and launch another application without having to manually find the application's card number and database ID. Listing 10-1 shows the definitions of both of these macros, which are supplied in the Palm OS header file `AppLaunchCmd.h`.

Listing 10-1: The AppCallWithCommand and AppLaunchWithCommand macros

```
#define AppCallWithCommand(appCreator, appCmd, appCmdParams) \
{ \
    UInt16   cardNo; \
    LocalID  dbID; \
    DmSearchStateType  searchState; \
    UInt32   result; \
    Err      err; \
    DmGetNextDatabaseByTypeCreator(true, &searchState, \
        sysFileTApplication, appCreator, true, &cardNo, \
        &dbID); \
    ErrNonFatalDisplayIf(!dbID, "Could not find app"); \
    if (dbID) { \
        err = SysAppLaunch(cardNo, dbID, 0, appCmd, (Ptr) \
                           appCmdParams, &result); \
        ErrNonFatalDisplayIf(err, "Could not launch app"); \
    } \
}

#define AppLaunchWithCommand(appCreator, appCmd, appCmdParams)\
{ \
    UInt16   cardNo; \
    LocalID  dbID; \
    DmSearchStateType  searchState; \
    Err      err; \
    DmGetNextDatabaseByTypeCreator(true, &searchState, \
        sysFileTApplication, appCreator, true, &cardNo, \
        &dbID); \
    ErrNonFatalDisplayIf(!dbID, "Could not find app"); \
    if (dbID) { \
        err = SysUIAppSwitch(cardNo, dbID, appCmd, \
                             appCmdParams); \
        ErrNonFatalDisplayIf(err, "Could not launch app"); \
    } \
}
```

Both macros have three arguments. The first, `appCreator`, is the creator ID of the application that should be launched. The second, `appCmd`, is the launch code to send to the application, and the third, `appCmdParams`, is a pointer to the parameter block that should be passed along with the launch code.

As an example of how to use these macros, CodeWarrior's project stationery for the Palm OS includes a function called **RomVersionCompatible**, which checks the current version of the operating system when the application starts to see if it meets a minimum version requirement. If the system version is older than the required version, **RomVersionCompatible** alerts the user about the version requirement, and then calls **AppLaunchWithCommand** to launch a "safe" default application that is guaranteed to exist on that version of the operating system. The relevant lines of code from **RomVersionCompatible** are shown in the following example:

```
if (romVersion < sysMakeROMVersion(2,0,0,sysROMStageRelease,0))
    AppLaunchWithCommand(sysFileCDefaultApp,
                         sysAppLaunchCmdNormalLaunch, NULL);
}
```

The constant `sysFileCDefaultApp` is defined in every version of the Palm OS headers to refer to a default application that exists in that version of the operating system. On Palm OS 1.0 and 2.0, this default is the Memory application; on Palm OS 3.0 and later, the system Preferences application is the default, because memory display on the more recent versions of the OS is part of the system launcher application, and the Memory application no longer exists.

Sending Launch Codes Globally

If you need to send a particular launch code to every application on the device, use the **SysBroadcastActionCode** function. This function takes two arguments: the launch code to send, and a pointer to the parameter block containing information needed by any applications that respond to the launch code.

Creating Your Own Launch Codes

The Palm OS allows you to define your own launch codes. This feature can allow two applications to talk to each other and control each other's data behind the scenes without the user's ever being aware of what the programs are doing. Using your own launch codes, you can create suites of applications that communicate with each other, shared libraries that may be called from many different applications, or very large applications that are composed of multiple smaller applications that communicate with one another via launch codes.

Launch codes in the Palm OS are 16-bit values. Codes from `0` through `32767` are reserved for use by Palm Computing for their own launch codes and future

enhancements to the operating system, which leaves numbers from 32768 to 65535 for your own applications, which should be more space than any application could realistically need for launch codes. Using your own launch codes is a simple matter of deciding what a particular number represents, and then making sure that your applications respond to that number in their **PilotMain** routines.

Generating Random Numbers

The **SysRandom** function is the only built-in pseudo-random number generator in the Palm OS. For most purposes, **SysRandom** should be sufficient for the task of creating random numbers. Some applications that require very random numbers, such as strong cryptography programs, may not work very well with **SysRandom**, requiring that you provide your own algorithms. The limited processor power available on Palm OS devices pretty much rules them out for applications that require this kind of complex number-crunching, though, so for any solution that requires very random numbers, you may want to put together a companion application that runs on a desktop computer.

The **SysRandom** function returns an integer value from 0 to sysRandomMax, which the Palm OS headers define as 32,767. To generate a random number with **SysRandom**, pass the function an unsigned long integer value to use as a seed value, or 0 to use the last seed value. The best way to ensure that your application produces reasonably random results is to seed the random number generator from the system clock when you first start the application. The following line of code in the application's **StartApplication** function will seed the random number generator from the Palm OS device's onboard clock:

```
SysRandom(TimGetTicks());
```

After the random number generator has been seeded, call **SysRandom** with a value of 0 each time you need a new random number. With a little simple math, you can use **SysRandom** to come up with almost any random number. The following simple function returns a number from 0 to n - 1, where n is the total number of possible choices:

```
UInt16 RandomNum(UInt n) {
    return SysRandom(0) / (1 + sysRandomMax / n);
}
```

For example, passing the value 52 for n causes **RandomNum** to return a value between 0 and 51, which would be appropriate for randomly drawing a card from a standard poker deck.

Random numbers, although necessary for some applications, can make debugging a nightmare, because the application may not produce the same results each time it is run. When debugging an application that calls SysRandom, seed the random number generator with a constant number instead of with TimGetTicks, like this:

```
SysRandom(1);
```

When seeded in this way, the random number generator will produce the same series of "random" numbers each time you run the application, greatly reducing your frustration level when debugging. When you are ready to distribute the application, replace the constant with a call to TimGetTicks so the application will produce different random numbers each time you run it.

Managing Power

The Palm OS provides the **SysBatteryInfo** function for determining information about the handheld's battery settings. The prototype for **SysBatteryInfo** looks like this:

```
UInt16 SysBatteryInfo (Boolean set, UInt16 *warnThresholdP,
    UInt16 *criticalThresholdP, UInt16 *maxTicksP,
    SysBatteryKind *kindP, Boolean *pluggedIn, UInt8 *percentP)
```

The **SysBatteryInfo** function returns a value equal to the current battery voltage in hundredths of a volt, which allows the system to store the value as an integer instead of as a floating-point number. Divide the return value from **SysBatteryInfo** by 100 to obtain the actual voltage level of the batteries in volts.

The first argument, `set`, should be `false` to retrieve battery settings. Presumably, a `true` value for `set` would change the settings, but the Palm Computing documentation states that applications should never change battery settings. This restriction is understandable, because changing the voltage threshold levels at which the device warns the user about a low battery condition could potentially result in data loss because of a drained battery. Be careful when using **SysBatteryInfo**.

Pass `NULL` for any of the remaining arguments to **SysBatteryInfo** that you wish to ignore. For any value you wish to retrieve, allocate a variable of the appropriate type and pass a pointer to that variable to the **SysBatteryInfo** function.

The `warnThresholdP` argument is a pointer to a variable to receive the system battery warning threshold. The battery warning threshold is the voltage at which the system first displays a warning to the user, stating that the batteries are low and should be changed or recharged. Like the return value of **SysBatteryInfo**, the value that `warnThresholdP` points to is the threshold's voltage level in hundredths of a volt. The `criticalThresholdP` argument points to a variable to receive the critical

battery threshold. When the battery voltage reaches the critical threshold level, the device drops to sleep mode without warning and will not return to normal operation until the battery voltage is above the critical threshold again. Again, the value `criticalThresholdP` points to is the threshold voltage level in hundredths of a volt.

The `maxTicksP` argument points to a variable to receive the value of the system battery timeout value, in system ticks, which the system uses internally to determine when to display the low battery warning dialog box to the user.

The `kindP` argument points to a variable that receives the type of battery installed in the device. The battery type alters the values that the system uses to calculate remaining voltage in the battery. Different battery types are defined in the Palm OS header file `SystemMgr.h` in the `SysBatteryKind` enum:

```
typedef enum {
    sysBatteryKindAlkaline = 0,
    sysBatteryKindNiCad,
    sysBatteryKindLiIon,
    sysBatteryKindRechAlk,
    sysBatteryKindLast = 0xFF
} SysBatteryKind;
```

The `pluggedIn` argument indicates whether the device is plugged into external power, which might be the case for a Palm V or similar device with a rechargeable internal battery when it is resting in its cradle. A `true` value for `pluggedIn` indicates that the device is plugged in, and `false` indicates that the device is not attached to external power.

Finally, the `percentP` contains the approximate percentage of power left in the device's batteries. Because the power regulation hardware on a Palm OS device is not very sophisticated, it is difficult for the system to determine the exact voltage in the device's batteries. This inaccuracy can result in a brand new set of batteries appearing to have less than 100 percent power, so **SysBatteryInfo** actually fudges the `percentP` value, returning 100 when the calculated percentage is 90 or higher.

The Palm OS also provides the function **SysBatteryInfoV20** for backward compatibility reasons. The **SysBatteryInfoV20** function is exactly like **SysBatteryInfo**, except that it has no `percentP` argument.

Reacting to Low Battery Conditions

When the batteries in a Palm OS device reach the battery warning threshold level, the system queues a `keyDownEvent` containing a special `lowBatteryChr` character code. Normally, letting this event fall through to the system's **SysHandleEvent** routine in your application's event loop should be sufficient, because **SysHandleEvent** reacts to this event by displaying the system low battery dialog box. If, for some

reason, your application needs to know when the low battery warning threshold has been reached, you can react to it in your own application's event handlers by reacting to the `lowBatteryChr` event.

If you do react to a `lowBatteryChr` event, be sure not to mark the event as handled so that SysHandleEvent still gets a chance to display the system low battery warning dialog.

Identifying the Device

Starting with the Palm III, some Palm OS devices have a unique 12-digit serial number in their ROM storage to identify the device. This serial number is stored in a text buffer with no null terminator. The user can view a device's serial number in the application launcher's Info view. Serial numbers may be used to implement copy protection for an application that is keyed to a specific handheld. In Palm OS devices that support wireless connections, the serial number may be used for authentication purposes to verify the security of a connection between the handheld and a remote server.

To retrieve the serial number from ROM, use the **SysGetROMToken** function. The prototype for **SysGetROMToken** looks like this:

```
Err SysGetROMToken (UInt16 cardNo, UInt32 token, UInt8 **dataP,
                    UInt16 *sizeP)
```

If **SysGetROMToken** successfully retrieves a valid serial number, the function's return value should be `0`. Because of the inner workings of **SysGetROMToken**, you should also verify that `dataP` is not `NULL`. Also, if `dataP` does contain data, its first character should not be `0xFF`. Once you have verified that all these things are true, you can be sure that **SysGetROMToken** has retrieved a valid serial number.

The `cardNo` argument is the memory card holding the ROM to be queried; because no Palm OS device currently has more than one card, pass `0` for this value.

Pass the constant `sysROMTokenSnum`, defined in the Palm OS `SystemMgr.h` header file, for the `token` argument. The **SysGetROMToken** function is capable of retrieving information from ROM other than the device's serial number, but the serial number is the only bit of information Palm Computing has made publicly available to developers, so `sysROMTokenSnum` is the only thing you can pass in the `token` argument and expect to get a useful result from **SysGetROMToken**.

The `dataP` argument is a pointer to a text buffer to hold the serial number **SysGet ROMToken** retrieves, and `sizeP` is a pointer to a variable to receive the size of the retrieved text, in bytes.

As an example, the following function draws a device's serial number on the screen at a given location, or a short message indicating the device's lack of a serial number:

```
static void DrawSerialNumber(Int16 x, Int16 y)
{
    char    *buffer;
    UInt16  length;
    Err     error;


    error = SysGetROMToken(0, sysROMTokenSnum, (BytePtr *)
                           &buffer, &length);
    if ( (! error) && (buffer) && ( (Byte) *buffer != 0xFF) )
        // There is a valid serial number, so draw it.
        WinDrawChars(buffer, length, x, y);
    else
        // There isn't a valid serial number, so draw a message
           indicating this to the user.
        WinDrawChars("No serial number", 16, x, y);
}
```

Manipulating Time Values

Because keeping track of dates and times is a common use for handheld computers, the Palm OS provides many functions for retrieving, converting, and altering time values.

Internally, the system's real-time clock keeps track of time as a 32-bit unsigned integer, representing the number of seconds since midnight on January 1, 1904. The real-time clock keeps track of date and time in 1-second increments, even while the device is in sleep mode. A Palm OS device also has a faster timer, which keeps track of time in *system ticks*. System ticks occur 100 times per second on an actual Palm OS device, or 60 times per second in a Palm OS Simulator application running on a Macintosh. The system resets the ticks counter to 0 whenever the device is reset, and the ticks counter does not update while the device is in sleep mode.

The Palm OS uses three basic structures in many of its time functions to keep track of the date, the time, or the date and time as a single unit. These structures are defined in the Palm OS header file `DateTime.h`, and they are repeated in the following example for reference:

```
typedef struct {
    Int16  second;
    Int16  minute;
    Int16  hour;
    Int16  day;
    Int16  month;
    Int16  year;
```

```
    Int16  weekDay;  // Days since Sunday (0 to 6)
} DateTimeType;

typedef struct {
    UInt8  hours;
    UInt8  minutes;
} TimeType;

typedef struct {
    UInt16  year  :7;  // Years since 1904 (Mac format)
    UInt16  month :4;
    UInt16  day   :5;
} DateType;
```

Retrieving and Setting Time Values

The **TimGetSeconds** function retrieves the current time from the system's real-time clock, in seconds since midnight, January 1, 1904. You can also set the system clock by passing the appropriate seconds past 1/1/1904 value to the **TimSetSeconds** function.

You can implement finer timing by using the **TimGetTicks** function, which retrieves the number of ticks that have passed since the last time the device was soft reset. Because ticks can vary in length between devices, you should use the **SysTicksPer Second** macro to retrieve the number of ticks per second on the device, and then divide the value from **TimGetTicks** by the value returned by **SysTicksPerSecond** to get the actual time in seconds since the device was last reset.

Converting Time Values

The **TimSecondsToDateTime** function converts between the internal seconds past 1/1/1904 value to a `DateTimeType` structure. You can convert from a `DateTimeType` structure back to seconds using **TimDateTimeToSeconds**. Much like the **TimSeconds ToDateTime** function, **DateSecondsToDate** converts seconds since 1/1/1904 into a `DateType` structure, but there is no corresponding function to convert a `DateType` structure back into seconds.

More functions that use the `DateType` structure include **DateDaysToDate**, which converts the number of days since 1/1/1904 into a `DateType` structure, and **DateToDays**, which converts a `DateType` structure into the number of days past 1/1/1904.

The function **DateToAscii** allows conversion of a particular date to an ASCII string, suitable for display to the user. The prototype for **DateToAscii** looks like this:

```
void DateToAscii (UInt8 months, UInt8 days, UInt16 years,
    DateFormatType dateFormat, Char *pString)
```

The first three arguments to **DateToAscii** allow you to specify the month (from 1 to 12), day (from 1 to 31), and year (in four-digit format) to convert to text. The dateFormat argument specifies the format that the date should take, as defined by the DateFormatType enum in the Palm OS header file Preferences.h:

```
typedef enum {
    dfMDYWithSlashes,        // 12/31/95
    dfDMYWithSlashes,        // 31/12/95
    dfDMYWithDots,           // 31.12.95
    dfDMYWithDashes,         // 31-12-95
    dfYMDWithSlashes,        // 95/12/31
    dfYMDWithDots,           // 95.12.31
    dfYMDWithDashes,         // 95-12-31

    dfMDYLongWithComma,      // Dec 31, 1995
    dfDMYLong,               // 31 Dec 1995
    dfDMYLongWithDot,        // 31. Dec 1995
    dfDMYLongNoDay,          // Dec 1995
    dfDMYLongWithComma,      // 31 Dec, 1995
    dfYMDLongWithDot,        // 1995.12.31
    dfYMDLongWithSpace,      // 1995 Dec 31

    dfMYMed,                 // Dec '95
    dfMYMedNoPost            // Dec 95
} DateFormatType;
```

The pString argument is a pointer to a string to receive the converted date text. The string pointed to by pString must be of length dateStringLength for short date strings, or length longDateStringLength for long date formats.

The **DateToDOWDMFormat** takes the same arguments as **DateToAscii**, but it adds a three-letter day of week abbreviation to the front of whatever string format is specified by the dateFormat argument. For example, the following call to **DateToDOWDM Format** fills the variable dateStr with the string Sun Dec 31, 1995:

```
DateToDOWDMFormat(12, 31, 1995, dfMDYLongWithComma, dateStr);
```

If you need to determine which ordinal day of the month a particular date lies on, you can use the **DayOfMonth** function. The value returned by this function is not the cardinal date (for example, 31 for the 31st of December), but rather the day's relative position within the month (for example, the last Sunday of December). The **DayOfMonth** function returns a value from the enum DayOfWeekType, defined in DateTime.h as follows:

```
typedef enum {
    dom1stSun, dom1stMon, dom1stTue, dom1stWen, dom1stThu,
        dom1stFri, dom1stSat,
    dom2ndSun, dom2ndMon, dom2ndTue, dom2ndWen, dom2ndThu,
```

```
        dom2ndFri, dom2ndSat,
    dom3rdSun, dom3rdMon, dom3rdTue, dom3rdWen, dom3rdThu,
        dom3rdFri, dom3rdSat,
    dom4thSun, dom4thMon, dom4thTue, dom4thWen, dom4thThu,
        dom4thFri, dom4thSat,
    domLastSun, domLastMon, domLastTue, domLastWen, domLastThu,
        domLastFri, domLastSat
} DayOfWeekType;
```

The **DaysInMonth** function returns the number of days in a month, given a month and a year, and the Palm OS also provides the **DayOfWeek** function, which, given a month, day, and year, returns the day of the week as a value from `0` to `6`, where `0` represents Sunday.

Altering Time Values

Two functions, **DateAdjust** and **TimAdjust**, allow for quick changes in dates, which free you of the burden of manually changing the month or the year when the addition or subtraction of a certain amount of time from an initial date would cause the date to wrap to a new month or year.

The **DateAdjust** function takes a pointer to a `DateType` structure and a number of days as arguments, altering the passed `DateType` structure by the specified number of days. If the number of days is positive, **DateAdjust** adds the days to the date; for a negative number of days, **DateAdjust** subtracts the number of days.

The **TimAdjust** function works like **DateAdjust**, except that it modifies a `DateTime Type` structure by a specified number of seconds.

Using the Clipboard

The built-in applications allow for cutting, copying, and pasting of data between text fields. The area of memory the system maintains for this kind of temporary data storage is called the *clipboard*.

There are actually three different clipboards, each used to store a different kind of data. The header file `Clipboard.h` defines the following enumerated type to identify these different types of clipboard data:

```
enum clipboardFormats {
    clipboardText,
    clipboardInk,
    clipboardBitmap
};
typedef enum clipboardFormats ClipboardFormatType;
```

The `clipboardText` format stores textual data, and the `clipboardBitmap` format stores bitmap image data. As of this writing, the `clipboardInk` format is reserved for future use, but as its name implies, it is probably intended for storing "digital ink," a hybrid between text and bitmap data. Digital ink data represents something that the user has drawn directly on the screen, composed of a doodle or text that has not been converted by the Graffiti engine, or even a mixture of pictures and text.

Adding cut, copy, and paste behavior to a field is the easiest way to use the clipboard. The functions **FldCut**, **FldCopy**, and **FldPaste** all take a pointer to a field object as an argument:

```
void FldCut (FieldType* fldP)
void FldCopy (const FieldType* fldP)
void FldPaste (FieldType* fldP)
```

Both **FldCut** and **FldCopy** copy the currently selected text from the indicated field and place it on the text clipboard. The **FldCut** function also deletes the selected text from the field. The **FldPaste** function replaces the currently selected text in the field with the contents of the text clipboard, or if there is no selection in the text, inserts the contents of the clipboard at the field's insertion point.

If you need to place text data on the clipboard directly, without using a field, or if you want to put bitmap data on the clipboard, you should use the **ClipboardAddItem** function. The prototype for **ClipboardAddItem** looks like this:

```
void ClipboardAddItem (const ClipboardFormatType format,
    const void *ptr, UInt16 length)
```

The first argument to **ClipboardAddItem** is `format`, which specifies which clipboard, text or bitmap, should accept the incoming data. The other two arguments, `ptr` and `length`, are a pointer to the beginning of the data and the length of that data in bytes, respectively. Whatever data you place on a clipboard with **ClipboardAddItem** overwrites the current contents of that clipboard.

The maximum size of text data on the clipboard is 1000 bytes.

Starting with Palm OS version 3.2, the **ClipboardAppendItem** function is available, which allows you to write more data onto the end of the clipboard without deleting what is already there, allowing you to build up larger clipboard contents from many smaller pieces of text. The **ClipboardAppendItem** function takes the same parameters as **ClipboardAddItem**. Because **ClipboardAppendItem** is only intended for use with text data, you should be sure to pass the `clipboardText` constant for the first argument to **ClipboardAppendItem**.

You can retrieve data from the clipboard with the **ClipboardGetItem** function, which has the following prototype:

```
MemHandle ClipboardGetItem (const ClipboardFormatType format,
    UInt16 *length)
```

The **ClipboardGetItem** function returns a handle to the actual memory chunk that contains the requested clipboard data. Because this is a handle to the clipboard's actual memory, attempting to modify the contents of this handle directly will result in an error. Copy the data out of the handle with the **MemCopy** function before attempting to modify the data. Likewise, do not free the handle returned from **ClipboardGetItem**; the system frees this handle automatically the next time data is copied to the clipboard.

Text data in the clipboard is not `NULL`-terminated, so be sure to use the value returned in the ClipboardGetItem function's `length` parameter to determine how much text is in the clipboard.

Summary

In this chapter, you got to take a look at functions that help you program the guts of your application, the parts that do all the work but that the user never sees. After reading this chapter, you should understand the following:

- Not all features of the Palm OS are present in all devices or in all versions of the OS, but the system does provide a way to query what features exist by using the **FtrGet** function.
- The Palm OS provides many functions for manipulating text, including font functions for handling on-screen display of text and string functions that mirror many basic functions in the C standard library.
- You can directly handle pen and key events before the operating system gets them to allow your application to do special things with stylus and hardware button input.
- Setting alarms in the Palm OS requires calling the **AlmSetAlarm** function, as well as handling the `sysAppLaunchCmdAlarmTriggered`, `sysAppLaunchCmdDisplayAlarm`, `sysAppLaunchCmdTimeChange`, and `sysAppLaunchCmdSystemReset` launch codes.
- The Palm OS allows you to programmatically activate the system's application launcher, or you can launch applications manually within your program's code using the **SysAppLaunch** and **SysUIAppSwitch** functions.
- Because managing time is an important part of a handheld digital assistant, the Palm OS provides a very complete set of functions for manipulating, converting, and setting time and date values.
- Many other utility functions exist in the Palm OS for doing things such as playing sounds, looking up phone numbers in the built-in Address Book, generating random numbers, managing battery power, and retrieving the device's onboard serial number.

✦ ✦ ✦

CHAPTER 11

Programming Tables

In This Chapter

Understanding how tables work

Initializing tables

Handling table events

Hiding and showing rows and columns

Attaching data to a table

Scrolling tables

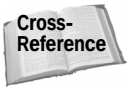
Handling table text fields

Tables are some of the most complex user interface elements in the Palm OS and, hence, some of the most difficult to implement. Although many internal functions of tables are handled by the Palm OS table manager, the system does not provide much of what users of the ROM applications expect as default behavior from a table. In this regard, tables are somewhat like gadgets; the Palm OS provides you with a user interface object to attach to your program, but much of the hard work that makes the object tick comes from the application itself. Fortunately, the built-in applications provide good examples of how to implement tables that operate in ways users expect to see, and much of the code required to operate a table is simply a boilerplate that you can modify slightly and re-use in your own applications.

Palm makes the source code for the Table Manager available on their Web site (`http://www.palm.com`). Looking through the source is an excellent way to gain insight into the subtle nuances of tables.

Both tables and lists tend to look and act in similar ways. How do you decide which to use, a table or a list? Tables are more suited to editing data in place, and lists are optimized for selecting items. If you want to allow a user to edit data in a tabular format, such as the way the interface for the Date Book and To Do List applications works, use a table. If you just want to present the user with a list of choices, a list is much easier to use.

This chapter is divided into two sections, “Creating a Simple Table” and “Creating More Complex Tables.” The first section explains the basic mechanics of creating, initializing, and drawing a simple table by way of a small sample program. The second section examines adding more complex behavior to a table, such as database interaction, scrolling, and expanding text fields, and uses the Librarian sample application to demonstrate these techniques.



Most tables are intimately linked to an application's database, so some Palm OS database terminology appears throughout this chapter. Refer to Chapter 12, "Storing and Retrieving Data," and to Chapter 13, "Manipulating Records," for more information about Palm OS database routines.

Creating a Simple Table

The example application in this section of the chapter demonstrates all of the available table item types. Figure 11-1 shows Table Example as it looks after you start the program.

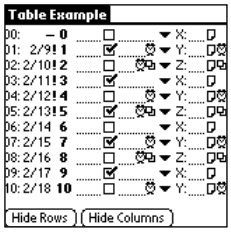


Figure 11-1: The Table Example application

The Table Example program and source code are available on the CD-ROM that accompanies this book.

Table Example contains a single main form, which is host to only four objects: the table, a list (hidden from view until invoked by the table's pop-up triggers), and two buttons for demonstrating how to hide rows and columns in the table. The following PilRC resource definition, from the file `table.rcp`, defines the form and its elements:

```
FORM ID MainForm 0 0 160 160
MENUID MainFormMenuBar
USABLE
BEGIN
    TITLE "Table Example"
    TABLE ID MainTable AT (0 16 160 121) ROWS 11 COLUMNS 9
        COLUMNWIDTHS 12 25 12 18 12 33 17 20 9
    BUTTON "Hide Rows"    ID MainHideRowsButton
        AT (1 147 50 12)
    BUTTON "Hide Columns" ID MainHideColumnsButton
        AT (56 147 64 12)
    GRAFFITISTATEINDICATOR AT (140 PrevTop)
    LIST "X" "Y" "Z" ID MainList AT (120 141 19 33) NONUSABLE
        VISIBLEITEMS 3
END
```

The table in the example program is 11 rows high, which in the standard Palm OS font fills most of the screen, leaving just enough room across the bottom of the

form for command buttons. There are nine columns in the table, one for each kind of table item supported by the table manager. Table items are described in the next section.

Understanding How Tables Work

Palm OS tables are essentially containers for a variety of other form elements, such as pop-up lists, text fields, and check boxes, and as such, each table maintains a complex array of subordinate controls. The Palm OS table manager keeps track of each item in a table with the `TableItemType` structure, which is declared as follows in `Table.h`:

```
typedef struct {
    TableItemStyleType  itemType;
    FontID              fontID;
    Int16               intValue;
    Char *              ptr;
} TableItemType;
```

The `itemType` field in the structure determines how the system draws each item, and it also controls which of the other three `TableItemType` fields are used with a particular table item. Different table items store different data in the `fontID`, `intValue`, and `ptr` fields. Some table item types allow the user to edit the value displayed in their table cells, whereas others are for display purposes only. Table 11-1 provides an overview of the available values for `itemType` and shows which are user-editable and which of the other three `TableItemType` fields each item type uses.

Table 11-1
Table Item Types

Item Type	*Editable by User?*	*TableItemType Fields Used by This Type*
`checkboxTableItem`	Yes	`intValue`
`customTableItem`	Yes	None, although you may store data in the `intValue` and `ptr` fields if required by your application.
`dateTableItem`	No	`intValue`
`labelTableItem`	No	`ptr`
`numericTableItem`	No	`intValue`
`popupTriggerTableItem`	Yes	`intValue`, `ptr`
`textTableItem`	Yes	`fontID`, `ptr`
`textWithNoteTableItem`	Yes	`fontID`, `ptr`
`narrowTextTableItem`	Yes	`fontID`, `intValue`, `ptr`

The Palm OS provides several functions for retrieving and setting values in a table item. The **TblSetItemStyle** function sets the value of a table item's `itemType` field, given a pointer to the table, the row and column of the item to set, and the type desired. In a similar vein, **TblSetItemFont** sets the `fontID` for a table item, **TblSetItemInt** sets the `intValue`, and **TblSetItemPtr** sets the `ptr` field. There are also three functions for retrieving table item values — **TblGetItemFont**, **TblGetItemInt**, and **TblGetItemPtr** — which all retrieve the desired value given a pointer to the table and the row and column of the desired table item. The use of these functions is covered in more detail later in this chapter.

Always use the various Palm OS table functions to modify the values of a table item's `TableItemType` structure. Directly editing this structure's values can confuse the table manager, resulting in unpredictable behavior and possible system crashes.

The following sections describe each data type in more detail.

checkboxTableItem

A `checkboxTableItem` is a simple check box without a label. The user may toggle the check box on or off by tapping the table cell containing the check box. This table item stores the value of the check box in `intValue`, with `0` representing an unchecked box and `1` representing a checked box.

customTableItem

The `customTableItem` type is the table equivalent of a gadget object, allowing you to create your own type of table item if none of the others fits the bill. Your application code should install a callback routine to draw the contents of a `customTableItem` cell. The callback function may use the cell's `intValue` and `ptr` fields to store whatever data might be required by the custom cell.

dateTableItem

This table item is display-only, showing the date in the form `month/day`. The date itself is stored in the table item's `intValue` field, and it should be a value that can be cast as a `DateType`. The Palm OS header file `DateTime.h` defines `DateType` as follows:

```
typedef struct {
    UInt16 year  :7;  // years since 1904 (MAC format)
    UInt16 month :4;
    UInt16 day   :5;
} DateType;
```

If the value of `intValue` is `-1`, the date table item displays a hyphen (`-`) instead of a date, and if the date in `intValue` occurs on or before today's date according to the handheld's system clock, the table manager displays an exclamation point (`!`) after the date. The table manager always draws a `dateTableItem` in the current font.

This display behavior should sound familiar to anyone who has used the built-in To Do application, because its due date column is composed of `dateTableItem` cells.

labelTableItem

A `labelTableItem` is simply a text label that the user cannot edit, except that instead of only displaying the string pointed to by its `ptr` field, the label table item appends a colon (`:`) to the text. The table manager draws the label in the system's default font. Selecting the label, or a text field in the same row as the label, highlights the label. Most of the field labels in the Address Book application belong to this type of table item.

numericTableItem

Numeric table items display the value stored in their `intValue` fields. The number cannot be directly edited by the user. The table manager draws the number in the system's default bold font.

popupTriggerTableItem

A `popupTriggerTableItem` allows the user to call up a pop-up list and make a selection from the list by tapping the table cell containing the pop-up trigger table item. The `popupTriggerTableItem` displays the currently selected list item and stores the index of the list selection in the `intValue` field. This table item keeps a pointer to the pop-up trigger's associated list in the table item's `ptr` field. Keep in mind that, as with an ordinary pop-up trigger object, you must provide a separate list object resource to attach to the pop-up trigger table item. The table manager draws the list and the currently selected item in the system's default font.

textTableItem

A `textTableItem` is an editable text field contained within a single table cell. The table item's `fontID` field stores the font used to display the text, and `ptr` contains a pointer to the string that contains the field's text. You must provide callback functions to load and save the text in each `textTableItem` cell.

textWithNoteTableItem

The `textWithNoteTableItem` type is identical to the `textTableItem` type, except that `textWithNoteTableItem` also has a note icon on the right side of the cell.

narrowTextTableItem

A `narrowTextTableItem` is similar to a `textTableItem`, but it has a certain amount of space reserved at the right side of the cell. The `intValue` field of the cell stores the number of pixels to set aside at the right of the cell. This space is useful for displaying small icons, such as the repeat and alarm indicators used in the built-in Date Book application. Along with the callbacks for loading and saving the text contents of the cell, a `narrowTextTableItem` should also have a callback function to draw the icons. This drawing callback is similar to the callback function used for a `customTableItem`.

Initializing a Table

Before you can implement user interaction with a table, or even use the table to display data, you must prepare the table for use. Initializing a table primarily involves telling the table manager what item type each cell of the table should be, along with setting up callback functions for certain columns to perform custom drawing routines, or to save and retrieve text from fields in the table.

You need to initialize a table before the system draws it to the screen. The best time to perform this initialization is when handling a form's `frmOpenEvent`. In Table Example, the main form's event handler, **MainFormHandleEvent**, delegates initialization of the table to **MainFormInit**:

```
static Boolean MainFormHandleEvent(EventType *event)
{
    Boolean   handled = false;
    FormType  *form;

    switch (event->eType) {
        case frmOpenEvent:
            form = FrmGetActiveForm();
            MainFormInit(form);
            FrmDrawForm(form);
            handled = true;

        // Other event handling omitted.

        default:
            break;

    }

    return handled;
}
```

The **MainFormInit** function itself is shown in Listing 11-1.

Listing 11-1: **MainFormInit**

```
static void MainFormInit(FormType *form)
{
    TableType  *table;
    Int16     numRows;
    Int16     i;
    DateType  dates[11], today;
    UInt32    now;
    UInt32    curDate;
    ListType  *list;
```

```
// Initialize the dates. The first date in the table is
// set to the constant noTime so it will display as a
// hyphen. The rest of the dates will range from four days
// ago to five days ahead of the current date on the
// handheld.
* ((Int16 *) &dates[0]) = noTime;
DateSecondsToDate(TimGetSeconds(), &today);
now = DateToDays(today);
for (i = 1; i < sizeof(dates) / sizeof(*dates); i++) {
    curDate = now - 5 + i;
    DateDaysToDate(curDate, &(dates[i]));
}

table = FrmGetObjectPtr(form, FrmGetObjectIndex(form,
                        MainTable));
list = FrmGetObjectPtr(form, FrmGetObjectIndex(form,
                       MainList));

// Set item types and values.
numRows = TblGetNumberOfRows(table);
for (i = 0; i < numRows; i++) {
    TblSetItemStyle(table, i, 0, labelTableItem);
    TblSetItemPtr(table, i, 0, gLabels[i]);

    TblSetItemStyle(table, i, 1, dateTableItem);
    TblSetItemInt(table, i, 1, DateToInt(dates[i]));

    TblSetItemStyle(table, i, 2, numericTableItem);
    TblSetItemInt(table, i, 2, i);

    TblSetItemStyle(table, i, 3, textTableItem);

    TblSetItemStyle(table, i, 4, checkboxTableItem);
    TblSetItemInt(table, i, 4, i % 2);

    TblSetItemStyle(table, i, 5, narrowTextTableItem);
    TblSetItemInt(table, i, 5, ((i % 3) * 7) + 6);

    TblSetItemStyle(table, i, 6, popupTriggerTableItem);
    TblSetItemInt(table, i, 6, i % 3);
    TblSetItemPtr(table, i, 6, list);

    TblSetItemStyle(table, i, 7, textWithNoteTableItem);

    TblSetItemStyle(table, i, 8, customTableItem);
    TblSetItemInt(table, i, 8, i % 3);
}

// Set columns usable and adjust column spacing.
for (i = 0; i < numTableColumns; i++) {
    TblSetColumnUsable(table, i, true);
```

Continued

Listing 11-1 *(continued)*

```
        switch (i) {
            case 2:
                TblSetColumnSpacing(table, i, 2);
                break;

            default:
                TblSetColumnSpacing(table, i, 0);
                break;
        }
    }

    // Set callback functions for loading, saving, and drawing.
    TblSetLoadDataProcedure(table, 3, LoadTextTableItem);
    TblSetLoadDataProcedure(table, 5, LoadTextTableItem);
    TblSetLoadDataProcedure(table, 7, LoadTextTableItem);

    TblSetSaveDataProcedure(table, 3, SaveTextTableItem);

    TblSetCustomDrawProcedure(table, 5,
                              DrawNarrowTextTableItem);
    TblSetCustomDrawProcedure(table, 8, DrawCustomTableItem);

    // Draw the form.
    FrmDrawForm(form);
}
```

Before we delve into **MainFormInit**, it would be useful to take a look at some of the constants and global variables used in Table Example, because these items furnish the values for some of the labels and text fields in the table. The following global variables are from the top of `table.c`:

```
// Table constants
#define numTextColumns   3
#define numTableColumns  9
#define numTableRows     11

// Global variables
static Char *  gLabels[] = {"00", "01", "02", "03", "04", "05",
                            "06", "07", "08", "09", "10"};
MemHandle  gTextHandles[numTextColumns][numTableRows];
Boolean    gRowsHidden = false;
Boolean    gColumnsHidden = false;
```

The constants `numTableColumns` and `numTableRows` are fairly self-explanatory, simply declaring how many columns and rows the table has; `numTextColumns` states the number of columns that contain a text field item. Labels for each row in the table are stored in the `gLabels` array, which Table Example initializes in the `gLabels` variable declaration. The `gTextHandles` two-dimensional array of memory handles stores the handles to each of the text fields in the table. Table Example uses `gRowsHidden` and `gColumnsHidden` to keep track of whether the application is currently hiding any rows or columns in the table.

Before **MainFormInit** can use the `gTextHandles` array, the application must initialize the array. Table Example accomplishes this in its **StartApplication** routine:

```
static Err StartApplication(void)
{
    Int16  i, j;


    for (i = 0; i < numTextColumns; i++) {
        for (j = 0; j < numTableRows; j++) {
            Char  *str;

            gTextHandles[i][j] = MemHandleNew(1);
            str = MemHandleLock(gTextHandles[i][j]);
            *str = '\0';
            MemHandleUnlock(gTextHandles[i][j]);
        }
    }

    return false;
}
```

The **StartApplication** function iterates over the `gTextHandles` array, allocating a new memory handle for each array element and filling the handle's contents with a single trailing null.

Now that all the global variables that Table Example requires have been readied, the program can get to the work of initializing the table with the **MainFormInit** function, shown in Listing 11-1.

The first part of **MainFormInit** simply fills `dates`, an array of `DateType` structures, with a few dates so the table's second column will have data to display:

```
// Initialize the dates. The first date in the table is
// set to the constant noTime so it will display as a
// hyphen. The rest of the dates will range from four days
// ago to five days ahead of the current date on the
// handheld.
```

```
* ((Int16 *) &dates[0]) = noTime;
DateSecondsToDate(TimGetSeconds(), &today);
now = DateToDays(today);
for (i = 1; i < sizeof(dates) / sizeof(*dates); i++) {
    curDate = now - 5 + i;
    DateDaysToDate(curDate, &(dates[i]));
}
```

After **MainFormInit** fills in the `dates` array, the real work of initializing the table begins. The first task is to iterate over the rows of the table and set the table item type for each cell.

Setting item types

Use the **TblSetItemStyle** function to set the item type for a particular cell column. The **TblSetItemStyle** function takes four arguments: a pointer to a table, the row of the cell to set, the column of the cell to set, and a `TableItemStyleType` value. It is possible to set cells within a column to different item types, though it is more common to make all of a column's cells share the same type.

In particular, `textTableItem`, `textWithNoteTableItem`, and `narrowTextTableItem` types do not play well with other data types, because the functions for setting load and save callback functions for text type table items, TblSetLoadDataProcedure and TblSetSaveDataProcedure, allow you to specify only an entire column. Setting a text loading or saving callback function for a cell that does not have a text field in it will cause your application to crash.

Because **MainFormInit** is iterating over the table's rows with a `for` loop, this is a handy time to set the `intValue` and `ptr` fields for each cell that requires these values. The **TblSetItemInt** and **TblSetItemPtr** functions accomplish this task in **MainFormInit**.

The first three columns of the table contain data types that the user cannot directly alter. Column one is a simple label. **MainFormInit** fills in the text for the labels from the global `gLabels` array:

```
TblSetItemStyle(table, i, 0, labelTableItem);
TblSetItemPtr(table, i, 0, gLabels[i]);
```

Values for the second column were set earlier in **MainFormInit** in the `dates` array. After **MainFormInit** sets the second column's type to `dateTableItem`, it fills in the date for each date cell with **TblSetItemInt**:

```
TblSetItemStyle(table, i, 1, dateTableItem);
TblSetItemInt(table, i, 1, DateToInt(dates[i]));
```

The DateToInt "function" used here is not documented anywhere in the Palm OS SDK Reference. Instead, DateToInt is a macro defined in the Palm OS header file `DateTime.h` as follows:

```
#define DateToInt(date) (*(UInt16 *) &date)
```

Because the `DateType` structure is exactly sixteen bits long, the DateToInt macro is very useful for shoehorning a `DateType` into a normal sixteen-bit integer value, such as the `intValue` field of a table item.

The third column, last of the columns that are not user-editable, is a `numericTableItem`. **MainFormInit** simply tosses the index of the current row into the `intValue` for each item in this column:

```
TblSetItemStyle(table, i, 2, numericTableItem);
TblSetItemInt(table, i, 2, i);
```

The table's fourth column is a `textTableItem`. Because a text table item's `intValue` field is unused and its `ptr` field is set via a callback function later in **MainFormInit**, the only thing initially set in this `for` loop is the cell's type:

```
TblSetItemStyle(table, i, 3, textTableItem);
```

The fifth column is a check box. After **MainFormInit** sets the cell's item type to `checkboxTableItem`, it sets every other check box in the column to be checked:

```
TblSetItemStyle(table, i, 4, checkboxTableItem);
TblSetItemInt(table, i, 4, i % 2);
```

After the check box comes the sixth column, which contains another type of text field: a `narrowTextTableItem`. A narrow text item's `intValue` indicates the amount of space to reserve at the right side of the text field for custom drawing. In Table Example, the space is reserved for drawing the alarm and repeat icons from the built-in Address Book application. Both of these icons are seven pixels wide apiece, and the `for` loop initializes space for neither, one, or both icons, depending on the row's index number:

```
TblSetItemStyle(table, i, 5, narrowTextTableItem);
TblSetItemInt(table, i, 5, ((i % 3) * 7) + 6);
```

The extra six pixels added to the space are a fudge factor to ensure that the icons at the right of the field are not cut off by the next column's pop-up triggers. Pop-up triggers are ill-behaved when it comes to staying within their allotted space in a table, and because they are right-justified, they will gladly overlap anything to their left if the text displayed in the trigger is too long to fit, as is the case in Table Example.

The seventh column contains pop-up triggers. Besides setting the item type for this column, **MainFormInit** also sets the `intValue` field to `0`, `1`, or `2`, depending on the table row, and sets the `ptr` field to point to `MainList`, a pop-up list containing three items. The `intValue` determines which list element from `MainList` is currently selected, and therefore drawn as the trigger's label. Note that a `popupTrigger TableItem` appends a colon (`:`) to the end of the pop-up trigger label, but this colon does not display in the pop-up list itself.

```
TblSetItemStyle(table, i, 6, popupTriggerTableItem);
TblSetItemInt(table, i, 6, i % 3);
TblSetItemPtr(table, i, 6, list);
```

Occupying the eighth column is the last type of table text field, a `textWithNote TableItem`. Like the `textTableItem` in column four, the item type is the only thing that needs to be set in this `for` loop:

```
TblSetItemStyle(table, i, 7, textWithNoteTableItem);
```

The ninth and last column contains a `customTableItem`. In the Table Example program, the custom widget displayed in this column's cells displays nothing, the alarm icon, or the repeat icon, depending on whether the value of the cell's `intValue` field is `0`, `1`, or `2`, respectively. Tapping in the last column causes the cell to cycle to the next icon. The `for` loop sets up the initial value for each of this column's cells, but the code that actually handles drawing and responding to taps is elsewhere in the application:

```
TblSetItemStyle(table, i, 8, customTableItem);
TblSetItemInt(table, i, 8, i % 3);
```

Setting static row height

In the ROM applications, tables that contain text fields automatically expand and contract the height of the row containing a text field when the user enters more text than will fit in the field. As a matter of fact, expansion and contraction of table fields is their default behavior, and the table manager handles adjustments to the height of the row automatically. Unfortunately, the table manager does only half the work required. The system will resize a row without help from your application, but if expanding the field shoves other table rows off the bottom of the table, and then the user deletes enough text to allow other rows to have space again, the table manager does not redraw the rows, resulting in a large blank space in the bottom part of the table.

Properly implementing expanding text fields in a table requires a fair amount of complex scrolling code in your application. Because Table Example is supposed to be a simple example without any scrolling, preventing rows from automatically resizing as text is added to their fields is necessary. Fortunately, the Palm OS

provides the **TblSetRowStaticHeight** function, which takes a pointer to a table, the row in the table to set, and a `Boolean` value indicating whether the row's height should be unchangeable (`true`) or resizable (`false`). The **MainFormInit** function in Table Example calls **TblSetRowStaticHeight** as its last action while iterating over each row in the table:

```
TblSetRowStaticHeight(table, i, true);
```

Setting column usability and spacing

After iterating over the rows of the table, **MainFormInit** needs to iterate over the columns to further set up the table:

```
for (i = 0; i < numTableColumns; i++) {
    TblSetColumnUsable(table, i, true);
    switch (i) {
        case 2:
            TblSetColumnSpacing(table, i, 2);
            break;

        default:
            TblSetColumnSpacing(table, i, 0);
            break;
    }
}
```

By default, table columns are not usable and, hence, not drawn by the table manager. To make each column visible, **MainFormInit** calls the **TblSetColumnUsable** function. A companion function, **TblSetRowUsable**, also exists to set the usability (and visibility) of table rows, but because rows default to usable, it is not necessary to call **TblSetRowUsable** in the table initialization.

After setting column usability, **MainFormInit** sets the spacing between columns. Without any intervention from your application code, each column in a cell automatically has a single space following it to separate it from the next column. Because the Table Example program is pressed for available screen space, **Main FormInit** uses **TblSetColumnSpacing** to set most of the columns to have no trailing space at all.

In a real application, text fields are easier to use with a bit of leading space before them; MainFormInit sets the spacing in the third column (column index `2`) to two pixels to make the `textTableItem` in column four easier to read. Compare column four with columns six and eight, which have no leading space, to see the difference that leaving space in front of a text field can make.

Setting custom load routines

The **MainFormInit** function, having finished with formatting and loading data into the table's cells, must now turn its attention to setting up callback functions for retrieving and saving the text in each of the table's text fields. First, **MainFormInit** sets the callbacks for loading data:

```
TblSetLoadDataProcedure(table, 3, LoadTextTableItem);
TblSetLoadDataProcedure(table, 5, LoadTextTableItem);
TblSetLoadDataProcedure(table, 7, LoadTextTableItem);
```

Setting a table item's data loading callback function requires the use of **TblSetLoad DataProcedure**. As parameters, **TblSetLoadProcedure** takes a pointer to the table, the index of the column that will use the callback function to load its data, and a pointer to a function of type `TableLoadDataFuncType`. The prototype for `Table LoadDataFuncType` looks like this:

```
Err TableLoadDataFuncType (void *tableP, Int16 row,
    Int16 column, Boolean editable, MemHandle *dataH,
    Int16 *dataOffset, Int16 *dataSize, FieldType *fld)
```

In `TableLoadDataFuncType`, `tableP` is a pointer to a table, and `row` and `column` indicate the row and column of the cell in `tableP` that should be loaded. If the system passes a value of `true` for the `editable` parameter, a text cell somewhere in the table is currently being edited; if `editable` is `false`, the table is merely being drawn, not edited. The `dataH` parameter is a pointer to a handle, which your application should fill with the unlocked handle of a block of memory containing a null-terminated string. You need to set `dataOffset` to the offset within `dataH`, in bytes, where the string data begins.

The `dataOffset` parameter allows you to store string data for table use in a memory structure other than a simple string. For example, consider the following structure:

```
typedef struct {
    Int16  someValue;
    Char   *string;
} MyDataType;
```

The `MyDataType` structure contains an integer value before its string data, but you can still pass a handle to memory containing this structure in the `dataH` parameter of `TableLoadDataFuncType` if you also pass the value `16` in the `dataOff set` parameter, to indicate that the string data begins sixteen bytes into the structure.

Your application should set `dataSize` to the allocated size of the text string in bytes; be sure not to set `dataSize` to the length of the string, because this may be different from its memory size. Finally, the `fld` parameter contains a pointer to the field in the cell that should be loaded.

Table Example's implementation of `TableLoadDataFuncType` is **LoadTextTable Item**, which looks like this:

```
static Err LoadTextTableItem (void *table, Int16 row,
    Int16 column, Boolean editable, MemHandle *dataH,
    Int16 *dataOffset, Int16 *dataSize, FieldType *field)
{
    *dataH = gTextHandles[GetTextColumn(column)][row];
    *dataOffset = 0;
    *dataSize = MemHandleSize(*dataH);

    return 0;
}
```

The **LoadTextTableItem** is very simple, merely retrieving the cell's text from the previously initialized `gTextHandles` array and passing it to the table manager. The **GetTextColumn** function called in **LoadTextTableItem** is a helper function that maps table column indices to the indices of the first dimension in the `gTextHandles` array. Here is what **GetTextColumn** looks like:

```
static Int16 GetTextColumn(Int16 column)
{
    Int16  result;


    switch (column) {
        case 3:
            result = 0;
            break;

        case 5:
            result = 1;
            break;

        case 7:
            result = 2;
            break;

        default:
            ErrFatalDisplay("Invalid text column");
            break;
    }

    return result;
}
```

Because the data stored in the handles in `gTextHandles` is composed of the desired strings only, **LoadTextTableItem** sets `dataOffset` to `0` to indicate the start of the data in the handle. Then **LoadTextTableItem** calls the Palm OS function

MemHandleSize to retrieve the amount of memory occupied by the string and passes this value back via the `dataSize` parameter.

Setting custom save routines

Strictly speaking, a custom save routine is not necessary if your application needs to save only the text entered in each table field verbatim. If you want to perform some processing on the text before saving it, you should also set up a callback function to customize the data saving behavior with **TblSetSaveDataProcedure**:

```
TblSetSaveDataProcedure(table, 3, SaveTextTableItem);
```

The **TblSetSaveDataProcedure** function takes three parameters: a pointer to a table, the index of a column in that table whose save behavior should be modified, and a callback function of type `TableSaveDataFuncType` to perform the saving. The `TableSaveDataFuncType` callback type is simpler than `TableLoadDataFuncType`, and its prototype looks like this:

```
Boolean TableSaveDataFuncType (void *tableP, Int16 row,
                               Int16 column);
```

In `TableSaveDataFuncType`, the `tableP` parameter is a pointer to a table object, and `row` and `column` contain the row and column of the cell in that table whose data should be processed before saving. A function implementing `TableSaveDataFuncType` should return `true` if the callback function changed the text in the field, or it should return `false` if the function left the text alone. The implementation of `TableSaveDataFuncType` in Table Example is **SaveTextTableItem**:

```
static Boolean SaveTextTableItem(void *table, Int16 row,
                                 Int16 column)
{
    Boolean    result = false;
    FieldType  *field;
    MemHandle  textH;
    Char       *str;
    Int16      i;


    field = TblGetCurrentField(table);

    // If the field has been changed, uppercase its text.
    if (field && FldDirty(field)) {
        textH = gTextHandles[GetTextColumn(column)][row];
        str = MemHandleLock(textH);
        for (i = 0; str[i] != '\0'; i++) {
            if (str[i] >= 'a' && str[i] <= 'z') {
                str[i] -= 'a' - 'A';
            }
        }
```

```
            MemHandleUnlock(textH);
            TblMarkRowInvalid(table, row);
            result = true;
        }

        return result;
    }
```

The **SaveTextTableItem** function first uses the **TblGetCurrentField** function to retrieve a pointer to the field the user is currently editing. If no field currently has the focus in the table, **TblGetCurrentField** returns `NULL`, which is why the next `if` statement first checks to see if `field` has a value. The `if` also uses **FldDirty** to see if the field has been changed at all; if the field hasn't been changed, there is no need to run the rest of the code in **SaveTextTableItem**.

If the user changes the contents of the field, **SaveTextTableItem** retrieves and locks a handle to the field's text, and then converts the characters in the field to uppercase letters. Then **SaveTextTableItem** unlocks the handle and marks the row invalid with **TblMarkRowInvalid**. The call to **TblMarkRowInvalid** is important, as it forces the table manager to redraw the row and display the changes that **SaveTextTableItem** made to the field's text. Finally, **SaveTextTableItem** sets the return value of the function to `true` to indicate to the table manager that the text has been changed by the callback function.

Setting custom drawing routines

Table Example uses two custom drawing routines, one for adding icons to the ends of the `narrowTextTableItem` cells in the table's sixth column, and one for drawing the `customTableItem` in the ninth column. The **MainFormInit** function sets the custom drawing callback functions using **TblSetCustomDrawProcedure**:

```
TblSetCustomDrawProcedure(table, 5, DrawNarrowTextTableItem);
TblSetCustomDrawProcedure(table, 8, DrawCustomTableItem);
```

The **TblSetCustomDrawProcedure** function takes three parameters: a pointer to a table, the column in that table that should have custom drawing behavior, and a pointer to a `TableDrawItemFuncType` function. The `TableDrawItemFuncType` prototype looks like this:

```
void TableDrawItemFuncType (void *tableP, Int16 row,
    Int16 column, RectangleType *bounds)
```

In `TableDrawItemFuncType`, `tableP` is a pointer to a table, `row` and `column` are the row and column of the table cell to draw, and `bounds` is a pointer to a rectangle that defines the boundaries of the table cell.

Table Example implements two versions of `TableDrawItemFuncType`. The first, **DrawNarrowTextTableItem**, draws the icons on the end of the `narrowTextTableItem` in column six:

```
static void DrawNarrowTextTableItem(void *table, Int16 row,
    Int16 column, RectangleType *bounds)
{
    Char    symbol[3];
    Int16   i;
    Int16   length = 0;


    for (i = 0; i < 3; i++)
        symbol[i] = '\0';

    switch(TblGetItemInt(table, row, column)) {
        case 13:
            symbol[0] = symbolAlarm;
            length = 1;
            break;

        case 20:
            symbol[0] = symbolAlarm;
            symbol[1] = symbolRepeat;
            length = 2;
            break;

        default:
            break;
    }

    if (symbol[0] != '\0') {
        FontID  curFont = FntSetFont(symbolFont);
        Coord   x;

        x = (bounds->topLeft.x + bounds->extent.x) -
            ((length * 7) + 6);
        WinDrawChars(&symbol[0], length, x, bounds->topLeft.y);
        FntSetFont(curFont);
    }
}
```

The **DrawNarrowTextTableItem** function starts by setting up a three-character-long string (`symbol`) and initializing its characters to trailing nulls. Then the callback function uses the **TblGetItemInt** function to retrieve the `intValue` stored in this table cell. As you will recall from earlier in this chapter, the `intValue` in a `narrowTextTableItem` represents the number of pixels to reserve at the right of the cell. The **MainFormInit** function sets up space for `0`, `1`, or `2` icons, each of which

is seven pixels wide, with six pixels of padding to allow space for the pop-up trigger in the next column. As a result, the value stored in `intValue` will be `0`, `13` (`7 × 1 + 6`), or `20` (`7 × 2 + 6`), which form the comparison values for the `switch` statement in **DrawNarrowTextTableItem**.

Depending on the number of icons that should be drawn, **DrawNarrowTextTable Item** fills the first part of the `symbol` string with `symbolAlarm` and `symbolRepeat` characters from the Palm OS symbol font, as appropriate, and then draws `symbol` at the correct screen location, using the rectangle pointed to by `bounds` as a guide.

The **DrawCustomTableItem** function in Table Example requires much less math to accomplish its simple goals. Depending on the value stored in the custom table cell's `intValue` field, **DrawCustomTableItem** draws nothing, an alarm icon, or a repeat icon:

```
static void DrawCustomTableItem(void *table, Int16 row,
    Int16 column, RectangleType *bounds)
{
    FontID  curFont;
    Char    symbol[2];
    Int16   i;


    for (i = 0; i < 2; i++)
        symbol[i] = '\0';

    switch(TblGetItemInt(table, row, column)) {
        case 1:
            symbol[0] = symbolAlarm;
            break;

        case 2:
            symbol[0] = symbolRepeat;
            break;

        default:
            break;
    }

    if (symbol[0] != '\0') {
        curFont = FntSetFont(symbolFont);
        WinDrawChars(&symbol[0], 1, bounds->topLeft.x + 1,
                     bounds->topLeft.y);
        FntSetFont(curFont);
    }
}
```

Handling Table Events

The tblSelectEvent provides a mechanism for reacting to taps within the bounds of a table. In a fully fledged application, a form's event handler would look for a tblSelectEvent and perform some action in response to the user's tapping a particular column and row, such as launching another form or dialog box.

Because it is only a sample program, Table Example does not require this level of interactivity. However, it does handle the tblEnterEvent to toggle the icon displayed by the customTableItem in the last column of the table. The following code from **MainFormHandleEvent** takes care of taps on cells in the last column; this technique is suitable for responding to pen events for any customTableItem.

```
case tblEnterEvent:
{
    Int16  row = event->data.tblEnter.row;
    Int16  column = event->data.tblEnter.column;

    if (column == 8) {
        TableType *table = event->data.tblEnter.pTable;
        Int16     oldValue = TblGetItemInt(table, row, column);

        TblSetItemInt(table, row, column, (oldValue + 1) % 3);
        TblMarkRowInvalid(table, row);
        TblRedrawTable(table);
        handled = true;
    }
}
break;
```

When the main form receives a tblEnterEvent, **MainFormEventHandler** checks to see if the stylus came down within the last column. If so, **MainFormEvent Handler** uses **TblSetItemInt** to cycle the tapped cell's intValue to the next number, and then marks the row invalid with **TblMarkRowInvalid** to ensure that the table manager will redraw the row. A call to **TblRedrawTable** forces the system to redraw invalid rows, causing the customTableItem cell's custom drawing routine to draw the appropriate new icon for the cell's new intValue.

Hiding Rows and Columns

Often an application with a table needs to be able to hide rows and columns from view. For example, hiding parts of the table allows for customizing the display by adding or removing columns. The built-in To Do application uses column hiding to great effect, letting the user choose how much information should be shown for each To Do item.

Hiding a column requires the **TblSetColumnUsable** function, which was introduced earlier in this chapter as part of Table Example's **MainFormInit** function. Columns are set unusable by default, so it is necessary to activate them in the initialization

of your table. After the table is up and running, you can use **TblSetColumnUsable** to toggle columns on and off. Likewise, the **TblSetRowUsable** function allows you to turn rows in the column on and off.

To demonstrate hiding and showing parts of a table, Table Example has two buttons at the bottom of its main form, `MainHideRowsButton` and `MainHideColumns Button`. The **MainFormHandleEvent** routine handles `ctlSelectEvents` from these buttons by calling the **ToggleRow** or **ToggleColumn** function, as appropriate:

```
case ctlSelectEvent:
    switch (event->data.ctlSelect.controlID) {
        case MainHideRowsButton:
            ToggleRows();
            handled = true;
            break;

        case MainHideColumnsButton:
            ToggleColumns();
            handled = true;
            break;

        default:
            break;
    }
    break;
```

Figure 11-2 shows what the Table Example program looks like when the user taps the two buttons in all their permutations. Notice that when the columns are hidden, the slightly oversized pop-up triggers overlap part of the check box column to their left. When all the columns are visible, this overlap is not apparent, because the code in **MainFormInit** and **DrawNarrowTextTableItem** compensates for the six pixels of overlap by drawing the narrow text table item's icons six pixels farther to the left.

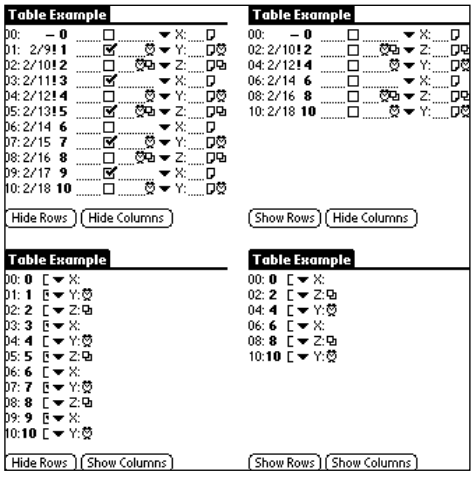


Figure 11-2: The Table Example program demonstrates hiding rows and columns. Clockwise from upper left: nothing hidden, rows hidden, both rows and columns hidden, columns hidden.

The **ToggleRows** function appears as shown in the following example:

```
static void ToggleRows(void)
{
    FormType     *form;
    TableType    *table;
    ControlType  *ctl;
    Int16        i;


    form = FrmGetActiveForm();
    table = FrmGetObjectPtr(form,
            FrmGetObjectIndex(form, MainTable));
    ctl = FrmGetObjectPtr(form,
          FrmGetObjectIndex(form, MainHideRowsButton));

    for (i = 0; i < numTableRows; i++) {
        if (i % 2)
            TblSetRowUsable(table, i , gRowsHidden);
        TblMarkRowInvalid(table, i);
    }

    if (gRowsHidden)
        CtlSetLabel(ctl, "Hide Rows");
    else
        CtlSetLabel(ctl, "Show Rows");

    TblRedrawTable(table);
    gRowsHidden = !gRowsHidden;
}
```

The **ToggleRows** function iterates over the rows of the table, hiding or showing every other row with **TblSetRowUsable**, depending on the value of `gRowsHidden`. Also, **TblSetRowUsable** marks every row in the table invalid with **TblMarkRowInvalid**; in order to hide or show rows, the entire table needs to be redrawn. After invalidating rows, **ToggleRows** changes the text of the `MainHideRowsButton` to an appropriate caption, redraws the table with **TblRedrawTable**, and toggles the value of `gRowsHidden`.

Table Example's **ToggleColumns** function is almost identical to that of **ToggleRows**:

```
static void ToggleColumns(void)
{
    FormType     *form;
    TableType    *table;
    ControlType  *ctl;
    Int16        i;
```

```
    form = FrmGetActiveForm();
    table = FrmGetObjectPtr(form,
            FrmGetObjectIndex(form, MainTable));
    ctl = FrmGetObjectPtr(form,
          FrmGetObjectIndex(form, MainHideColumnsButton));

    for (i = 0; i < numTableColumns; i++) {
        if (i % 2)
            TblSetColumnUsable(table, i , gColumnsHidden);
    }

    for (i = 0; i < numTableRows; i++)
        TblMarkRowInvalid(table, i);

    if (gColumnsHidden)
        CtlSetLabel(ctl, "Hide Columns");
    else
        CtlSetLabel(ctl, "Show Columns");

    TblRedrawTable(table);
    gColumnsHidden = !gColumnsHidden;
}
```

Notice that **ToggleColumns** contains an extra `for` loop to invalidate table rows. This step is necessary because the first `for` loop iterates over the table's columns, not its rows.

Creating More Complex Tables

Now that you have been introduced to the basics of initializing and interacting with tables, it is time to look at a more complicated example. The Librarian sample application has two forms that contain tables: the List view and the Edit view, pictured in Figure 11-3. The List view's table is mostly for display purposes, and it contains a variable number of columns depending on what kind of status information the user wishes to display in the view. Status information may be altered by tapping a status cell, which pops up a list for selection of a new status value. The Edit view's table has only two columns: a set of row labels and a series of variable-height text fields.

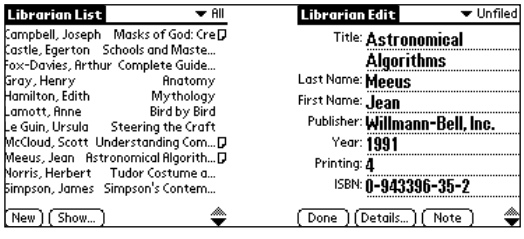


Figure 11-3: Librarian's List (left) and Edit (right) views

The biggest difference between these tables and the table in Table Example is that Librarian's tables are scrollable, allowing them to provide access to more information than will fit on a single screen. Before getting too far into the intricacies of scrolling tables, though, this chapter will look at how Librarian initializes its tables.

Connecting a Table to Data

This section will use Librarian's List view table to demonstrate one of the most common uses for tables, which is to display information from an application's database. Each row in the table represents a single record from Librarian's database. The List view table has the following PilRC resource definition:

```
TABLE ID ListTable AT (0 16 160 121) ROWS 11 COLUMNS 6
    COLUMNWIDTHS 113 10 10 10 10 6
```

The six columns in the List view table, from left to right, are used to display a record's title, book status, print status, format, read or unread status, and a note indicator to let the user know that a particular record has an attached note. Librarian uses a number of constants to identify these columns, defined in `librarian.h` as follows:

```
// List view constants
#define titleColumn        0
#define bookStatusColumn   1
#define printStatusColumn  2
#define formatColumn       3
#define readColumn         4
#define noteColumn         5
```

Initializing the List view table

To initialize this table, the form handler for the List view, **ListFormHandleEvent**, calls **ListFormInit** in response to a `frmOpenEvent`. Listing 11-2 shows the **ListFormInit** function.

Listing 11-2: Librarian's ListFormInit function

```
static void ListFormInit(FormType *form)
{
    UInt16      row;
    UInt16      rowsInTable;
    TableType   *table;
    ControlType *ctl;
    Int16       statusWidth;
    FontID      curFont;
    char        noteChar;
    Boolean     statusExists;
```

```
curFont = FntSetFont(gListFont);

// Initialize the book list table.
table = FrmGetObjectPtr(form,
        FrmGetObjectIndex(form, ListTable));
rowsInTable = TblGetNumberOfRows(table);
for (row = 0; row < rowsInTable; row++) {
    TblSetItemStyle(table, row, titleColumn,
                    customTableItem);
    TblSetItemStyle(table, row, bookStatusColumn,
                    customTableItem);
    TblSetItemStyle(table, row, printStatusColumn,
                    customTableItem);
    TblSetItemStyle(table, row, formatColumn,
                    customTableItem);
    TblSetItemStyle(table, row, readColumn,
                    customTableItem);
    TblSetItemStyle(table, row, noteColumn,
                    customTableItem);

    if (gROMVersion >=
        sysMakeROMVersion(3,0,0,sysROMStageRelease,0)) {
        TblSetItemFont(table, row, titleColumn, gListFont);
        TblSetItemFont(table, row, bookStatusColumn,
                       gListFont);
        TblSetItemFont(table, row, printStatusColumn,
                       gListFont);
        TblSetItemFont(table, row, formatColumn,
                       gListFont);
        TblSetItemFont(table, row, readColumn, gListFont);
    }

    TblSetRowUsable(table, row, false);
}

TblSetColumnUsable(table, titleColumn, true);
TblSetColumnUsable(table, bookStatusColumn,
                   gShowBookStatus);
TblSetColumnUsable(table, printStatusColumn,
                   gShowPrintStatus);
TblSetColumnUsable(table, formatColumn, gShowFormat);
TblSetColumnUsable(table, readColumn, gShowReadUnread);
TblSetColumnUsable(table, noteColumn, true);

// Set the width of the book status column.
if (gShowBookStatus) {
    TblSetColumnWidth(table, bookStatusColumn,
        FntCharWidth(libWidestBookStatusChr));
    if (gShowPrintStatus || gShowFormat || gShowReadUnread)
        TblSetColumnSpacing(table, bookStatusColumn, 0);
    else
```

Continued

Listing 11-2 *(continued)*

```
        TblSetColumnSpacing(table, bookStatusColumn, 1);
}

// Set the width of the print status column.
if (gShowPrintStatus) {
    TblSetColumnWidth(table, printStatusColumn,
        FntCharWidth(libWidestPrintStatusChr));
    if (gShowFormat || gShowReadUnread)
        TblSetColumnSpacing(table, printStatusColumn, 0);
    else
        TblSetColumnSpacing(table, printStatusColumn, 1);
}

// Set the width of the format column.
if (gShowFormat) {
    TblSetColumnWidth(table, formatColumn,
        FntCharWidth(libWidestFormatStatusChr));
    if (gShowReadUnread)
        TblSetColumnSpacing(table, formatColumn, 0);
    else
        TblSetColumnSpacing(table, formatColumn, 1);
}

// Set the width of the read column.
if (gShowReadUnread) {
    TblSetColumnWidth(table, readColumn,
        FntCharWidth(libWidestReadUnreadChr));
    TblSetColumnSpacing(table, readColumn, 1);
}

// Set the width of the note column.
FntSetFont(symbolFont);
noteChar = symbolNote;
TblSetColumnWidth(table, noteColumn,
                  FntCharWidth(noteChar));
FntSetFont(gListFont);

statusExists = (gShowBookStatus || gShowPrintStatus ||
                gShowFormat || gShowReadUnread);

// Set the width and column spacing of the title column.
statusWidth = ((statusExists ? spaceBeforeStatus + 1 : 1) +
               (gShowBookStatus ? TblGetColumnWidth(table,
                   bookStatusColumn) : 0) +
               (gShowPrintStatus ? TblGetColumnWidth(table,
                   printStatusColumn) : 0) +
               (gShowFormat ? TblGetColumnWidth(table,
                   formatColumn) : 0) +
```

```
                    (gShowReadUnread ? TblGetColumnWidth(table,
                        readColumn) : 0) +
                    TblGetColumnWidth(table, noteColumn) + 1);

    TblSetColumnWidth(table, titleColumn,
                      table->bounds.extent.x - statusWidth);
    if (statusExists)
        TblSetColumnSpacing(table, titleColumn,
                            spaceBeforeStatus);
    else
        TblSetColumnSpacing(table, titleColumn, 1);

    // Set the callback routine that will draw the records.
    TblSetCustomDrawProcedure(table, titleColumn,
                              ListFormDrawRecord);
    TblSetCustomDrawProcedure(table, bookStatusColumn,
                              ListFormDrawRecord);
    TblSetCustomDrawProcedure(table, printStatusColumn,
                              ListFormDrawRecord);
    TblSetCustomDrawProcedure(table, formatColumn,
                              ListFormDrawRecord);
    TblSetCustomDrawProcedure(table, readColumn,
                              ListFormDrawRecord);
    TblSetCustomDrawProcedure(table, noteColumn,
                              ListFormDrawRecord);

    // Load records into the address list.
    ListFormLoadTable();

    FntSetFont(curFont);

    // Other form initializing code omitted.
}
```

The first thing **ListFormInit** takes care of is defining what kind of data each table cell contains. Just as in the Table Example program, Librarian iterates over each row in the table with a `for` loop, but unlike Table Example, Librarian sets every single cell to the `customTableItem` type. Although the `labelTableItem` type allows for static display of text, which is primarily what the `ListTable` is for, `labelTableItem` cells tack a colon (:) onto the end of their text strings, an unwelcome side effect for the List view's table.

Within the first `for` loop, **ListFormInit** also sets the font of the four status cells if Librarian is running on Palm OS 3.0 or later:

```
    if (gROMVersion >=
        sysMakeROMVersion(3,0,0,sysROMStageRelease,0)) {
```

```
        TblSetItemFont(table, row, titleColumn, gListFont);
        TblSetItemFont(table, row, bookStatusColumn, gListFont);
        TblSetItemFont(table, row, printStatusColumn, gListFont);
        TblSetItemFont(table, row, formatColumn, gListFont);
        TblSetItemFont(table, row, readColumn, gListFont);
    }
```

Since Palm OS versions 3.0 and later have the big, legible `largeBoldFont` available for use, Librarian allows the user to customize the display font used in the List view. The user can select the standard Palm OS font to fit as much information on the screen as possible, or the larger font when readability from a distance is an issue. Calling **TblSetItemFont** for each of the table's cells to set that cell's font to `gListFont` (a Librarian global variable that keeps track of the current font for the List view) ensures that the cells are the proper height to hold their contents, whether those contents are in `stdFont`, `boldFont`, or `largeBoldFont`. Note that it is unnecessary to set the last column's font, because the note icon it contains will never be taller than the default `stdFont` height.

After setting fonts for each row's cells, **ListFormInit** makes all the rows of the table unusable:

```
    TblSetRowUsable(table, row, false);
```

Librarian makes all the rows unusable by default because there may not be enough records displayed to fill the entire screen, which would result in empty rows that still responded to user interaction. The rows become usable only if they contain data, which the **ListFormLoadTable** function (described in the next section) handles as it fills the table from Librarian's database.

Next, **ListFormInit** sets the usability of the table's columns:

```
        TblSetColumnUsable(table, titleColumn, true);
        TblSetColumnUsable(table, bookStatusColumn,
                           gShowBookStatus);
        TblSetColumnUsable(table, printStatusColumn,
                           gShowPrintStatus);
        TblSetColumnUsable(table, formatColumn, gShowFormat);
        TblSetColumnUsable(table, readColumn, gShowReadUnread);
        TblSetColumnUsable(table, noteColumn, true);
```

The `titleColumn` and `noteColumn` are always visible in the List view, but Librarian allows the user to turn the four status columns on or off. Librarian uses four global variables, `gShowBookStatus`, `gShowPrintStatus`, `gShowFormat`, and `gShowReadUnread`, to keep track of which fields should be displayed. A `true` value in any of these variables indicates that the appropriate column should show up in the List view. Figure 11-4 shows three views of the List form with varying numbers of the status columns turned on or off.

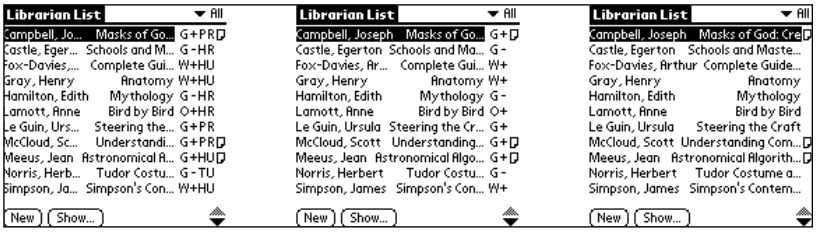


Figure 11-4: Librarian's List view, with all the status columns displayed (left), only two status columns displayed (middle), and none of the status columns displayed (right)

To make the most efficient use of the handheld's limited screen real estate, Librarian resizes the columns in the List view to allow the largest amount of space possible for the first column. To determine the size of the first column, **ListFormInit** first calculates and sets the widths of the four status columns and the note indicator column:

```
// Set the width of the book status column.
if (gShowBookStatus) {
    TblSetColumnWidth(table, bookStatusColumn,
        FntCharWidth(libWidestBookStatusChr));
    if (gShowPrintStatus || gShowFormat || gShowReadUnread)
        TblSetColumnSpacing(table, bookStatusColumn, 0);
    else
        TblSetColumnSpacing(table, bookStatusColumn, 1);
}

// Set the width of the print status column.
if (gShowPrintStatus) {
    TblSetColumnWidth(table, printStatusColumn,
        FntCharWidth(libWidestPrintStatusChr));
    if (gShowFormat || gShowReadUnread)
        TblSetColumnSpacing(table, printStatusColumn, 0);
    else
        TblSetColumnSpacing(table, printStatusColumn, 1);
}

// Set the width of the format column.
if (gShowFormat) {
    TblSetColumnWidth(table, formatColumn,
        FntCharWidth(libWidestFormatStatusChr));
    if (gShowReadUnread)
        TblSetColumnSpacing(table, formatColumn, 0);
    else
        TblSetColumnSpacing(table, formatColumn, 1);
}
```

```
    // Set the width of the read column.
    if (gShowReadUnread) {
        TblSetColumnWidth(table, readColumn,
            FntCharWidth(libWidestReadUnreadChr));
        TblSetColumnSpacing(table, readColumn, 1);
    }

    // Set the width of the note column.
    FntSetFont(symbolFont);
    noteChar = symbolNote;
    TblSetColumnWidth(table, noteColumn,
                      FntCharWidth(noteChar));
    FntSetFont(gListFont);
```

The widest character that can appear in each status column is stored in a constant value, one of `libWidestBookStatusChr`, `libWidestPrintStatusChr`, `libWidestFormatStatusChr`, or `libWidestReadUnreadChr`. Feeding the character constant to the **FntCharWidth** function determines the width in pixels of the character, which is suitable to pass to **TblSetColumnWidth** to actually set the width of each column.

To conserve screen space, the four status columns are jammed next to each other with no intervening space, but to provide a little visual separation between the status columns and the note indicator, the last visible status column should have a single pixel of separation between it and the note column. The **ListFormInit** function uses **TblSetColumnSpacing** to set the after-column spacing of each status column.

After the widths of the last five columns have been calculated, it is possible to set the width of the `titleColumn`:

```
statusExists = (gShowBookStatus || gShowPrintStatus ||
                gShowFormat || gShowReadUnread);

// Set the width and column spacing of the title column.
statusWidth = ( (statusExists ? spaceBeforeStatus + 1 : 1) +
                (gShowBookStatus ? TblGetColumnWidth(table,
                    bookStatusColumn) : 0) +
                (gShowPrintStatus ? TblGetColumnWidth(table,
                    printStatusColumn) : 0) +
                (gShowFormat ? TblGetColumnWidth(table,
                    formatColumn) : 0) +
                (gShowReadUnread ? TblGetColumnWidth(table,
                    readColumn) : 0) +
                TblGetColumnWidth(table, noteColumn) + 1);

TblSetColumnWidth(table, titleColumn, table->bounds.extent.x -
                  statusWidth);
if (statusExists)
    TblSetColumnSpacing(table, titleColumn, spaceBeforeStatus);
else
    TblSetColumnSpacing(table, titleColumn, 1);
```

The `Boolean` value `statusExists`, if `true`, indicates that at least one status column is visible in the table. Armed with this information, **ListFormInit** can fill in `statusWidth` with the width in pixels of any visible status column plus the width of the note column, using the Palm OS function **TblGetColumnWidth** to determine the widths of the appropriate columns. Then **ListFormInit** can calculate the width of `titleColumn` as the total width of the table, less `statusWidth`.

Finally, if any status columns are displayed, **ListFormInit** sets the spacing after `titleColumn` to the constant value `spaceBeforeStatus` (three pixels), which provides some visual separation between `titleColumn` and the first status column. This separation is important, because `titleColumn` and the status columns are composed of text in the same font, and without a little extra space, it is hard to tell where the title ends and the status indicators begin.

The **ListFormInit** function uses **TblSetCustomDrawProcedure** to set the same callback function for all the table's columns:

```
TblSetCustomDrawProcedure(table, titleColumn,
                          ListFormDrawRecord);
TblSetCustomDrawProcedure(table, bookStatusColumn,
                          ListFormDrawRecord);
TblSetCustomDrawProcedure(table, printStatusColumn,
                          ListFormDrawRecord);
TblSetCustomDrawProcedure(table, formatColumn,
                          ListFormDrawRecord);
TblSetCustomDrawProcedure(table, readColumn,
                          ListFormDrawRecord);
TblSetCustomDrawProcedure(table, noteColumn,
                          ListFormDrawRecord);
```

Now that the table has been initialized, all that remains is to fill it with data from Librarian's database. The **ListFormInit** function calls another of Librarian's functions, **ListFormLoadTable**, to accomplish this task.

Loading data into the table

Listing 11-3 contains the **ListFormLoadTable** function, which is responsible for filling the List form's table with data.

Listing 11-3: Librarian's ListFormLoadTable function

```
static void ListFormLoadTable(void)
{
    UInt16    row, numRows, visibleRows;
    UInt16    lineHeight;
    UInt16    recordNum;
    FontID    curFont;
    TableType  *table;
```

Continued

Listing 11-3 *(continued)*

```
table = GetObjectPtr(ListTable);

TblUnhighlightSelection(table);

// Try going forward to the last record that should be
// visible.
visibleRows = ListFormNumberOfRows(table);
recordNum = gTopVisibleRecord;
if (! SeekRecord(&recordNum, visibleRows - 1,
    dmSeekForward)) {
    // At least one line has no record. Fix it.
    // Try going backwards one page from the last record.
    gTopVisibleRecord = dmMaxRecordIndex;
    if (! SeekRecord(&gTopVisibleRecord, visibleRows - 1,
                     dmSeekBackward)) {
        // Not enough records to fill one page. Start with
        // the first record.
        gTopVisibleRecord = 0;
        SeekRecord(&gTopVisibleRecord, 0, dmSeekForward);
    }
}

curFont = FntSetFont(gListFont);
lineHeight = FntLineHeight();
FntSetFont(curFont);

recordNum = gTopVisibleRecord;

for (row = 0; row < visibleRows; row++) {
    if (! SeekRecord(&recordNum, 0, dmSeekForward))
        break;

    // Make the row usable.
    TblSetRowUsable(table, row, true);

    // Mark the row invalid so that it will draw when we
    // call the draw routine.
    TblMarkRowInvalid(table, row);

    // Store the record number as the row ID.
    TblSetRowID(table, row, recordNum);

    TblSetRowHeight(table, row, lineHeight);

    recordNum++;
}

// Hide the items that don't have any data.
```

```
    numRows = TblGetNumberOfRows (table);
    while (row < numRows) {
        TblSetRowUsable(table, row, false);
        row++;
    }

    // Update the List view's scroll buttons.
    ListFormUpdateScrollButtons();
}
```

Because **ListFormLoadTable** completely redraws the items displayed in the table, the highlighted selection becomes invalid. The first thing **ListFormLoadTable** does is to retrieve a handle to the List form's table and unhighlight the table's current selection with the **TblUnhighlightSelection** function.

The **ListFormLoadTable** function then retrieves the number of rows the table can display at once using the helper function **ListFormNumberOfRows**:

```
static UInt16 ListFormNumberOfRows(TableType *table)
{
    UInt16  rows, rowsInTable;
    UInt16  tableHeight;
    FontID  curFont;
    RectangleType  r;


    rowsInTable = TblGetNumberOfRows(table);

    TblGetBounds(table, &r);
    tableHeight = r.extent.y;

    curFont = FntSetFont(gListFont);
    rows = tableHeight / FntLineHeight();
    FntSetFont(curFont);

    if (rows <= rowsInTable)
        return (rows);
    else
        return (rowsInTable);
}
```

The **ListFormNumberOfRows** function simply compares the height of a line in the current font, determined using the **FntLineHeight** function, with the height of the table itself. If the user changes the display font Librarian uses for drawing the List form's data, the number of displayable rows changes because some fonts are taller than others. For example, the `stdFont` allows for eleven table rows, whereas the `largeBoldFont` allows space for only eight.

Once the number of visible rows is stored in the variable `visibleRows`, **ListForm LoadTable** looks for an appropriate record in the database to use as a starting point for filling in the table. The global variable `gTopVisibleRecord` stores the record ID of the record at the top of the table. If the program is just starting up, `gTopVisible Record` is equal to `0`, representing the first record in Librarian's database. In order for a record to be a good candidate for `gTopVisibleRecord`, the following things must be true:

1. There must be enough displayable records after `gTopVisibleRecord` in Librarian's database to fill the entire table, leaving no blank rows at the end.
2. Failing that, `gTopVisibleRecord` should be the first displayable record in the database.

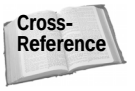
The **ListFormLoadTable** function determines the viability of `gTopVisibleRecord` with the following code:

```
// Try going forward to the last record that should be visible.
visibleRows = ListFormNumberOfRows(table);
recordNum = gTopVisibleRecord;
if (! SeekRecord(&recordNum, visibleRows - 1, dmSeekForward)) {
    // At least one line has no record. Fix it.
    // Try going backwards one page from the last record.
    gTopVisibleRecord = dmMaxRecordIndex;
    if (! SeekRecord(&gTopVisibleRecord, visibleRows - 1,
                     dmSeekBackward)) {
        // Not enough records to fill one page. Start with the
        // first record.
        gTopVisibleRecord = 0;
        SeekRecord(&gTopVisibleRecord, 0, dmSeekForward);
    }
}
```

Librarian's **SeekRecord** function is a wrapper for the Palm OS function **DmSeek RecordInCategory**, which looks for the next available record in the database that is a certain offset from the current record. The details of using **DmSeekRecordIn Category** are best left for Chapter 13, "Manipulating Records," but for the purpose of understanding how **ListFormLoadTable** works, all that is necessary is to understand what **SeekRecord** does. Here is the prototype for **SeekRecord**:

```
static Boolean SeekRecord(UInt16 *index, Int16 offset,
                          Int16 direction)
```

The `index` parameter is the index of a record in the database, `offset` is an integer value indicating how many records from `index` the search should start at, and `direction` tells **SeekRecord** whether to search forward or backward through the database.



See Chapter 13, "Manipulating Records," for more details of how to use DmSeek RecordInCategory and other database-related functions.

The first thing **ListFormLoadTable** attempts to do is search forward through the database for a record that is `visibleRows - 1` records away from the current record at the top of the list. If such a record does not exist, there will be one or more empty rows at the bottom of the table. To prevent empty rows, **ListFormLoadTable** then searches backward through the database, starting with the last record, again looking `visibleRows - 1` records away. If no record exists to satisfy that condition either, then there are not enough records in the entire database to fill the table. In that case, **ListFormLoadTable** resigns itself to using the first record in the database as the top of the table, assigning `0` to `gTopVisibleRecord`.

One more call to **SeekRecord** is necessary, though, because the first record might not be in the category Librarian is currently displaying; therefore, this last call to **SeekRecord** advances to the first displayable record in the database.

Now that **ListFormLoadTable** has determined a starting point in the database, it can get to the business of actually filling in the table:

```
curFont = FntSetFont(gListFont);
lineHeight = FntLineHeight();
FntSetFont(curFont);

recordNum = gTopVisibleRecord;

for (row = 0; row < visibleRows; row++) {
    if (! SeekRecord(&recordNum, 0, dmSeekForward))
        break;

    // Make the row usable.
    TblSetRowUsable(table, row, true);

    // Mark the row invalid so that it will draw when we
    // call the draw routine.
    TblMarkRowInvalid(table, row);

    // Store the record number as the row ID.
    TblSetRowID(table, row, recordNum);

    TblSetRowHeight(table, row, lineHeight);

    recordNum++;
}
```

For each visible row, **ListFormLoadTable** searches for the next displayable record using **SeekRecord**; if it doesn't find one, **ListFormLoadTable** is finished filling in

records and breaks out of the `for` loop. The **ListFormLoadTable** function sets each row that does have a record usable with **TblSetRowUsable**, and then marks that row invalid with **TblMarkRowInvalid** so that the table manager redraws the row when it next draws the table.

Most important, the `for` loop stores the index of the record as the row's ID number with the **TblSetRowID** function. Every row in a table has an ID value, which you may use to store whatever unsigned 16-bit integer value you wish. This ID number is an ideal place to stash the database index of the record displayed in a particular row. The Palm OS also provides the companion functions **TblGetRowID** and **TblFind RowID**. The **TblGetRowID** function simply returns the ID value assigned to a row, given the row number. If you need to retrieve the row number and all you have is the row's ID number, use the **TblFindRowID** function instead.

The `for` loop finishes by setting the height of each row with the **TblSetRowHeight** function.

After filling in the table, **ListFormLoadTable** hides with a `while` loop the rows that do not contain any data:

```
// Hide the items that don't have any data.
numRows = TblGetNumberOfRows (table);
while (row < numRows) {
    TblSetRowUsable(table, row, false);
    row++;
}
```

Finally, **ListFormLoadTable** calls Librarian's **ListFormUpdateScrollButtons** function, which makes sure that any changes to the table made by **ListFormLoadTable** are reflected in the appearance the repeating arrow buttons in the lower right corner of the List form:

```
// Update the List view's scroll buttons.
ListFormUpdateScrollButtons();
```

The details of **ListFormUpdateScrollButtons** are omitted here; see the next section's description of the Edit form's table for more information about implementing scrolling tables.

Drawing individual rows

The part of Librarian that does all the drawing work in the List form's table is the **ListFormDrawRecord** callback function, which **ListFormInit** sets up for all six columns in the table. Listing 11-4 shows the code for **ListFormDrawRecord**.

Listing 11-4: **Librarian's ListFormDrawRecord callback function**

```
static void ListFormDrawRecord (MemPtr table, Int16 row,
    Int16 column, RectangleType *bounds)
{
    UInt16  recordNum;
    Err     error;
    MemHandle  recordH;
    char    noteChar;
    FontID  curFont;
    LibDBRecordType  record;
    char    statusChr;
    Int16   x;
    UInt8   showInList;


#ifdef __GNUC__
    CALLBACK_PROLOGUE;
#endif

    curFont = FntSetFont(gListFont);

    // Get the record number that corresponds to the table item
    // to draw.
    recordNum = TblGetRowID(table, row);

    // Retrieve a locked handle to the record. Remember to
    // unlock recordH later when finished with the record.
    error = LibGetRecord(gLibDB, recordNum, &record, &recordH);
    ErrNonFatalDisplayIf((error), "Record not found");
    if (error) {
        MemHandleUnlock(recordH);
        return;
    }

    switch (column) {
        case titleColumn:
            showInList = LibGetSortOrder(gLibDB);
            DrawRecordName(&record, bounds, showInList,
                           &gNoAuthorRecordString,
                           &gNoTitleRecordString);
            break;

        case bookStatusColumn:
            switch (record.status.bookStatus) {
                case bookStatusHave:
                    statusChr = libHaveStatusChr;
                    break;
```

Continued

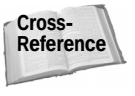
Listing 11-4 *(continued)*

```
            case bookStatusWant:
                statusChr = libWantStatusChr;
                break;

            case bookStatusOnOrder:
                statusChr = libOnOrderStatusChr;
                break;

            case bookStatusLoaned:
                statusChr = libLoanedStatusChr;
                break;

            default:
                break;
        }
        x = bounds->topLeft.x + (bounds->extent.x / 2) -
            (FntCharWidth(statusChr) / 2);
        WinDrawChars(&statusChr, 1, x, bounds->topLeft.y);
        break;

    case printStatusColumn:
        switch (record.status.printStatus) {
            case printStatusInPrint:
                statusChr = libInPrintStatusChr;
                break;

            case printStatusOutOfPrint:
                statusChr = libOutOfPrintStatusChr;
                break;

            case printStatusNotPublished:
                statusChr = libNotPublishedStatusChr;
                break;

            default:
                break;
        }
        x = bounds->topLeft.x + (bounds->extent.x / 2) -
            (FntCharWidth(statusChr) / 2);
        WinDrawChars(&statusChr, 1, x, bounds->topLeft.y);
        break;

    case formatColumn:
        switch (record.status.format) {
            case formatHardcover:
                statusChr = libHardcoverStatusChr;
                break;

            case formatPaperback:
                statusChr = libPaperbackStatusChr;
                break;
```

```
                case formatTradePaperback:
                    statusChr = libTradePaperStatusChr;
                    break;

                case formatOther:
                    statusChr = libOtherStatusChr;
                    break;

                default:
                    break;
            }
            x = bounds->topLeft.x + (bounds->extent.x / 2) -
                (FntCharWidth(statusChr) / 2);
            WinDrawChars(&statusChr, 1, x, bounds->topLeft.y);
            break;

        case readColumn:
            if (record.status.read)
                statusChr = libReadStatusChr;
            else
                statusChr = libUnreadStatusChr;
            x = bounds->topLeft.x + (bounds->extent.x / 2) -
                (FntCharWidth(statusChr) / 2);
            WinDrawChars(&statusChr, 1, x, bounds->topLeft.y);
            break;

        case noteColumn:
            // Draw a note symbol if the field has a note.
            if (record.fields[libFieldNote]) {
                FntSetFont(symbolFont);
                noteChar = symbolNote;
                WinDrawChars (&noteChar, 1, bounds->topLeft.x,
                              bounds->topLeft.y);
                FntSetFont(gListFont);
            }
            break;

        default:
            break;
    }

    // Since the handle returned from LibGetRecord (recordH) is
    // no longer needed, unlock it.
    MemHandleUnlock(recordH);

    FntSetFont(curFont);

#ifdef __GNUC__
    CALLBACK_EPILOGUE;
#endif
}
```

When called by the table manager, **ListFormDrawRecord** first determines from the row's ID number the database index of the record displayed in a particular row, using the **TblGetRowID** function. Then **ListFormDrawRecord** passes this index to Librarian's **LibGetRecord** function to retrieve the actual record from the database as follows:

```
// Get the record number that corresponds to the table item
// to draw.
recordNum = TblGetRowID(table, row);

// Retrieve a locked handle to the record. Remember to
// unlock recordH later when finished with the record.
error = LibGetRecord(gLibDB, recordNum, &record, &recordH);
ErrNonFatalDisplayIf((error), "Record not found");
if (error) {
    MemHandleUnlock(recordH);
    return;
}
```



The details of LibGetRecord are not important to this discussion; see Chapter 13, "Manipulating Records," for more information about Librarian's database-handling routines.

After retrieving the record, **ListFormDrawRecord** enters a large `switch` statement to perform different drawing tasks based on the `column` value passed to the callback function by the table manager.

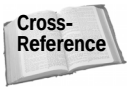
If the column that should be drawn is the `titleColumn`, **ListFormDrawRecord** calls the **LibGetSortOrder** and **DrawRecordName** helper functions to draw the appropriate record title data into the first column in the table:

```
case titleColumn:
    showInList = LibGetSortOrder(gLibDB);
    DrawRecordName(&record, bounds, showInList,
                   &gNoAuthorRecordString,
                   &gNoTitleRecordString);
    break;
```

The **LibGetSortOrder** function retrieves the current display style for the List form, which is one of three things:

- ✦ Author of the book followed by title
- ✦ Title of the book followed by author
- ✦ Title of the book only

Passing the `showInList` value retrieved by **LibGetSortOrder** to **DrawRecordName** allows the latter function to know which database fields it should pull from Librarian's database and the order in which they should be displayed. The **Draw RecordName** function performs the actual drawing to the screen, within the limits of the `bounds` parameter provided to **ListFormDrawRecord** by the table manager.



The LibGetSortOrder and DrawRecordName functions both involve a fair amount of database access, which is a topic better left to another chapter of this book. Chapter 13, "Manipulating Records," covers these Librarian functions in more detail.

If the column to be drawn is one of the status columns, **ListFormDrawRecord** looks at the database record to determine the correct character to display in the cell, and then draws that character in the appropriate place on the screen. The four status column sections of the `switch` statement and the note column section perform nearly identical operations, so only the part that handles the `bookStatusColumn` is shown in the following example for reference:

```
case bookStatusColumn:
    switch (record.status.bookStatus) {
        case bookStatusHave:
            statusChr = libHaveStatusChr;
            break;

        case bookStatusWant:
            statusChr = libWantStatusChr;
            break;

        case bookStatusOnOrder:
            statusChr = libOnOrderStatusChr;
            break;

        case bookStatusLoaned:
            statusChr = libLoanedStatusChr;
            break;

        default:
            break;
    }
    x = bounds->topLeft.x + (bounds->extent.x / 2) -
        (FntCharWidth(statusChr) / 2);
    WinDrawChars(&statusChr, 1, x, bounds->topLeft.y);
    break;
```

Based on the value of `bounds`, which is a rectangle defining the edges of the cell to draw in, **ListFormDrawRecord** centers the status character horizontally in the cell and draws it with the **WinDrawChars** function.

After drawing the status and note columns, **ListFormDrawRecord** performs some cleanup, unlocking the handle to the row's database record and setting the font back to its former state before **ListFormDrawRecord** started playing around with it:

```
// Since the handle returned from LibGetRecord (recordH) is
// no longer needed, unlock it.
MemHandleUnlock(recordH);

FntSetFont(curFont);
```

Scrolling Tables

Librarian's Edit form contains a table that is ideal for demonstrating table scrolling, because one of the form's two columns is full of text fields that expand and contract in height as the user adds text to or removes it from the fields, which is a fairly complex scenario to implement. Figure 11-5 illustrates what happens to the table as a user enters more text than will fit in a field at once. On the left, the user has entered text almost to the end of a line in a text field. On the right, the field has expanded by one line to accommodate more text, pushing the table rows underneath down a line, which causes the last line in the table to disappear entirely.

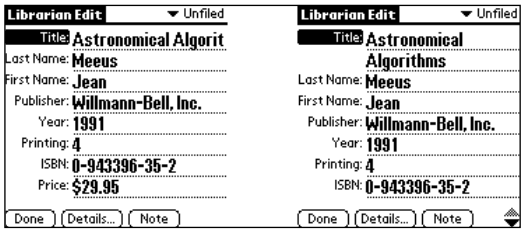


Figure 11-5: Two views of the Edit form, both before (left) and after (right) expanding a field to hold more text

Initializing a table with resizable text fields

The PilRC resource definition of the Edit form's table looks like this:

```
TABLE ID EditTable AT (0 18 160 121) ROWS 11 COLUMNS 2
    COLUMNWIDTHS 45 115
```

Librarian initializes the table in the function **EditFormInit**. The most important things that happen in **EditFormInit** are setting the table item types and setting the callback functions for loading and saving data in the text field column. The **EditFormInit** function is also responsible for setting the widths of the table's two columns as follows:

```
// Initialize the edit table.
table = FrmGetObjectPtr(form, FrmGetObjectIndex(form,
                                                EditTable));
rowsInTable = TblGetNumberOfRows(table);
for (row = 0; row < rowsInTable; row++) {
    TblSetItemStyle(table, row, labelColumn, labelTableItem);
    TblSetItemStyle(table, row, dataColumn, textTableItem);
    TblSetRowUsable(table, row, false);
}

TblSetColumnUsable(table, labelColumn, true);
TblSetColumnUsable(table, dataColumn, true);

TblSetColumnSpacing(table, labelColumn, spaceBeforeData);
```

```
// Set the callback routines that will load and save the data
// column.
TblSetLoadDataProcedure(table, dataColumn,
EditFormGetRecordField);
TblSetSaveDataProcedure(table, dataColumn,
EditFormSaveRecordField);

// Compute the width of the data column; account for the space
// between the two columns.
TblGetBounds(table, &bounds);
dataColumnWidth = bounds.extent.x - spaceBeforeData -
    gEditLabelColumnWidth;

TblSetColumnWidth(table, labelColumn, gEditLabelColumnWidth);
TblSetColumnWidth(table, dataColumn, dataColumnWidth);

EditFormLoadTable();
```

After setting up the table itself, **EditFormInit** calls **EditFormLoadTable**, which performs a similar function to the List view's **ListFormLoadTable**. The **EditForm LoadTable** function, shown in Listing 11-5, is somewhat more complex than **List FormLoadTable**, because unlike those in List view, rows in the Edit view's table can have different heights.

Listing 11-5: **Librarian's EditFormLoadTable function**

```
static void EditFormLoadTable(void)
{
    UInt16    row, numRows;
    UInt16    lineHeight;
    UInt16    fieldIndex, lastFieldIndex;
    UInt16    dataHeight;
    UInt16    tableHeight;
    UInt16    columnWidth;
    UInt16    pos, oldPos;
    UInt16    height, oldHeight;
    FontID    fontID;
    FontID    curFont;
    FormType  *form;
    TableType *table;
    Boolean   rowUsable;
    Boolean   rowsInserted = false;
    Boolean   lastItemClipped;
    RectangleType  r;
```

Continued

Listing 11-5 *(continued)*

```
LibDBRecordType  record;
MemHandle  recordH;
LibAppInfoType  *appInfo;
Boolean   fontChanged;


appInfo = MemHandleLock(LibGetAppInfo(gLibDB));

form = FrmGetActiveForm();

// Get the current record
LibGetRecord(gLibDB, gCurrentRecord, &record, &recordH);

// Get the height of the table and the width of the data
// column.
table = GetObjectPtr(EditTable);
TblGetBounds(table, &r);
tableHeight = r.extent.y;
columnWidth = TblGetColumnWidth(table, dataColumn);

// If a field is currently selected, make sure that it is
// not above the first visible field.
if (gCurrentFieldIndex != noFieldIndex) {
    if (gCurrentFieldIndex < gTopVisibleFieldIndex)
        gTopVisibleFieldIndex = gCurrentFieldIndex;
}

row = 0;
dataHeight = 0;
oldPos = pos = 0;
fieldIndex = gTopVisibleFieldIndex;
lastFieldIndex = fieldIndex;

// Load fields into the table.
while (fieldIndex <= editLastFieldIndex) {
    // Compute the height of the field's text string.
    height = EditFormGetFieldHeight(table, fieldIndex,
             columnWidth, tableHeight, &record, &fontID);

    // Is there enough room for at least one line of the
    // data?
    curFont = FntSetFont(fontID);
    lineHeight = FntLineHeight();
    FntSetFont (curFont);
    if (tableHeight >= dataHeight + lineHeight) {
        rowUsable = TblRowUsable(table, row);

        // Get the height of the current row.
```

```
        if (rowUsable)
            oldHeight = TblGetRowHeight(table, row);
        else
            oldHeight = 0;

        // If the field is not already displayed in the
        // current row, load the field into the table.
        if (gROMVersion >=
            sysMakeROMVersion(3,0,0,sysROMStageRelease,0))
            fontChanged = (TblGetItemFont(table, row,
                           dataColumn) != fontID);
        else
            fontChanged = false;

        if ((! rowUsable) ||
            (TblGetRowID(table, row) != fieldIndex) ||
            fontChanged) {
            EditInitTableRow(table, row, fieldIndex,
                             height, fontID,
                             &record, appInfo);
        }

        // If the height or the position of the item has
        // changed, draw the item.
        else if (height != oldHeight) {
            TblSetRowHeight(table, row, height);
            TblMarkRowInvalid(table, row);
        }
        else if (pos != oldPos) {
            TblMarkRowInvalid(table, row);
        }

        pos += height;
        oldPos += oldHeight;
        lastFieldIndex = fieldIndex;
        fieldIndex++;
        row++;
    }

    dataHeight += height;

    // Is the table full?
    if (dataHeight >= tableHeight) {
        // If a field is currently selected, make sure that
        // it is not below the last visible field. If the
        // currently selected field is the last visible
        // record, make sure the whole field is visible.
```

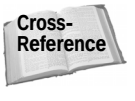
Continued

Listing 11-5 *(continued)*

```
            if (gCurrentFieldIndex == noFieldIndex)
                break;

            // Above last visible?
            else if (gCurrentFieldIndex < fieldIndex)
                break;

            // Last visible?
            else if (fieldIndex == lastFieldIndex) {
                if ((fieldIndex == gTopVisibleFieldIndex) ||
                    (dataHeight == tableHeight))
                    break;
            }

            // Remove the top item from the table and reload
            // the table again.
            gTopVisibleFieldIndex++;
            fieldIndex = gTopVisibleFieldIndex;

            row = 0;
            dataHeight = 0;
            oldPos = pos = 0;
        }
    }

    // Hide the items that don't have any data.
    numRows = TblGetNumberOfRows(table);
    while (row < numRows) {
        TblSetRowUsable(table, row, false);
        row++;
    }

    // If the table is not full and the first visible field is
    // not the first field in the record, display enough fields
    // to fill out the table by adding fields to the top of the
    // table.
    while (dataHeight < tableHeight) {
        fieldIndex = gTopVisibleFieldIndex;
        if (fieldIndex == 0) break;
        fieldIndex--;

        // Compute the height of the field.
        height = EditFormGetFieldHeight(table, fieldIndex,
            columnWidth, tableHeight, &record, &fontID);

        // If adding the item to the table will overflow the
        // height of the table, don't add the item.
        if (dataHeight + height > tableHeight)
            break;
```

```
            // Insert a row before the first row.
            TblInsertRow(table, 0);

            EditInitTableRow(table, 0, fieldIndex, height, fontID,
                             &record, appInfo);

            gTopVisibleFieldIndex = fieldIndex;
            rowsInserted = true;
            dataHeight += height;
        }

        // If rows were inserted to fill out the page, invalidate
        // the whole table; it all needs to be redrawn.
        if (rowsInserted)
            TblMarkTableInvalid(table);

        // If the height of the data in the table is greater than
        // the height of the table, then the bottom of the last row
        // is clipped and the table is scrollable.
        lastItemClipped = (dataHeight > tableHeight);

        // Update the scroll arrows.
        EditFormUpdateScrollers(form, lastFieldIndex,
                                lastItemClipped);

        MemHandleUnlock(recordH);
        MemPtrUnlock(appInfo);
    }
```

The **EditFormLoadTable** function begins by retrieving Librarian's application info block with the helper function **LibGetAppInfo** and the current record from Librarian's database with **LibGetRecord**.



See Chapter 12, "Storing and Retrieving Data," for more information about application info blocks and Librarian's LibGetAppInfo function; and Chapter 13, "Manipulating Records," for more information about retrieving records from databases and Librarian's LibGetRecord function.

After setting up a few values that will come in handy later in the function, **EditFormLoadTable** checks to see if the field that is currently selected, represented by the global variable `gCurrentFieldIndex`, is less than the first displayable field in the table, which is held in the global variable `gTopVisibleFieldIndex`. If `gCurrentFieldIndex` is less than `gTopVisibleFieldIndex`, **EditFormLoadTable** makes `gTopVisibleFieldIndex` equal the currently selected field index:

```
    // If a field is currently selected, make sure that it is not
    // above the first visible field.
    if (gCurrentFieldIndex != noFieldIndex) {
```

```
        if (gCurrentFieldIndex < gTopVisibleFieldIndex)
            gTopVisibleFieldIndex = gCurrentFieldIndex;
    }
```

The **EditFormLoadTable** then starts to load data into the table. This process is tricky, because unlike in the List view's table, there is not a one-to-one relationship between the number of data fields in each record and the number of rows available in the Edit view's table. Each field may occupy more than one row. Figure 11-6 shows the extremes involved in filling the table. The **EditFormLoadTable** function must be able to handle having fewer data fields than table rows, fewer table rows than data fields, and everything in between.

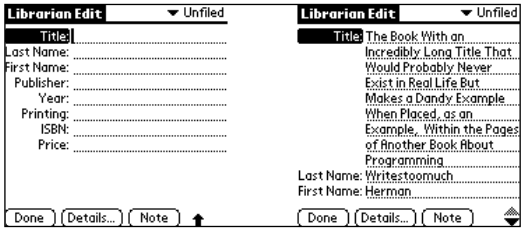


Figure 11-6: The Edit view in Librarian must handle the extremes of fewer data fields than table rows (left) and fewer table rows than data fields (right).

The logic in the massive `while` loop at the heart of **EditFormLoadTable** goes something like this:

1. Starting with the topmost visible data field, stored in `gTopVisibleFieldIndex`, iterate through the data fields until the last data field, represented by the constant `editLastFieldIndex`, has been reached.
2. Call the helper function **EditFormGetFieldHeight** to retrieve the amount of vertical space the current field would occupy on the screen and store that value in the variable `height` for later use.
3. Check to see if at least one line of the current field's data will fit within the table:

   ```
   if (tableHeight >= dataHeight + lineHeight)
   ```

   The `while` loop uses the variable `dataHeight` to keep track of the total height of the data displayed in the field, `tableHeight` is the total height available in the table itself, and `lineHeight` is the height of a single line of table text. If the first line of the current field does not fit, skip to Step 9.
4. Determine if the current table row is usable by calling **TblRowUsable**:

   ```
   rowUsable = TblRowUsable(table, row);
   ```

5. If the row is usable, find the height of the current table row with **TblGetRow Height** and store this value in the variable `oldHeight`:

   ```
   oldHeight = TblGetRowHeight(table, row);
   ```

   If the row is not usable, set `oldHeight` equal to `0`.

6. Determine whether the current table row needs to be initialized. The row needs to be initialized if any of the following criteria are met:

   - The row is not usable (`rowUsable != true`). A usable row has already been initialized.
   - The row's ID does not match the index of the current field (`TblGetRow ID(table, row) != fieldIndex`). If it does match, the current field is already drawn in the current row.
   - The font has changed (`fontChanged == true`).

   If the row requires initialization, call the helper function **EditInitTableRow** to do the job.

7. If the row's height or position has changed, mark the row invalid so it will be redrawn:

   ```
   else if (height != oldHeight) {
       TblSetRowHeight(table, row, height);
       TblMarkRowInvalid(table, row);
   }
   else if (pos != oldPos) {
       TblMarkRowInvalid(table, row);
   }
   ```

8. Increment and set values for the next pass through the `while` loop:

   ```
   pos += height;
   oldPos += oldHeight;
   lastFieldIndex = fieldIndex;
   fieldIndex++;
   row++;
   ```

9. Check to see if the table is full:

   ```
   if (dataHeight >= tableHeight)
   ```

   If the table is not full, the `while` loop repeats with the new values set in Step 8.

10. If the table is full, check to see that the currently selected field (`gCurrent FieldIndex`) appears on the screen. The selected field is visible if its index is less than the index of the current `fieldIndex` used by the `while` loop, or if the selected field is the only field displayed in the table. If the selected field is visible, break out of the `while` loop:

    ```
    if (gCurrentFieldIndex == noFieldIndex)
        break;
    ```

```
// Above last visible?
else if (gCurrentFieldIndex < fieldIndex)
    break;

// Last visible?
else if (fieldIndex == lastFieldIndex) {
    if ((fieldIndex == gTopVisibleFieldIndex) ||
        (dataHeight == tableHeight))
        break;
}
```

11. The field is full, but the selected field is not visible. Peel the top row off the table to bump all the other rows up one slot, and start at the beginning of the `while` loop again:

```
// Remove the top item from the table and reload the table
// again.
gTopVisibleFieldIndex++;
fieldIndex = gTopVisibleFieldIndex;

row = 0;
dataHeight = 0;
oldPos = pos = 0;
```

Once this `while` loop is completed, **EditFormLoadTable** hides any table rows that do not contain data:

```
numRows = TblGetNumberOfRows(table);
while (row < numRows) {
    TblSetRowUsable(table, row, false);
    row++;
}
```

At this point, you might be thinking that this function (not to mention the developer who wrote it) should take a long and well-deserved vacation, but **EditFormLoad Table** is not quite finished. After **EditFormLoadTable** has run through all the preceding code, it is still possible that the table is not quite filled and that the first field visible in the table is not the first field in the record. In this case, **EditFormLoadTable** tries to pad out the empty space in the table by adding more fields at the *top* of the table, pushing the rest of the table data down the page. Another `while` loop kicks off this whole process, set to run while the height of the displayed data is less than the height of the table itself:

```
while (dataHeight < tableHeight)
```

The new `while` loop walks backward through all the fields before the field currently displayed at the top of the table. If the height of the previous field, as returned by **EditFormGetFieldHeight**, is short enough to fit within the table, **EditFormLoadTable**

calls the Palm OS function **TblInsertRow** to put a new row at the top of the table as follows:

```
// Insert a row before the first row.
TblInsertRow(table, 0);

EditInitTableRow(table, 0, fieldIndex, height, fontID, &record,
                 appInfo);

gTopVisibleFieldIndex = fieldIndex;
rowsInserted = true;
dataHeight += height;
```

The **TblInsertRow** function does not actually increase the total number of table rows; it merely bumps all the other rows down by one, losing the last row of the table. A companion to **TblInsertRow** is **TblRemoveRow**, which also does not affect the number of rows in a table but rather moves the indicated row to the bottom of the table and marks it unusable. Neither **TblInsertRow** nor **TblRemoveRow** redraws the display, so the application must call **TblRedrawTable** to make the changes made by these row insertion and deletion functions visible.

After inserting the new row with **TblInsertRow**, **EditFormLoadTable** calls **EditInit TableRow** to set up the new row. Once the `while` loop either runs out of previous fields to try or encounters a field that would make the total data height larger than the height of the table, **EditFormLoadTable** is in the home stretch. All that remains is to mark the entire table invalid, using **TblMarkTableInvalid**, if the second `while` loop inserted any rows; update the Edit form's scroll buttons by calling **EditForm UpdateScrollers**; and unlock the record and application info block handles:

```
// If rows were inserted to fill out the page, invalidate the
// whole table, it all needs to be redrawn.
if (rowsInserted)
    TblMarkTableInvalid(table);

// If the height of the data in the table is greater than the
// height of the table, then the bottom of the last row is
// clipped and the table is scrollable.
lastItemClipped = (dataHeight > tableHeight);

// Update the scroll arrows.
EditFormUpdateScrollers(form, lastFieldIndex, lastItemClipped);

MemHandleUnlock(recordH);
MemPtrUnlock(appInfo);
```

The **EditInitTableRow** function, used in a couple places in **EditFormLoadTable** to initialize individual table rows, looks like this:

```
static void EditInitTableRow(TablePtr table, UInt16 row,
    UInt16 fieldIndex, short rowHeight, FontID fontID,
```

```
    LibDBRecordType *record, LibAppInfoType *appInfo)
{
    // Make the row usable.
    TblSetRowUsable(table, row, true);

    // Set the height of the row to the height of the data text
    // field.
    TblSetRowHeight(table, row, rowHeight);

    // Store the record number as the row ID.
    TblSetRowID(table, row, fieldIndex);

    // Mark the row invalid so that it will draw when calling
    // the draw routine.
    TblMarkRowInvalid(table, row);

    // Set the text font if Librarian is running on version 3.0
    // or later.
    if (gROMVersion >=
        sysMakeROMVersion(3,0,0,sysROMStageRelease,0))
        TblSetItemFont(table, row, dataColumn, fontID);

    // Set the labels in the label column.
    TblSetItemPtr(table, row, labelColumn,
                  appInfo->fieldLabels[fieldIndex]);
}
```

Scrolling a table

Most of the hard work required to scroll the Edit view's table has already been taken care of by the **EditFormLoadTable** function. Scrolling involves changing the top visible field index (stored in the global `gTopVisibleFieldIndex`), marking the table invalid, and then calling **EditFormLoadTable** to redraw the table in its new position. Listing 11-6 shows the **EditFormScroll** function, which determines what the new value of `gTopVisibleFieldIndex` should be, given a direction to scroll.

Listing 11-6: **Librarian's EditFormScroll function**

```
static void EditFormScroll(WinDirectionType direction)
{
    UInt16    row;
    UInt16    height;
    UInt16    fieldIndex;
    UInt16    columnWidth;
    UInt16    tableHeight;
    TableType *table;
    FontID    curFont;
    RectangleType  r;
```

```
LibDBRecordType  record;
MemHandle  recordH;


curFont = FntSetFont(stdFont);

table = GetObjectPtr(EditTable);
TblReleaseFocus(table);

// Get the height of the table and the width of the
// description column.
TblGetBounds(table, &r);
tableHeight = r.extent.y;
height = 0;
columnWidth = TblGetColumnWidth(table, dataColumn);

// Scroll the table down.
if (direction == winDown) {
    // Get the index of the last visible field; this will
    // become the index of the top visible field, unless it
    // occupies the whole screen, in which case the next
    // field will be the top field.
    row = TblGetLastUsableRow(table);
    fieldIndex = TblGetRowID(table, row);

    // If the last visible field is also the first visible
    // field, then it occupies the whole screen.
    if (row == 0)
        fieldIndex = min(editLastFieldIndex,
                         fieldIndex + 1);
}

// Scroll the table up.
else {
    // Scan the fields before the first visible field to
    // determine how many fields we need to scroll. Since
    // the heights of the fields vary, total the height of
    // the records until we get a screenful.
    fieldIndex = TblGetRowID(table, 0);
    ErrFatalDisplayIf(fieldIndex > editLastFieldIndex,
                      "Invalid field Index");
    // If we're at the top of the fields already, there is
    // no need to scroll.
    if (fieldIndex == 0) {
        FntSetFont(curFont);
        return;
    }

    // Get the current record.
    LibGetRecord(gLibDB, gCurrentRecord, &record,
                 &recordH);
```

Continued

Listing 11-6 *(continued)*

```
        height = TblGetRowHeight(table, 0);
        if (height >= tableHeight)
            height = 0;

        while (height < tableHeight && fieldIndex > 0) {
            height +=
              FldCalcFieldHeight(record.fields[fieldIndex - 1],
                                 columnWidth) * FntLineHeight();
            if ( (height <= tableHeight) ||
                 (fieldIndex == TblGetRowID(table, 0)) )
                fieldIndex--;
        }

        MemHandleUnlock(recordH);
    }

    TblMarkTableInvalid(table);
    gCurrentFieldIndex = noFieldIndex;
    gTopVisibleFieldIndex = fieldIndex;
    gEditRowIDWhichHadFocus = editFirstFieldIndex;
    gEditFieldPosition = 0;

    // Remove the highlight before reloading the table to
    // prevent the selection information from being out of
    // bounds, which can happen if the newly loaded data has
    // fewer rows than the old data.
    TblUnhighlightSelection(table);
    EditFormLoadTable();
    TblRedrawTable(table);
    FntSetFont(curFont);
}
```

When the user scrolls down, the **EditFormScroll** finds the last visible field in the table and makes it the topmost field, unless the last visible field occupies the entire screen, in which case **EditFormScroll** scrolls to the next field after the last visible one:

```
row = TblGetLastUsableRow(table);
fieldIndex = TblGetRowID(table, row);

// If the last visible field is also the first visible
// field, then it occupies the whole screen.
if (row == 0)
    fieldIndex = min(editLastFieldIndex,
                     fieldIndex + 1);
```

The **TblGetLastUsableRow** function returns the index of the last usable row in the table. From this index, **EditFormScroll** can determine which field occupies the last usable row by calling **TblGetRowID,** because the row IDs in the Edit view's table store the indices of each row's associated field.

Scrolling up is somewhat more complex, because **EditFormScroll** must iterate through the fields prior to the current top visible field, calculating their total height until a screenful has been accumulated or until there are no more prior fields to look at. The **EditFormScroll** function calls Librarian's **LibGetRecord** to retrieve the current record so it may look through the record's fields and directly calculate the height of each using the Palm OS function **FldCalcFieldHeight**.

The **FldCalcFieldHeight** function determines the number of lines long a field will be in the current font, given a pointer to the text occupying the field and the width of the field in pixels. From this information, multiplying the number of lines in the field by the height of a single line gives the total vertical space occupied by the text field:

```
height += FldCalcFieldHeight(record.fields[fieldIndex - 1],
                             columnWidth) * FntLineHeight();
```

Once **EditFormScroll** has determined what the new top visible field index is, it sets appropriate variables related to the table's new position, marks the entire table invalid, reloads the table with **EditFormLoadTable**, and redraws it with **TblRedraw Table**:

```
TblMarkTableInvalid(table);
gCurrentFieldIndex = noFieldIndex;
gTopVisibleFieldIndex = fieldIndex;
gEditRowIDWhichHadFocus = editFirstFieldIndex;
gEditFieldPosition = 0;

// Remove the highlight before reloading the table to prevent
// the selection information from being out of bounds, which
// can happen if the newly loaded data has fewer rows than the
// old data.
TblUnhighlightSelection(table);
EditFormLoadTable();
TblRedrawTable(table);
FntSetFont(curFont);
```

The Edit form's event handler, **EditFormHandleEvent**, calls **EditFormScroll** in response to a `ctlRepeatEvent` from either of the form's two repeating arrow buttons, or in response to a `keyDownEvent` containing either a `pageUpChr` or a `pageDownChr`, the two of which are generated by the hardware scroll buttons.

One more thing is required to keep scrolling running smoothly in the Edit form: the repeating scroll buttons must be visually updated to reflect the current scroll state of the form. The Palm OS function **FrmUpdateScrollers**, given the right information,

takes care of redrawing a pair of scroll arrows so they reflect a specific scroll status. Librarian contains an **EditFormUpdateScrollers** function that wraps **FrmUpdate Scrollers** and gives it the information it needs:

```
static void EditFormUpdateScrollers(FormType *form,
    UInt16 bottomFieldIndex, Boolean lastItemClipped)
{
    UInt16 upIndex;
    UInt16 downIndex;
    Boolean scrollableUp;
    Boolean scrollableDown;

    // If the first field displayed is not the first field in
    // the record, enable the up scroller.
    scrollableUp = gTopVisibleFieldIndex > 0;

    // If the last field displayed is not the last field in the
    // record, enable the down scroller.
    scrollableDown = (lastItemClipped ||
                      (bottomFieldIndex < editLastFieldIndex));

    // Update the scroll button.
    upIndex = FrmGetObjectIndex(form, EditScrollUpRepeating);
    downIndex = FrmGetObjectIndex(form,
                                  EditScrollDownRepeating);
    FrmUpdateScrollers(form, upIndex, downIndex, scrollableUp,
                       scrollableDown);
}
```

The **FrmUpdateScrollers** function takes five arguments: a pointer to a form, the index of the scroll up button, the index of the scroll down button, a `Boolean` value indicating whether it is currently possible to scroll up, and another `Boolean` to indicate the possibility of scrolling down. Depending on the values of `scrollableUp` and `scrollableDown`, **FrmUpdateScrollers** will enable, disable, or redraw the scroll buttons according to the values listed in Table 11-2.

Table 11-2
FrmUpdateScroller Function Behavior

scrollableUp	*scrollableDown*	*Result*
`true`	`true`	Both arrows drawn solid
`true`	`false`	Up arrow draw solid, down arrow drawn grayed out
`false`	`true`	Down arrow drawn solid, up arrow drawn grayed out
`false`	`false`	Both arrows disabled (not visible)

The **EditFormUpdateScrollers** function knows that the table may be scrolled up if the top visible field index is not the first field in the record. Likewise, if the last field displayed in the table is not the last field in the record, or if the bottom of the last displayed field is clipped, scrolling down is possible.

Handling Table Text Fields

When tables, already a complex user interface object, collide with text fields, another very complex object, all manner of mysterious behavior can result. Text fields contained within a table require some special handling above and beyond what a regular text field requires, and this section will attempt to unravel some of the mystery and point out a few "gotchas" you might encounter when implementing text field table items.

Understanding fields in tables

The basic underlying principle behind text fields in a table is that *there is only ever one field object associated with a table at any one time*. Only the field that is currently being edited actually exists; the table manager creates, draws, and discards field objects as needed to give the illusion of multiple fields in a table.

What effect does this amazing fact have on your Palm OS table programming? Because the table manager maintains only one active field at a time, it discards the text handle for a field as soon as it finishes drawing that field, unless the field is currently being edited. Whenever an application performs an action that releases the focus from the current field, the system calls the `TableSaveDataFuncType` callback function you have set up (using the **TblSetSaveDataProcedure** function) for the column that contains the current field.

Within the save data callback, you should copy any value contained in the current field's text handle rather than just save a pointer to the handle itself, because when the table manager destroys the field, it deallocates the field's text handle. You can retrieve a reference to the current field with the **TblGetCurrentField** function. As an example, here are the relevant portions of Librarian's **EditFormSaveRecordField** function, which **EditFormInit** attaches to the text fields in the Edit view's data column:

```
static Boolean EditFormSaveRecordField (MemPtr table,
    Int16 row, Int16 column)
{
    FieldType  *field;
    MemHandle  textH;
    Char       *text;
    Boolean    redraw = false;


#ifdef __GNUC__
    CALLBACK_PROLOGUE;
```

```
#endif

    field = TblGetCurrentField(table);
    textH = FldGetTextHandle(field);

    if (FldDirty(field)) {
        if (textH == 0)
            text = NULL;
        else {
            text = MemHandleLock(textH);
            if (text[0] == '\0')
                text = NULL;
        }

        // EditFormSaveRecordField saves the text data to
        // Librarian's database here, using the contents of the
        // text character pointer. Most of the code has been
        // omitted for clarity.
        if (text)
            MemPtrUnlock(text);
    }

    // Free the memory used for the field's text.
    FldFreeMemory(field);

#ifdef __GNUC__
    CALLBACK_PROLOGUE;
#endif

    // The code that actually sets EditFormSaveRecordField's
    // return value has been omitted.
    return redraw;
}
```

After retrieving the current field's text handle, the **EditFormSaveRecordField** function checks to see if any changes have been made to the field by calling **FldDirty**. If so, **EditFormSaveRecordField** locks the text handle, providing access to its contents via the `text` character pointer.

Once the callback has saved the data contained in `text`, **EditFormSaveRecordField** unlocks the text handle and frees the memory allocated for the field object by calling **FldFreeMemory**. This last step is important to prevent memory leaks; although the system discards the text handle, it does not deallocate it. Any handle you allocate within the save callback function must be deallocated before the end of the function.

Tables have an *edit mode*, which is activated as soon as a `tblEnterEvent` occurs within an editable text field in the table, or when your application calls the **Tbl GrabFocus** function to deliberately select a specific text cell for editing. You can

check whether a table is in edit mode by calling the **TblEditing** function, which returns `true` if a text field is currently active for editing. It is also possible to retrieve a pointer to the current field, if any, by calling **TblGetCurrentField**.

The table manager calls the save data callback function any time the current field loses the focus. This can occur when the user changes the focus by tapping in a different table cell, when the user taps a note indicator in a `textWithNoteTableItem` cell, or if your code explicitly calls **TblReleaseFocus**. Calling **TblReleaseFocus** manually is necessary any time your code redraws the table in such a way that the current field may no longer appear in the table, thus causing it to lose the focus. For example, Librarian's **EditFormScroll** function calls **TblReleaseFocus** before attempting to scroll the table.

Handling resizable text table items

The event handler for a form with a table that contains resizable text fields, such as those in Librarian's Edit view or the built-in Date Book and Address Book applications, needs to handle the `fldHeightChangedEvent` to properly deal with changes to a text field's height. Whenever the height of a field changes in response to user input, the system posts a `fldHeightChangedEvent` to the queue.

Librarian's **EditFormHandleEvent** takes care of a `fldHeightChangeEvent` by calling the **EditFormResizeData** function, shown here:

```
static void EditFormResizeData(EventType *event)
{
    UInt16     pos;
    Int16      row, column;
    UInt16     lastRow;
    UInt16     fieldIndex, lastFieldIndex, topFieldIndex;
    FieldType  *field;
    TableType  *table;
    Boolean    restoreFocus = false;
    Boolean    lastItemClipped;
    RectangleType  itemR;
    RectangleType  tableR;
    RectangleType  fieldR;


    // Get the current height of the field;
    field = event->data.fldHeightChanged.pField;
    FldGetBounds(field, &fieldR);

    // Have the table object resize the field and move the
    // items below the field up or down.
    table = GetObjectPtr(EditTable);
    TblHandleEvent(table, event);

    // If the field's height has expanded, we're done.
```

```
        if (event->data.fldHeightChanged.newHeight >=
            fieldR.extent.y) {
            topFieldIndex = TblGetRowID(table, 0);
            if (topFieldIndex != gTopVisibleFieldIndex)
                gTopVisibleFieldIndex = topFieldIndex;
            else {
                // Since the table has expanded we may be able to
                // scroll when before we might not have.
                lastRow = TblGetLastUsableRow(table);
                TblGetBounds(table, &tableR);
                TblGetItemBounds(table, lastRow, dataColumn,
                                 &itemR);
                lastItemClipped =(itemR.topLeft.y +
                    itemR.extent.y > tableR.topLeft.y +
                    tableR.extent.y);
                lastFieldIndex = TblGetRowID(table, lastRow);

                EditFormUpdateScrollers(FrmGetActiveForm(),
                    lastFieldIndex, lastItemClipped);

                return;
            }
        }

        // If the field's height has contracted and the first edit
        // field is not visible, then the table may be scrolled.
        // Release the focus, which will force saving the currently
        // edited field.
        else if (TblGetRowID (table, 0) != editFirstFieldIndex) {
            TblGetSelection(table, &row, &column);
            fieldIndex = TblGetRowID(table, row);

            field = TblGetCurrentField(table);
            pos = FldGetInsPtPosition(field);
            TblReleaseFocus(table);

            restoreFocus = true;
        }

        // Add items to the table to fill in the space made
        // available by shortening the field.
        EditFormLoadTable();
        TblRedrawTable(table);

        // Restore the insertion point position.
        if (restoreFocus) {
            TblFindRowID(table, fieldIndex, &row);
            TblGrabFocus(table, row, column);
            FldSetInsPtPosition(field, pos);
            FldGrabFocus(field);
        }
    }
```

The **EditFormResizeData** function determines if expanding or contracting a text field requires redrawing the form, which can occur when a text field shrinks, causing new rows to appear at the bottom of the table. It also checks to see if the change in size of a text field makes it possible to scroll when it was not possible before, which happens when the expansion of a text field causes the field to expand below the bottom of the table.

Enabling autoshifting in table text fields

Autoshifting in a standard text field is easy to set up. Simply define autoshifting as part of the field's properties when you create the field resource.

Unfortunately, this easy solution is not possible for text fields in a table. Because of the "virtual" nature of a table's fields, there is no way to set the autoshifting property at design time. Instead, you must enable it via application code.

Librarian's Edit table enables autoshifting in the **EditFormGetRecordField** function, which is the callback **EditFormInit** sets up for the Edit table's `dataColumn`. Within **EditFormGetRecordField** is a call to a helper function, **EditSetGraffitiMode**, which is listed in the following example:

```
static void EditSetGraffitiMode(FieldType *field)
{
    FieldAttrType  attr;


    if (field) {
        FldGetAttributes(field, &attr);
        attr.autoShift = true;
        FldSetAttributes(field, &attr);
    }
}
```

The **EditSetGraffitiMode** function manually sets a field's autoshift attribute. Because the table manager calls **EditFormGetRecordField** every time it needs to draw one of the Edit view's text fields, all the fields in the table gain the autoshift attribute when they are initialized.

Summary

In this chapter, you have learned the ins and outs of programming tables, the most complex user interface element in the Palm OS. After reading this chapter, you should know the following:

- ✦ Palm OS tables can be composed of nine different item types.
- ✦ A table must be initialized before it can be used.

- Initializing a table involves defining each cell's item type with **TblSetItemStyle**, setting initial values for nontext table items with **TblSetItemInt** and **TblSetItemPtr**, and attaching callback functions for drawing, loading, and saving with **TblSetCustomDrawProcedure**, **TblSetLoadDataProcedure**, and **TblSetSaveDataProcedure**.
- You can hide and display rows and columns of a table with the **TblSetRowUsable** and **TblSetColumnUsable** functions.
- Tables that must support scrolling require a function to load and redraw the table whenever its data changes.
- Only one text field ever exists at any one time in a table.

✦ ✦ ✦

CHAPTER 12

Storing and Retrieving Data

In This Chapter

Understanding the data manager

Creating and deleting databases

Opening and closing databases

Finding databases

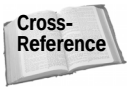
Retrieving and modifying database information

Creating an application info block

Reading and writing application and system preferences

Using feature memory

In its role as a portable data storage and display device, a handheld running the Palm OS must be able to store a variety of different data and allow the user quick access to that data. Because all data storage in the Palm OS occurs in RAM, creating and maintaining space for permanent data in the Palm OS requires an approach different from those desktop systems use to store files on non-volatile storage media. The first part of this chapter details the mechanics involved with creating, finding, and accessing Palm OS databases, the major Palm OS memory structures that contain data for long-term storage.



For information on handling individual records within a database, see Chapter 13, "Manipulating Records."

Not all data that should be saved between invocations of a particular program is appropriate for storage in a database. For example, keeping track of the last view the user visited in an application, or storing the user's preferred display font, requires saving a trivial amount of data, probably too little to go to the trouble of trying to store it in a database. For this kind of small data storage, the Palm OS offers *application preferences*, which this chapter also covers.

Finally, some special applications may require the use of another Palm OS storage technique called *feature memory*. The last part of this chapter discusses what feature memory is and how to use it.

Understanding the Data Manager

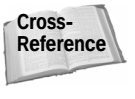
As mentioned in Chapter 2, "Understanding the Palm OS," a Palm OS *database* is simply a list of memory chunks in the

handheld's storage RAM, along with some header information that describes the database itself. In order to allow for better use of limited memory, each *record* in the database may actually be contained in any memory heap on the handheld, often mixed in with records from other application's databases. This mixture of records allows the system to shuffle data around at will, allowing records to be moved if the space they occupy is needed to store a larger record that needs to occupy a single, contiguous area of memory.

Within the database's record list, each record is listed by its *LocalID*, a memory offset from the beginning address of the card that contains the record. Storing record locations as LocalIDs instead of using pointers to the records lets the system relocate an entire memory card without needing to adjust the location of each record stored in a database's record list.

The system can relocate data at will within the handheld's storage area, so LocalIDs are valid only between the time you open a database and the time you close it again. Saving the LocalID of a database record or even of the start of a database is futile, because once your application has closed a database, the system could move the data anywhere, thereby invalidating the LocalID you have saved.

Even though the database's record list uses LocalIDs to keep track of its records, application code actually requests records by their index within the record list. The Palm OS data manager uses the index to look up the LocalID of the requested record, converts the LocalID to a handle based on the card the database header is located in, and then returns the handle to the requesting application.



More details about getting, modifying, and deleting database records are available in Chapter 13, "Manipulating Records."

Figure 12-1 shows the general layout of a database, starting with its header and record list. Notice how the actual records in the database do not need to share the same storage heap as the database header.

The database header consists of a number of fields, outlined in Table 12-1, that describe everything from the name of the database to the number of records the database contains.

The format of the database header may change in future versions of the Palm OS. When working with database structures, never assume that the database has a specific format; always use the Palm OS database functions.

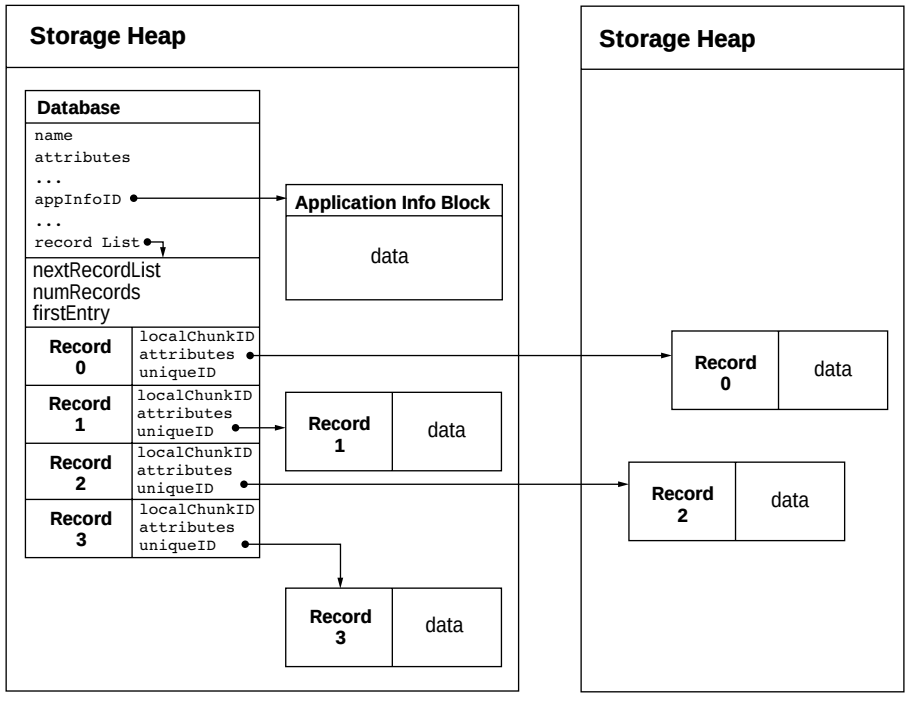


Figure 12-1: A typical Palm OS database and its relation to its records

Table 12-1
Palm OS Database Header

Field	*Size in Bytes*	*Description*
`name`	32	Null-terminated string containing the name of the database
`attributes`	2	Flags specifying properties of the database
`version`	2	Version number of the database format, and value that is application-defined and does not necessarily equal the version number of the application itself

Continued

Table 12-1 *(continued)*

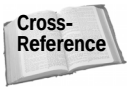
Field	*Size in Bytes*	*Description*
`creationDate`	4	Date and time when the database was created, expressed in seconds since midnight, January 1, 1904
`modificationDate`	4	Date and time when the database was last modified, expressed in seconds since midnight, January 1, 1904
`lastBackupDate`	4	Date and time when the database was last backed up during a HotSync operation
`modificationNumber`	4	Incremented every time an application adds, modifies, or deletes a record in the database
`appInfoID`	4	LocalID where the database's application info block, if any, begins
`sortInfoID`	4	LocalID where the database's sort info block, if any, begins
`type`	4	Application-defined database type
`creator`	4	Creator ID of the database
`uniqueIDSeed`	4	Used by the system to generate a unique ID number for each record in the database
`recordList`	4	LocalID of the first record list, and value that is `0` if there is only one record list for this database

The `recordList` field of the header holds the LocalID of the database's first record list. A database may contain only one record list structure if it has few enough records to fit the entire list within the database header; otherwise, the record list's `nextRecordList` field will contain the LocalID of the next list of records. The structure of a record list is outlined in Table 12-2.

Table 12-2
Record List Structure

Field	*Size in Bytes*	*Description*
`nextRecordListID`	4	LocalID of the next list of records. This value is `0` if there are no other record lists beyond the current list.
`numRecords`	2	Number of records in this record list.
`firstEntry`	2	Placeholder for the memory address of the first record entry in this list.

Starting at the memory location of the `firstEntry` field in the record list, each record list contains an array of record entries, whose structure is outlined in Table 12-3. This structure holds the LocalID of the actual record, the record's attribute flags, and a unique ID for the record within its database. The HotSync Manager uses the unique ID of each record when performing synchronization with the desktop version of a database; if the unique ID of a record on the desktop matches the unique ID of a handheld record, the HotSync Manager considers them to be the same record.



For further details of the inner workings of the HotSync Manager, see Chapter 19, "Understanding Conduits."

Table 12-3
Record List Element

Field	*Size in Bytes*	*Description*
`localChunkID`	4	LocalID of the actual record
`attributes`	1	Attribute and category information for this record
`uniqueID`	3	Unique ID for this record

A record entry's `attributes` field contains four flags and a 4-bit number indicating which category the record belongs to. Table 12-4 shows the byte offsets and meanings of the contents of the `attributes` byte.

Table 12-4
Record Entry Attributes

Field	*Offset*	*Description*
category	0x00	Category of the record
secret	0x10	If set, indicates that this record is marked private
busy	0x20	If set, indicates that this record is currently in use
dirty	0x40	If set, indicates that this record should be archived at the next HotSync operation
delete	0x80	If set, indicates that this record should be deleted at the next HotSync operation

Resource Databases

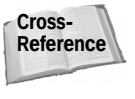
The database format described so far in this chapter is what the Palm OS uses to store record information. There is also a database format in the Palm OS for storing resources. Unlike standard records, a resource contained in a database is tagged with a resource type and an ID number.

Palm OS applications are simply resource databases containing the data, code, and user interface resources necessary to run a program.

Resource databases differ from record databases only in the structure used in the record list to indicate each resource stored in the database. Table 12-5 outlines the structure of a resource entry.

Table 12-5
Record List Resource Element

Field	*Size in Bytes*	*Description*
type	4	Resource type
id	2	ID number for the resource, which is unique among all resources in a database that share the same type
localChunkID	4	LocalID of the actual resource



More details about handling individual resources within a database are available in Chapter 13, “Manipulating Records.”

Working with Databases

Databases on a Palm OS handheld are roughly analogous to files on a desktop computer, except that Palm OS databases reside in RAM instead of the permanent storage medium occupied by desktop files. Using functions in the Palm OS API, your application can create and delete databases, as well as open and close them, much as a desktop system handles its files. Where a Palm OS database differs most from a desktop file is in reading data from and writing data to the database.

Creating Databases

To create a new database, call the **DmCreateDatabase** function. The prototype for **DmCreateDatabase** looks like this:

```
Err DmCreateDatabase (UInt16 cardNo, const Char * nameP,
                      UInt32 creator, UInt32 type,
                      Boolean resDB)
```

The `cardNo` parameter is the memory card on which you wish to create the database.

As of this writing, no Palm OS handheld actually has more than one memory card, so you can pass `0` for the `cardNo` parameter to indicate the first card on the device.

The `nameP` parameter of **DmCreateDatabase** accepts a string to use as the human-readable name for the database. As shown earlier in this chapter, only 32 bytes are available in the database header to store this name, which includes 1 byte for the terminating null, so database names can be a maximum of 31 ASCII characters long.

Aside from meeting the size requirement for a database name, you should also take steps to ensure that the name of your database will be unique. The best way to do this is to append the application's creator ID to the end of the database name. For example, the Librarian sample application's database is named `Librarian-LFlb`.

The `creator` parameter takes a 4-byte creator ID code. Every database belonging to a particular application should be marked with that application's creator ID. Among other things, this ensures that the system application launcher can properly delete all of a program's data along with the program itself when the user chooses to delete it. This also makes it easy for your application to find databases that belong to it.

Another 4-byte code goes into the `type` parameter. This code is application-specific, and it identifies what type of data the database contains. Unlike the creator ID, the type code does not need to be unique among all Palm OS applications. The Palm OS

reserves the code `appl` to represent executable applications; other than this, you can assign just about any 4-byte code you wish, preferably a mnemonic to help you identify the kind of data in the database. By convention, most applications use the code `Data` or `DATA` to indicate a normal database that just contains an application's records.

Many applications require only one database, which is true of all four of the basic ROM applications: the built-in applications all use the type `DATA` for their databases. A good example of multiple database usage is Palm Computing's Expense application, which uses three databases. Expense stores its main records in a `DATA` database, a list of cities in a `city` database, and a list of vendors in a `vend` database.

Another common database type is `HACK`, which is used by Edward Keyes's HackMaster application to represent a system hack application. There is nothing to stop you from using `HACK` for a database type, should you wish to, but it could be confusing to HackMaster users, because your application would then appear in HackMaster as an installed hack.

The last parameter to **DmCreateDatabase**, `resDB`, is a `Boolean` value that tells the data manager to create a resource database instead of a record database if `resDB` is set to `true`. For most normal application databases, leave `resDB` set to `false`.

In general, an application should check for the existence of its database in its **StartApplication** routine. If the database does not exist, **StartApplication** can then call **DmCreateDatabase** to make a new database for the application. The following excerpt from Librarian's **StartApplication** function checks for Librarian's database and creates it if it does not already exist:

```
#define libDBName     "LibrarianDB-LFlb"
#define libDBType     'DATA'
#define libCreatorID  'LFlb'

Err      error = 0;
UInt16   mode;


// Code to set the database mode omitted

// Find Librarian's database. If it doesn't exist, create it.
gLibDB = DmOpenDatabaseByTypeCreator(libDBType, libCreatorID,
                                     mode);
if (! gLibDB) {
    error = DmCreateDatabase(0, libDBName, libCreatorID,
                             libDBType, false);
    if (error)
        return error;

    gLibDB = DmOpenDatabaseByTypeCreator(libDBType,
```

```
                                                libCreatorID, mode);
    if (! gLibDB)
        return DmGetLastErr();

    // Code for initializing the application info block omitted
}
```

The **StartApplication** routine first checks to see if Librarian's database exists by attempting to open it with **DmOpenDatabaseByTypeCreator**, which returns either a reference to the open database or the value `0`. If the database does not exist, **StartApplication** calls **DmCreateDatabase** with Librarian's database name, creator ID, and the type `DATA` to create the new database. Then **StartApplication** opens the newly created database so that Librarian will have access to it. If this second database opening fails, **StartApplication** calls the **DmGetLastErr** function to get the code of the last error encountered by the data manager and passes this error code back to the **PilotMain** routine.

Opening Databases

The **DmOpenDatabaseByTypeCreator** function, shown in the previous example, opens a database given the type and creator ID of the database, and it returns a reference to the open database if successful. The Palm OS type used for an open database reference, and for the `gLibDB` global variable in the last example, is `DmOpenRef`.

A third parameter to **DmOpenDatabaseByTypeCreator** sets the mode in which the database should be opened. The Palm OS provides a number of constants, shown in Table 12-6, for defining database access modes. Your code should OR these values together to form the `mode` parameter to **DmOpenDatabaseByTypeCreator**.

Table 12-6
Database Access Mode Constants

Constant	*Description*
`dmModeReadWrite`	Read/write access
`dmModeReadOnly`	Read-only access
`dmModeWriteOnly`	Write-only access
`dmModeLeaveOpen`	Leave the database open after the application quits
`dmModeExclusive`	Exclude other applications from opening this database
`dmModeShowSecret`	Show records in this database that are marked private

Most of the time, you should use the dmModeReadWrite mode when your application starts, because that mode will allow modification and display of the application's data. To support private records in your application, also check the system preferences to see if the user currently has private records hidden or not, and then set the dmModeShowSecret mode as appropriate. The following code from Librarian's **StartApplication** sets the mode to read/write access and sets the mode to show or hide private records:

```
if (gPrivateRecordStatus == hidePrivateRecords)
    mode = dmModeReadWrite;
else
    mode = dmModeReadWrite | dmModeShowSecret;
```

The global gPrivateRecordStatus variable is of type privateRecordViewEnum, an enumeration defined in the Palm OS 3.5 header PrivateRecords.h as follows:

```
typedef enum privateRecordViewEnum {
    showPrivateRecords = 0x00,
    maskPrivateRecords,
    hidePrivateRecords
} privateRecordViewEnum;
```

Librarian uses gPrivateRecordStatus in various parts of its code to determine how private records should be displayed.

You are entirely responsible for implementing private record behavior in your application if you want to allow the user to mark records as private; the only thing the operating system does to maintain private records is to keep track of how they should be displayed. Your application code must do all the work of checking the system preferences and making sure that hidden records are displayed, masked, or hidden, as appropriate.

Besides **DmOpenDatabaseByTypeCreator**, the Palm OS also allows opening databases with **DmOpenDatabase**. Instead of the type and creator ID of the desired database, **DmOpenDatabase** requires the database's card number and LocalID. See the "Finding Databases" section later in this chapter for information on how to determine a database's LocalID.

Closing Databases

When your application is finished with a database, it should call **DmCloseDatabase** to close it. The **DmCloseDatabase** function takes a DmOpenRef type reference to an open database as a parameter:

```
DmCloseDatabase(gLibDB);
```

An application's **StopApplication** function is a good place to put a call to **DmCloseDatabase**, right after any code that performs any other cleanup required by the application.

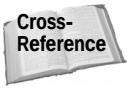
The system allocates approximately 50-100 bytes of memory from the dynamic heap for each open database, so be sure to close databases when they are no longer needed to avoid a memory leak. More importantly, any future attempt to open a database that is already open will fail, including attempts from a desktop conduit during synchronization. Only a soft reset will close databases that are left open.

Finding Databases

A number of functions exist for finding databases on the handheld, given different criteria. The most basic is **DmFindDatabase**, which returns the LocalID of a database header given a card number and the name of the database to search for:

```
LocalID  dbID;

LocalID = DmFindDatabase(0, "Librarian-LFlb");
```

If **DmFindDatabase** cannot find a database that matches the given name, it returns 0; in this case, call **DmGetLastErr** to find out the exact reason **DmFindDatabase** failed.



See Appendix A, "Palm OS API Quick Reference," for a complete list of error codes that may be returned by DmGetLastErr.

The **DmGetDatabase** function returns the LocalID of a database given a card number and the index of the database on the card. Use **DmGetDatabase** to retrieve a list of all the databases on a card. Index numbers for databases range from 0 to the total number of databases on the card, minus one. If you pass the card number to the function **DmNumDatabases** to determine the number of databases, you can then iterate over all the databases on that card:

```
LocalID  dbID;
Int16    i;

for (i = 0; i < DmNumDatabases(0); i++) {
    dbID = DmGetDatabase(0, i);
    // Do something with the dbID, the database's LocalID.
}
```

Finally, the Palm OS also offers **DmGetNextDatabaseByTypeCreator** for more complex searches. The **DmGetNextDatabaseByTypeCreator** function searches all the

handheld's memory cards for a database given a type, creator ID, or both, and it has the following prototype:

```
Err DmGetNextDatabaseByTypeCreator (Boolean newSearch,
    DmSearchStatePtr stateInfoP, UInt32 type, UInt32 creator,
    Boolean onlyLatestVers, UInt16* cardNoP, LocalID* dbIDP)
```

The **DmGetNextDabaseByTypeCreator** function returns `0` if it successfully found a matching database, or the constant `dmErrCantFind` if no database matching the supplied `type` or `creator` can be found.

It is necessary to call **DmGetNextDatabaseByTypeCreator** more than once to get all the databases on the handheld with the specified criteria. The first parameter, `newSearch`, tells **DmGetNextDatabaseByTypeCreator** to start a brand new search if its value is `true`. Passing a pointer to a `DmSearchStateType` structure in the `stateInfoP` parameter allows the function to keep track of its search state, so subsequent calls to the function with a `newSearch` value of `false` can pick up the search where it left off after the last call to **DmGetNextDatabaseByTypeCreator**.

The `type` and `creator` parameters accept the database type and creator ID you wish to search for, respectively. You may pass `NULL` for either of these parameters to specify a wildcard search. If `type` is `NULL`, the routine returns databases of any type that match the specified `creator`. Likewise, if `creator` is `NULL`, the search will return databases of the specified `type`, but with any creator ID. Passing `NULL` to both parameters returns every database on the handheld.

Pass `true` for the value of the `onlyLatestVers` parameter to restrict the search to the latest version of each database. A `false` value for `onlyLatestVers` allows for retrieval of all databases matching the specified `type` and `creator`, regardless of their versions.

Databases in RAM are always considered a more recent version than databases stored in ROM memory. This allows you to replace any of the built-in applications stored in ROM with your own.

There is a special case to look out for when `onlyLatestVers` is set to `true` that has to do with changes in implementation of **DmGetNextDatabaseByTypeCreator** between Palm OS versions 3.0 and 3.1. If multiple databases exist on the handheld that all share the same type, creator ID, and version, Palm OS 3.0 and earlier return all of those databases. However, Palm OS 3.1 and later return only one of the multiple databases when `onlyLatestVers` is `true`, effectively selected at random. If `onlyLatestVers` is `false`, **DmGetNextDatabaseByTypeCreator** works the same way across all versions of the Palm OS.

To ensure that calls to DmGetNextDatabaseByTypeCreator work in the same way across all versions of the Palm OS, either set `onlyLatestVers` to `false` when specifying both the `type` and `creator` parameters or pass `NULL` for the value of `type`, `creator`, or both.

The two remaining parameters to **DmGetNextDatabaseByTypeCreator**, `cardNoP` and `dbIDP`, receive the card number in which a found database resides and the database's LocalID on that card.

The following helper function counts the number of databases on the handheld matching a given type and creator ID:

```
Int16 CountDatabases(UInt32 type, UInt32 creator) {
    DmSearchStateType  searchState;
    Int16    count = 0;
    UInt16   cardNo;
    LocalID  dbID;


    if (DmGetNextDatabaseByTypeCreator(true, &searchState,
        type, creator, false, &cardNo, &dbID) {
        do {
            count++;
            // Do something with each database here, if
            // desired, using cardNo and dbID.
        } while (DmGetNextDatabaseByTypeCreator(false,
                 &searchState, myType, myCreator, false,
                 &cardNo, &dbID);
    }

    return count;
}
```

Deleting Databases

To remove a database and all its records from the handheld, call the **DmDelete Database** function. The function takes two parameters, the card number where the database is located and the database's LocalID, and it returns `0` if successful or an error code if deletion failed for some reason. Use one of the functions in the previous section to retrieve the card and LocalID for the database you wish to delete.

There is no way to recover data deleted using DmDeleteDatabase, so be careful how you use this function.

A related function, **DmDatabaseProtect**, allows you to prevent a database from being deleted, thus allowing you to keep a particular record or resource in a database locked without keeping the database open. The **DmDatabaseProtect** function operates by increasing or decreasing the *protection count* on a particular database. If the protection count assigned to a database is greater than 0, the database may not be deleted. The prototype for **DmDatabaseProtect** looks like this:

```
Err DmDatabaseProtect (UInt16 cardNo, LocalID dbID,
                       Boolean protect)
```

The `cardNo` and `dbID` parameters accept the card number and LocalID of the database to modify, respectively. Passing a value of `true` the `protect` parameter increments the protection count; a `false` `protect` value decrements the protection counter.

The system keeps protection count information in dynamic memory, so all databases become "unprotected" whenever the handheld is reset.

Retrieving and Modifying Database Information

A database stores a lot of information about itself in its header. The Palm OS function for retrieving this information is **DmDatabaseInfo**, which has the following prototype:

```
Err DmDatabaseInfo (UInt16 cardNo, LocalID dbID, Char* nameP,
    UInt16* attributesP, UInt16* versionP, UInt32* crDateP,
    UInt32* modDateP, UInt32* bckUpDateP, UInt32* modNumP,
    LocalID* appInfoIDP, LocalID* sortInfoIDP, UInt32* typeP,
    UInt32* creatorP)
```

The `cardNo` and `dbID` parameters take the card number and LocalID of the database whose information you wish to retrieve. Only the first two parameters provide input to **DmDatabaseInfo**; everything else receives return values from the function. Pass `NULL` for the value of any of the remaining parameters to ignore that particular piece of information.

First in the long list of properties is `nameP`, which retrieves the name of the database. Make sure the character array whose pointer you pass in `nameP` is 32 bytes long so it has enough room to contain the longest possible database name string.

The `attributesP` parameter receives the attribute flags associated with the database. In the Palm OS header file `DataMgr.h`, a number of constants are defined for handling database attributes. Table 12-7 shows the attribute constants and what they mean.

Table 12-7
Database Attribute Flag Constants

Constant	*Value*	*Description*
`dmHdrAttrResDB`	`0x0001`	Set if database is a resource database.
`dmHdrAttrReadOnly`	`0x0002`	Set if database is read-only.
`dmHdrAttrAppInfoDirty`	`0x0004`	Set if application info block is dirty.
`dmHdrAttrBackup`	`0x0008`	Set if this database should be backed up to the desktop during a HotSync operation; this bit should be set if there is no conduit associated with this application to perform backup duties.
`dmHdrAttrOKToInstallNewer`	`0x0010`	Set if it is okay for the HotSync backup conduit to install a newer version of this database with a different name if this database is currently open.
`dmHdrAttrResetAfterInstall`	`0x0020`	Set if the handheld requires a reset after installing this database.
`dmHdrAttrCopyPrevention`	`0x0040`	Set if the database should not be copied via IR beaming or other methods.
`dmHdrAttrStream`	`0x0080`	Set if this database is a file stream.
`dmHdrAttrHidden`	`0x0100`	Set if this database should be hidden from view. For example, the application launcher on Palm OS 3.2 and later hides applications with this bit set. For record databases, setting this bit hides the record count in the application launcher's Info screen.
`DmHdrAttrLaunchableData`	`0x0200`	Set on a non-application database if this database may be "launched" by its name being passed to its owner (an application database with the same creator ID) via the `sysAppLaunchCmdOpenNamedDB` launch code; for example, Palm Query Applications (PQAs) have this bit set so they will appear in the application launcher, even though they are not actually applications.
`DmHdrAttrOpen`	`0x8000`	Set if the database is open.

The DataMgr.h header also defines the constants dmAllHdrAttrs and dmSysOnly HdrAttrs, which are bit masks representing all the header attributes and attributes that may be altered only by the system, respectively:

```
#define dmAllHdrAttrs (dmHdrAttrResDB | \
                       dmHdrAttrReadOnly | \
                       dmHdrAttrAppInfoDirty | \
                       dmHdrAttrBackup | \
                       dmHdrAttrOKToInstallNewer | \
                       dmHdrAttrResetAfterInstall | \
                       dmHdrAttrCopyPrevention | \
                       dmHdrAttrStream | \
                       dmHdrAttrOpen)

#define dmSysOnlyHdrAttrs (dmHdrAttrResDB | \
                           dmHdrAttrOpen)
```

The versionP parameter receives the version number for this database. By default, databases all have a version of 0.

If you change the format of the records within a database between different versions of your application, it is also a good idea to increment the database version so your application can tell the difference between old and new database formats and deal with them appropriately.

Three parameters, crDateP, modDateP, and bckUpDateP, receive timestamps related to the database; crDateP is the database's creation date, modDateP is the data of the last modification made to the database, and bckUpDateP is the last time the database was backed up via a HotSync operation. All of these values are stored as the number of seconds since midnight on January 1, 1904.

Different versions of the Palm OS deal differently with the modification date field. Version 1.0 never updates the modification date, and version 2.0 updates the modification date when a database opened in writable mode is closed. Not until version 3.0 does the system actually update the modification date only when something in the database has actually been changed, such as adding, deleting, archiving, rearranging, or resizing records; setting a record's dirty bit via DmReleaseRecord; rearranging or deleting categories; or updating the database's header fields using DmSetDatabaseInfo. If you need to ensure that the modification date is updated the same way across all versions of the Palm OS, set the modification date manually with the DmSetDatabaseInfo function.

The modNumP parameter receives the database's modification number, a value that the system increments every time a record in the database is added, modified, or deleted.

Retrieving the `appInfoIDP` and `sortInfoIDP` values gives you the LocalIDs of the application info block and sort info block for this database, respectively. Because both of these blocks are optional in any database, these values will be `NULL` if no application info block or sort info block exists for the database.

The `typeP` and `creatorP` parameters receive the type and creator ID of the database, respectively.

To set database header information, use **DmSetDatabaseInfo**. Like its companion **DmDatabaseInfo**, **DmSetDatabaseInfo** has a lot of parameters:

```
Err DmSetDatabaseInfo (UInt16 cardNo, LocalID dbID,
    const Char* nameP, UInt16* attributesP, UInt16* versionP,
    UInt32* crDateP, UInt32* modDateP, UInt32* bckUpDateP,
    UInt32* modNumP, LocalID* appInfoIDP, LocalID* sortInfoIDP,
    UInt32* typeP, UInt32* creatorP)
```

Just as in **DmDatabaseInfo**, the first two parameters to **DmSetDatabaseInfo** are the card number and LocalID of the database to work with. All the other parameters are pointers to values that should be modified in the database. Passing `NULL` for any parameter leaves that parameter's associated value unchanged in the database header.

As an example of how to use **DmSetDatabaseInfo**, the following code snippet from Librarian's **StartApplication** routine sets the backup bit on a newly created database:

```
UInt16   cardNo;
LocalID  dbID;
UInt16   attributes;


DmOpenDatabaseInfo(gLibDB, &dbID, NULL, NULL, &cardNo, NULL);
DmDatabaseInfo(cardNo, dbID, NULL, &attributes, NULL, NULL,
               NULL, NULL, NULL, NULL, NULL, NULL, NULL);
attributes |= dmHdrAttrBackup;
DmSetDatabaseInfo(cardNo, dbID, NULL, &attributes, NULL, NULL,
                  NULL, NULL, NULL, NULL, NULL, NULL, NULL);
```

Setting a database's `attributes` field is a two-step process. First, you must retrieve the existing attributes using **DmDatabaseInfo**. Then, you can OR the retrieved attributes together with one or more of the database attribute constants, such as `dmHdrAttrBackup`.

If you do not plan to write a conduit for an application, you should set the backup bit on its database so the HotSync backup conduit will make a copy of the application's data in the user's backup folder on the desktop machine. If you wish to convert a Palm OS application's saved data to a more desktop-friendly format, you will need to either write a conduit or a desktop application that can perform the conversion.

Aside from the database header functions just mentioned, the Palm OS also provides functions for retrieving other pieces of valuable data about databases on the handheld. The **DmOpenDatabaseInfo** function returns information about an open database, and it has the following prototype:

```
Err DmOpenDatabaseInfo (DmOpenRef dbP, LocalID* dbIDP,
    UInt16* openCountP, UInt16* modeP, UInt16* cardNoP,
    Boolean* resDBP)
```

The first parameter, `dpP`, is a pointer to an open database reference. Much as its cousin **DmDatabaseInfo** does, the rest of the parameters retrieve various pieces of data about the database. Pass `NULL` for the value of any parameter you are not interested in.

Retrieving `dpIDP` gives you the LocalID of the database. The `openCountP` parameter receives the number of applications that have this database open, `modeP` receives the mode used to open the database, `cardNoP` holds the number of the card where this database resides, and `resDBP`, if requested, contains `true` if this is a resource database or `false` if it is a regular record database.

Use the **DmGetDatabaseLockState** function to retrieve information about the number of locked and busy records in a database:

```
void DmGetDatabaseLockState (DmOpenRef dbR, UInt8* highest,
                             UInt32* count, UInt32* busy)
```

Pass an open database reference for the value of the `dbR` parameter, and **DmGetDatabaseLockState** returns the highest lock count of any of the database's records in `highest`, the number of records having the lock count specified by `highest` in the `count` parameter, and the number of records with their busy bits set in `busy`. As usual, pass `NULL` for any value you do not wish to retrieve.

The **DmDatabaseSize** function returns information about the size of the database:

```
Err DmDatabaseSize (UInt16 cardNo, LocalID dbID,
                    UInt32* numRecordsP, UInt32* totalBytesP,
                    UInt32* dataBytesP)
```

Given the card number and LocalID of a database, **DmDatabaseSize** returns the number of records in the database via the `numRecordsP` parameter, the total size in bytes occupied by the database in `totalBytesP`, and the total memory occupied by just the data, not counting the overhead in the database's header, in the `dataBytesP` parameter.

Creating an Application Info Block

The *application info block* is an optional block that stores application-defined data related to the database as a whole, as opposed to the data stored in individual records. Among other things, most of the ROM applications use an application info block to keep track of the user-customizable category names in the application. The memory devoted to an application info block, much like the memory used for individual records, may be located anywhere on the same card as a database's header, which keeps track of the LocalID of the application info block.

Databases may also have an optional *sort info block*, which Palm Computing originally intended to contain information about how the records in the database should be sorted. However, under current implementations of the Palm OS, the HotSync manager does not back up the sort info block, so it is not a good place to store information that needs to persist across synchronizations. You could use the sort info block to temporarily cache information about the database that can be regenerated from other data, but most Palm OS applications do not use it at all.

When you first create a new database that has an application info block, you should initialize the new database's application info block. The Librarian sample application calls a helper function, **LibAppInfoInit**, from its **StartApplication** routine to take care of creating the new application info block:

```
error = LibAppInfoInit(gLibDB);
if (error) {
    DmCloseDatabase(gLibDB);
    DmDeleteDatabase(cardNo, dbID);
    return error;
}
```

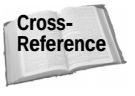
Notice that if **LibAppInfoInit** fails for some reason to create the application info block, **StartApplication** cannot successfully complete its mission to create Librarian's database, so it calls **DmCloseDatabase** to close the empty database it just created, then **DmDeleteDatabase** to remove it.

Librarian's application info block is a structure called `LibAppInfoType`, defined as follows in `librarianDB.h`:

```
typedef struct {
    UInt16  renamedCategories;  // bit field of categories with
                                // a different name
    char  categoryLabels[dmRecNumCategories][dmCategoryLength];
    UInt8 categoryUniqIDs[dmRecNumCategories];
    UInt8 lastUniqID;
    UInt8 reserved1;  // from the compiler word aligning things
    UInt16  reserved2;
    // End of category structure; Librarian-specific
```

```
    // application info starts here.
    UInt8     showInList;  // Current sort order for database
    libLabel  fieldLabels[libNumFieldLabels];  // Labels in
                                               // Edit view
    // The following are status strings for the Record view:
    libLabel  bookStatusStrings[libNumBookStatusStrings];
    libLabel  printStatusStrings[libNumPrintStatusStrings];
    libLabel  formatStatusStrings[libNumFormatStatusStrings];
    libLabel  readStatusStrings[libNumReadStatusStrings];
} LibAppInfoType;
```

The first six fields in `LibAppInfoType` are required in the application info block of any application that supports the standard implementation of Palm OS categories; just tack these fields onto the front of any application info structure you define for your own application, and the various Palm OS category functions will be able to function properly.

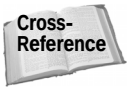


Chapter 13, "Manipulating Data," covers the category functions in detail.

The `showInList` field stores the current sort order used to display records in Librarian's List view. Librarian's `librarianDB.h` header defines the following constants for keeping track of sort order:

```
#define libShowAuthorTitle  0
#define libShowTitleAuthor  1
#define libShowTitleOnly    2
```

The rest of the fields in `LibAppInfoType` store the various strings used to display status information about a record in Librarian's Record view. These strings are things like "Got this book," "Paperback," and "Unread," and Librarian has application info string resources containing these string values. To save time while the application is running, Librarian avoids retrieving these resource strings from the application's resources every time it needs them. Instead, it copies these resources into the application info block when it first creates and initializes its database. In that way, Librarian needs to open the application info block only once each time it runs to retrieve these values, instead of having to retrieve each individual string every time it is needed.



More details of the mechanism Librarian uses to keep track of status strings in the Record view are available in Chapter 13, "Manipulating Records."

The relevant portions of Librarian's **LibAppInfoInit** function look like this:

```
Err LibAppInfoInit(DmOpenRef db)
{
    UInt16     cardNo;
    LocalID    dbID, appInfoID;
```

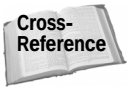
```
    MemHandle  h;
    LibAppInfoType  *appInfo;


    if (DmOpenDatabaseInfo(db, &dbID, NULL, NULL, &cardNo,
        NULL))
        return dmErrInvalidParam;
    if (DmDatabaseInfo(cardNo, dbID, NULL, NULL, NULL, NULL,
        NULL, NULL, NULL, &appInfoID, NULL, NULL, NULL))
        return dmErrInvalidParam;

    // If there isn't an app info block make space for one.
    if (appInfoID == NULL) {
        h = DmNewHandle(db, sizeof(LibAppInfoType));
        if (!h) return dmErrMemError;

        appInfoID = MemHandleToLocalID(h);
        DmSetDatabaseInfo(cardNo, dbID, NULL, NULL, NULL, NULL,
            NULL, NULL, NULL, &appInfoID, NULL, NULL, NULL);
    }

    appInfo = MemLocalIDToLockedPtr(appInfoID, cardNo);

    // Clear the app info block.
    DmSet(appInfo, 0, sizeof(LibAppInfoType), 0);

    // Code for initializing categories and fields omitted

    MemPtrUnlock(appInfo);

    return 0;
}
```

The **LibAppInfoInit** function first retrieves the card number and LocalID of the database using the **DmOpenDatabaseInfo** function, and then passes these values to **DmDatabaseInfo** to retrieve `appInfoID`, the LocalID of the database's application info block.

If `appInfoID` is equal to `NULL`, then there is no application info block defined for the database, in which case **LibAppInfoInit** proceeds to make space for a new application info block by calling **DmNewHandle**. Notice that **LibAppInfoInit** uses the **DmNewHandle** function here instead of **MemHandleNew**. **DmNewHandle** allocates space in the same storage heap occupied by Librarian's database, as opposed to **MemHandleNew**, which would allocate space in the dynamic heap. Also, because handles cannot be stored in the database, **LibAppInfoInit** converts the handle to a LocalID with the **MemHandleToLocalID** function, and then passes this value to **DmSetDatabaseInfo** to set the application info block location in the database's header.

Then **LibAppInfoInit** locks a pointer to the new application info block using the **MemLocalIDToLocketPtr** function. With this pointer, **LibAppInfoInit** can zero the memory of the application info block using **DmSet**, the storage heap equivalent of the **MemSet** function.

After creating the new application info block, **LibAppInfoInit** initializes the categories and other fields in the `appInfo` structure. This initialization delves into managing categories and retrieving individual resources, topics left for a later chapter.



Chapter 13, "Manipulating Records," revisits the `LibAppInfoInit` function and fills in the details of initializing individual fields in an application info block.

Once all the values have been initialized, **LibAppInfoInit** is finished with the pointer to the application info block, so it unlocks it with **MemPtrUnlock**.

Storing Application Preferences

Devoting an entire database to the storage of small pieces of data, such as which columns are visible in a given application view, is simply overkill. Databases are well suited to storing many pieces of data with a common format, but there is a better way to keep track of smaller odds and ends that a program needs to remember between invocations: *application preferences*. Application preferences are somewhat analogous to `.ini` files in Windows or `rc` files in Unix, in that they are a handy place to store configuration data for an application.

The Palm OS maintains a database of preference information for all the applications on the device that wish to use it. Each record in this database is application-defined, so a program can store whatever data it needs to for its own purposes.

Before using application preferences, you need to define a structure to contain them. As an example, the following code from `librarian.c` defines `LibPreferenceType`, the structure used in Librarian to store application preferences:

```
typedef struct {
    UInt16   currentCategory;
    Boolean  showAllCategories;
    Boolean  showBookStatus;
    Boolean  showPrintStatus;
    Boolean  showFormat;
    Boolean  showReadUnread;
    Boolean  saveBackup;
    FontID   listFont;
    FontID   recordFont;
    FontID   editFont;
    FontID   noteFont;
} LibPreferenceType;
```

None of the data stored in `LibPreferenceType` is particularly lengthy; it is simply a mishmash of values that should be maintained between invocations of Librarian for consistency's sake, but that don't really fit anywhere in its database. The `currentCategory` and `showAllCategories` fields keeps track of the category currently displayed by Librarian. The `showBookStatus`, `showPrintStatus`, `showFormat`, and `showReadUnread` fields store which columns of the List view are visible, and `saveBackup` saves whether the check box in the delete confirmation dialog box is checked or not. Finally, the various `FontID` fields save the user's font preferences for various views in Librarian.

Once you have determined the format required to store application preferences, your application needs to retrieve these settings when it first opens, usually in its **StartApplication** routine. The parts of Librarian's **StartApplication** function that deal with retrieving Librarian's preferences look like this:

```
LibPreferenceType  prefs;
UInt16  prefsSize;


prefsSize = sizeof(LibPreferenceType);
if (PrefGetAppPreferences(libCreatorID, libPrefID, &prefs,
    &prefsSize, true) != noPreferenceFound) {
    gCurrentCategory = prefs.currentCategory;
    gShowAllCategories = prefs.showAllCategories;
    gShowBookStatus = prefs.showBookStatus;
    gShowPrintStatus = prefs.showPrintStatus;
    gShowFormat = prefs.showFormat;
    gShowReadUnread = prefs.showReadUnread;
    gSaveBackup = prefs.saveBackup;
    gListFont = prefs.listFont;
    gRecordFont = prefs.recordFont;
    gEditFont = prefs.editFont;
    gNoteFont = prefs.noteFont;
}
else {
    // No preferences exist yet, so set the defaults for the
    // global font variables.
    gListFont = stdFont;
    gNoteFont = stdFont;

    // If Librarian is running on Palm OS 2.0, the
    // largeBoldFont is invalid. In that case, substitute
    // stdFont.
    if (gROMVersion <
        sysMakeROMVersion(3,0,0,sysROMStageRelease,0)) {
        gRecordFont = stdFont;
        gEditFont = stdFont;
    }
    else {
        gRecordFont = largeBoldFont;
        gEditFont = largeBoldFont;
    }
}
```

Librarian's **StartApplication** routine calls **PrefGetAppPreferences** to retrieve Librarian's `LibPreferenceType` structure from the system's list of application preferences. The **PrefGetAppPreferences** function has the following prototype:

```
Int16 PrefGetAppPreferences (UInt32 creator, UInt16 id,
    void* prefs, UInt16* prefsSize, Boolean saved)
```

The first parameter to **PrefGetAppPreferences**, `creator`, is the creator ID of the application whose preferences should be retrieved.

It is possible for an application to have more than one preference type, which is where the `id` parameter comes into play. The `id` lets you assign an application-defined `UInt16` value to identify each of the application's preference types. Librarian has only one preference type, defined by the constant `libPrefID` in `librarian.h`, which has a value of `0`.

The `void` pointer `prefs` receives the actual preferences structure. To let the system know how large this structure is you must also pass the size of the structure, in bytes, via the `prefsSize` parameter. It is also possible to retrieve variable-length structures from application preferences. To find out how large the structure stored in an application's preferences is, call **PrefGetAppPreferences** once with a `NULL` pointer for the `prefs` parameter and a `prefsSize` of `0`. The **PrefGetAppPreferences** functions sets `prefsSize` to the actual size of the buffer holding the application's preferences. Once you have the actual size of the structure, you can allocate a buffer large enough to hold the preferences and call **PrefGetAppPreferences** a second time to retrieve them.

The **PrefGetAppPreferences** function's last parameter, `saved`, specifies whether to retrieve saved preferences or unsaved preferences. The system maintains two lists of preferences, those that should be saved during a HotSync operation and those that do not need to be backed up. In general, you should set `saved` to `true` so that your application's preferences are saved by the HotSync manager.

If **PrefGetAppPreferences** is unable to find the specified application preferences, it returns the constant `noPreferenceFound`. If the preferences were retrieved, Librarian's **StartApplication** function sets a series of global variables based on the contents of the saved preferences. If the preferences do not exist yet, Librarian sets the same global variables to reasonable defaults.

Once your application is ready to exit, it should save its preferences again using the **PrefSetAppPreferences** function, which takes almost the same parameters as **PrefGetAppPreferences**, with the addition of an `Int16` value to indicate what version of the application's preferences should be saved. Librarian's **StopApplication** function saves application preferences with the following code:

```
LibPreferenceType  prefs;


// Save Librarian's preferences.
```

```
    prefs.currentCategory = gCurrentCategory;
    prefs.showAllCategories = gShowAllCategories;
    prefs.showBookStatus = gShowBookStatus;
    prefs.showPrintStatus = gShowPrintStatus;
    prefs.showFormat = gShowFormat;
    prefs.showReadUnread = gShowReadUnread;
    prefs.saveBackup = gSaveBackup;
    prefs.listFont = gListFont;
    prefs.recordFont = gRecordFont;
    prefs.editFont = gEditFont;
    prefs.noteFont = gNoteFont;

    PrefSetAppPreferences(libCreatorID, libPrefID,
        libPrefVersionNum, &prefs, sizeof(prefs), true);
```

After filling `prefs` with the data from several of Librarian's global variables, **StopApplication** calls **PrefSetAppPreferences** to store these values in the system's application preferences database.

You may have noticed that it is possible to create preferences with PrefSetApp Preferences that do not match your application's creator ID, simply by passing a different creator ID to the function. However, when the user deletes an application, the system removes only those preferences that match the creator ID of the deleted application. Any other preferences created by the application remain on the device, occupying precious storage space and unnecessarily slowing down HotSync operations, since the system preferences database is updated on the desktop system at every synchronization.

Some shareware developers have used this quirk of the Palm OS preferences mechanism to implement programs that disable themselves after they have been installed for a certain period of time. These applications store a timestamp of when the application was first run in a preference whose creator ID does not match that of the application itself. When the shareware trial period is up, deleting the application and reinstalling it will not reset the timestamp, resulting in an application that is still disabled.

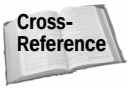
The issue of whether or not to write Palm OS applications that leave orphaned preferences behind has been hotly debated in Palm OS programming forums. On one hand, it is a reasonably effective way to enforce shareware trial periods. On the other hand, it is rude to a handheld user to leave garbage data behind, data that slows down every HotSync operation. Weigh the advantages of using orphaned preferences carefully against their disadvantages before using them.

Reading and Setting System Preferences

The Palm OS also maintains a database of global settings for the system itself. Some of these settings, such as whether private records are currently hidden or what the current date and time formats are, may be of interest to your application. The

PrefGetPreferences function allows you to retrieve the system preferences, which the function returns as a `SystemPreferencesType` structure:

```
SystemPreferencesType  sysPrefs;

PrefGetPreferences(&sysPrefs);
```

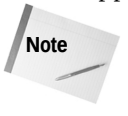


The `SystemPreferencesType` structure is large and convoluted, and it has evolved continuously with each new release of the Palm OS. See Appendix A, "Palm OS API Quick Reference," for a complete listing of all the system preference fields and their meanings.

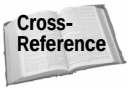
Because the `SystemPreferencesType` occupies a fair chunk of memory, if you need to retrieve only a single value from the system preferences, use the **PrefGet Preference** function, instead of **PrefGetPreferences**. In fact, the documentation from Palm Computing recommends that you use the newer **PrefGetPreference** instead of **PrefGetPreferences**. The **PrefGetPreference** function takes a member of the enum `SystemPreferencesChoice`, defined in the Palm OS header `Preferences.h`, as an argument, which determines the preference the function retrieves and returns as a `UInt32` value. You may have to cast the return value, because the data stored in the system preferences are not all `UInt32` values. As an example, here is the section of Librarian's **StartApplication** function that retrieves the private records system preference:

```
if (gROMVersion >=
    sysMakeROMVersion(3,5,0,sysROMStageRelease,0))
    gPrivateRecordStatus = (privateRecordViewEnum)
        PrefGetPreference(prefShowPrivateRecords);
else {
    if ((Boolean) PrefGetPreference(prefHidePrivateRecordsV33))
        gPrivateRecordStatus = hidePrivateRecords;
    else
        gPrivateRecordStatus = showPrivateRecords;
}
```

Prior to Palm OS 3.5, record masking does not exist, so the system preference controlling whether or not private records should be shown is a simple `Boolean` value, `prefHidePrivateRecordsV33`, which the code in the example translates into an appropriate `privateRecordViewEnum` value for use in Librarian.



If you are using earlier Palm OS header files than those that ship with the 3.5 SDK, you should retrieve private record display status by passing the `prefHidePrivateRecords` constant to PrefGetPreference. The `prefHidePrivateRecordsV33` constant was introduced with the advent of Palm OS 3.5, because some restructuring of the `SystemPreferencesChoice` enum was necessary to squeeze in the new `prefShowPrivateRecords` constant.



Much like the `SystemPreferencesType` structure, the `SystemPreferencesChoice` enum has evolved considerably since its introduction in Palm OS version 2.0. See Appendix A, "Palm OS API Quick Reference," for a complete listing of the members of the `SystemPreferencesChoice` enumerated type.

The Palm OS also offers **PrefSetPreference** for directly setting system-wide preferences. The **PrefSetPreference** function has the following prototype:

```
void PrefSetPreference (SystemPreferencesChoice choice,
                        UInt32 value)
```

You must cast any value you wish to set via **PrefSetPreference** into a `UInt32` value before passing it to the function, even though many values in the `SystemPreferencesType` structure are not of type `UInt32`. For example, the following code sets the handheld's default long date format to display dates in the form of `27 Feb 2000`:

```
PrefSetPreference(prefLongDateFormat, (UInt32) dfDMYLong);
```

You may also use the older **PrefSetPreferences** function to set system preferences, but like **PrefGetPreferences**, it requires making space in memory for the entire `SystemPreferencesType` structure. Palm Computing recommends that you not use **PrefSetPreferences** in current versions of the Palm OS.

Using Feature Memory

Starting with version 3.1, the Palm OS supports the use of *feature memory*. Feature memory stores data in a storage heap rather than in the dynamic heap, allowing stored values to persist between invocations of an application. However, feature memory does not survive across a soft reset, so you should not store any data that requires actual permanence exclusively in feature memory. Also, since feature memory exists in a storage heap, it is no quicker to write to than it is to write to a database, so anything stored in feature memory should not require frequent modification. Primarily, feature memory is a performance optimization, used in situations where an application does not have access to its global variables, or where the 32-bit storage available in a regular feature (see Chapter 10, "Programming System Elements) is not enough to hold the data.

To allocate a chunk of feature memory, call the **FtrPtrNew** function, which has the following prototype:

```
Err FtrPtrNew (UInt32 creator, UInt16 featureNum, UInt32 size,
               void **newPtrP)
```

The `creator` parameter is the creator ID of the application requesting a chunk of feature memory, and `featureNum` is the application-defined number identifying the feature itself. Specify the size of the desired memory chunk, in bytes, in the `size` parameter. The **FtrPtrNew** function returns a pointer to the newly allocated chunk via the `newPtrP` parameter.

Once you have allocated feature memory, you write to it using the **DmWrite** function, because memory in the storage area does not support direct writing:

```
DmWrite(myData, 0, &myData, sizeof(myData));
```

You can retrieve values from feature memory using the **FtrGet** function:

```
Err  error;

error = FtrGet(myCreator, myFeatureNum, (UInt32 *)&value);
```

The **FtrGet** function returns `0` if it successfully retrieves a feature, or an error code if it could not find the requested feature. Because the values normally stored by the feature manager are simple `UInt32` numeric values, you must cast the return value in the **FtrGet** function's third parameter to a pointer type. Once you have the pointer, you may read from it like any other pointer; if you want to modify the value at the pointer, though, be sure to use **DmWrite**.

An example for proper use of feature memory is a function that needs to access an application's preferences in response to a launch code other than `sysAppLaunch CmdNormalLaunch`. If this function is called frequently, opening the preferences database each time could be quite slow. The following example allocates a chunk of feature memory and caches the application's preferences within it, so that the function needs to perform the slow database opening routine only once after a soft reset instead of every time the function is called:

```
MyAppPreferencesType  prefs,
void  *newPrefs;


if (FtrGet(myCreator, myFeatureNum, (UInt32 *)&prefs) != 0) {
    // The feature memory does not exist, so allocate it.
    FtrPtrNew(myCreator, myFeatureNum, 32, &newPrefs);

    // Open the preferences database.
    PrefGetAppPreferences(myCreator, myPrefID, &prefs,
                          sizeof(prefs), true);

    // Write the preferences to feature memory.
    DmWrite(newPrefs, 0, &prefs, sizeof(prefs));
}
```

The Palm OS also offers other functions for manipulating feature memory, such as **FtrPtrResize** for resizing a chunk of feature memory and **FtrPtrFree** for explicitly releasing the memory allocated to a chunk of feature memory. The **FtrPtrFree** function also unregisters the feature that holds the pointer to the feature memory chunk, which clears the reference to the freed memory from the system feature table, thereby preventing accidental use of memory that no longer exists.

Summary

In this chapter, you were shown three different persistent storage techniques in the Palm OS — databases, preferences, and feature memory — and how to use them. After reading this chapter, you should understand the following:

- A Palm OS *database* consists of some header information and a list of *records* or *resources*, which may reside anywhere on the same card as the database header.
- Databases keep track of the locations of their records using the *LocalID* of each record, an offset from the start of the memory card where the record lives.
- Every database may be identified by its unique name, a creator ID, and its database type.
- Databases may be opened via the **DmOpenDatabase** function in a number of different modes, including read/write, read only, write only, and either showing or hiding private records.
- Most of a database's header information may be viewed with **DmDatabaseInfo** and modified with **DmSetDatabaseInfo**.
- An important part of creating a new database is the creation and initialization of its *application info block*, a section of memory that maintains information about the database as a whole.
- *Application preferences* provide a way to store information that is not appropriate for storage in the application's database but that should still be saved between invocations of the application.
- Use the **PrefGetPreference** and **PrefSetPreference** functions to retrieve and set the values of system preferences.
- *Feature memory* is a performance optimization for storing small amounts of data that does not need to persist over soft system resets.

✦ ✦ ✦

CHAPTER 13

Manipulating Records

In This Chapter

Comparing records

Creating, deleting, and modifying records

Finding and sorting records

Categorizing records

Implementing private records

Creating, deleting, and modifying resources

Reading resource data

Implementing the global find facility

Chapter 12, "Storing and Retrieving Data," discussed manipulation of databases on a grand scale, covering creation, deletion, and modification of entire databases. This chapter looks at databases on a smaller scale, concentrating on individual records within a database. Most of the work a Palm OS application must perform to save, retrieve, and modify its data happens at the record level.

This chapter also deals with records' close cousins, resources. Most resources are user interface elements, which you need only create in Constructor or PilRC and then access with the usual Palm OS user interface routines. Some resources, like bitmaps and app info string lists, require an application to use different techniques to make use of their data when an application is running, and this chapter will show you how.

Once you have an understanding of how to use database records, you can implement one of the most useful features of the Palm OS: the global find system. The last section of this chapter shows you what you need to add to your application to make it support the global find facility.

Working with Records

Before we dive into the specifics of manipulating records, an explanation of the Palm OS philosophy behind organizing records is in order. In the interests of efficiency, the ROM applications and the Palm OS database routines rely on a pre-sorted database model. It is not a requirement that your database be sorted, and the Palm OS can handle unsorted databases, but storing records in a sorted order allows for rapid population of tables and lists from database records.

Decreasing the time required to display data in this way causes data to appear more quickly on the screen, thereby not irritating an impatient user. In a database containing many records, it is much quicker for an application to iterate over record indices in a sorted database than it is to skip around through the database, searching for individual records.

So if the Palm OS database model prefers a pre-sorted set of records, does this mean you are stuck with sorting your database in only one way? Not at all. Instead of trying to sort records every time you try to display them, you can re-sort the database only when the user wants to change the sort order. The built-in Address Book application and Librarian both use this approach. When the user changes the sort order in either application's preferences dialog, the application sorts the entire database when the user closes the dialog. In this way, the lengthy sorting process happens only infrequently, when the user requests a change in the sort order, instead of every single time the records must be drawn in a table or list.

Looking at Records in the Librarian Sample Application

Librarian's record format is designed to use as little space as possible to store each record. Because a record in Librarian is composed mostly of variable-length strings, it would be wasteful to devote a fixed amount of space to each record, when only enough space to store each string is really required.

To achieve this kind of storage efficiency, Librarian uses the same technique employed by the built-in Address Book application. Librarian effectively has two database formats: `LibPackedDBRecord`, a packed format for actual record storage, and `LibDBRecordType`, an expanded format that is easier to access once a record has been retrieved. The `LibPackedDBRecord` structure, along with a couple structures it relies on, is defined in `librarianDB.h` as follows:

```
typedef struct {
    unsigned reserved     :1;
    unsigned bookStatus   :2;
    unsigned printStatus  :2;
    unsigned format       :2;
    unsigned read         :1;
} LibStatusType;

typedef union {
    struct {
        unsigned reserved   :7;
        unsigned note       :1;
        unsigned price      :1;
        unsigned isbn       :1;
        unsigned printing   :1;
        unsigned year       :1;
        unsigned publisher  :1;
        unsigned firstName  :1;
        unsigned lastName   :1;
        unsigned title      :1;
```

```
    } bits;
    UInt16 allBits;
} LibDBRecordFlags;

typedef struct {
    LibStatusType     status;
    LibDBRecordFlags  flags;
    unsigned char     lastNameOffset;
    unsigned char     firstNameOffset;
    unsigned char     yearOffset;
    unsigned char     noteOffset;
    char              firstField;
} LibPackedDBRecord;
```

The `LibStatusType` structure contains fixed-length information that every Librarian record must keep track of, such as whether the book is in print or in what format (hardcover or paperback) the book was printed. These pieces of information are stored in bit fields, which Librarian accesses using the following enumerated types, also defined in `librarianDB.h`:

```
// BookStatusType
// Enum for the general status of a book. Used with the
// bookStatus member of LibStatusType.

typedef enum {
    bookStatusHave = 0,
    bookStatusWant,
    bookStatusOnOrder,
    bookStatusLoaned,
    bookStatusCount
} BookStatusType;


// PrintStatusType
// Enum for the print status of a book record. Note that the
// status field is actually four bits long, but Librarian only
// makes use of three of those bits. Used with the printStatus
// member of LibStatusType.

typedef enum {
    printStatusInPrint = 0,
    printStatusOutOfPrint,
    printStatusNotPublished,
    printStatusCount
} PrintStatusType;


// FormatType
// Enum for the format of the book. Used with the format member
// of LibStatusType.
```

```
typedef enum {
    formatHardcover = 0,
    formatPaperback,
    formatTradePaperback,
    formatOther,
    formatCount
} FormatType;
```

The second field in `LibPackedDBRecord` is of type `LibDBRecordFlags`, which is a union used to keep track of which fields in a record actually contain data. The `lastNameOffset`, `firstNameOffset`, `yearOffset`, and `noteOffset` fields store the offsets of important fields from the start of `firstField` so they may be quickly accessed in place elsewhere in the application.

Starting at `firstField`, which is actually just a placeholder for the first character of the first string stored in the packed record, Librarian crams each field's data into the record, one after another, each one terminated with a trailing null character. This scheme means that the total size of a Librarian record varies widely, depending entirely on its contents. Librarian's **LibUnpackedSize** function provides an easy way to determine how big a record is:

```
static UInt32 LibUnpackedSize (LibDBRecordType *record)
{
    UInt32  size;
    Int16   i;

    // Initial size is the size of a packed record, minus the
    // character placeholder that provides the position of the
    // first field.
    size = sizeof(LibPackedDBRecord) - sizeof(char);

    // Add the length of each field that contains data, plus
    // one byte for each to accommodate a terminating null
    // character.
    for (i = 0; i < libFieldsCount; i++) {
        if (record->fields[i] != NULL)
            size += StrLen(record->fields[i]) + 1;
    }
    return size;
}
```

An unpacked record in Librarian, and the enumerated type used to access its fields, looks like this:

```
typedef enum {
    libFieldTitle = 0,
    libFieldLastName,
    libFieldFirstName,
    libFieldPublisher,
    libFieldYear,
    libFieldPrinting,
    libFieldIsbn,
```

```
    libFieldPrice,
    libFieldNote,
    libFieldsCount
} LibFields;

typedef struct {
    LibStatusType  status;
    char           *fields[libFieldsCount];
} LibDBRecordType;
```

The `LibDBRecordType` structure is much easier to work with than the packed record structure, since all the text fields are readily available through the `fields` array.

In order for Librarian to make use of this dual record structure scheme, it needs to be able to translate records between the two formats. The **PackRecord** and **UnpackRecord** functions serve this purpose. These two functions look like this:

```
static void PackRecord (LibDBRecordType *record,
                        MemPtr recordDBEntry)
{
    UInt32  offset;
    Int16   index;
    UInt16  length;
    MemPtr  p;
    LibDBRecordFlags  flags;
    LibPackedDBRecord  *packed = 0;
    Char    lastNameOffset = 0, firstNameOffset = 0,
            yearOffset = 0, noteOffset = 0;


    flags.allBits = 0;

    // Write book status structure into packed record.
    DmWrite(recordDBEntry, (UInt32)&packed->status,
            &record->status, sizeof(record->status));
    offset = (UInt32)&packed->firstField;

    for (index = 0; index < libFieldsCount; index++) {
        if (record->fields[index] != NULL) {
            p = record->fields[index];
            length = StrLen(p) + 1;

            // Write text field data to packed record.
            DmWrite(recordDBEntry, offset, p, length);
            offset += length;
            SetBitMacro(flags.allBits, index);
        }
    }

    // Write field flags to packed record.
    DmWrite(recordDBEntry, (UInt32)&packed->flags.allBits,
            &flags.allBits, sizeof(flags.allBits));
```

```
    // Set or clear field offsets, as necessary.
    index = 0;
    if (record->fields[libFieldTitle] != NULL)
        index += StrLen(record->fields[libFieldTitle]) + 1;
    if (record->fields[libFieldLastName] != NULL) {
        lastNameOffset = index;
        index += StrLen(record->fields[libFieldLastName]) + 1;
    }
    if (record->fields[libFieldFirstName] != NULL) {
        firstNameOffset = index;
        index += StrLen(record->fields[libFieldFirstName]) + 1;
    }
    if (record->fields[libFieldPublisher] != NULL)
        index += StrLen(record->fields[libFieldPublisher]) + 1;
    if (record->fields[libFieldYear] != NULL) {
        yearOffset = index;
        index += StrLen(record->fields[libFieldYear]) + 1;
    }
    if (record->fields[libFieldPrinting] != NULL)
        index += StrLen(record->fields[libFieldPrinting]) + 1;
    if (record->fields[libFieldIsbn] != NULL)
        index += StrLen(record->fields[libFieldIsbn]) + 1;
    if (record->fields[libFieldPrice] != NULL)
        index += StrLen(record->fields[libFieldPrice]) + 1;
    if (record->fields[libFieldNote] != NULL)
        noteOffset = index;

    DmWrite(recordDBEntry, (UInt32)(&packed->lastNameOffset),
            &lastNameOffset, sizeof(lastNameOffset));
    DmWrite(recordDBEntry, (UInt32)(&packed->firstNameOffset),
            &firstNameOffset, sizeof(firstNameOffset));
    DmWrite(recordDBEntry, (UInt32)(&packed->yearOffset),
            &yearOffset, sizeof(yearOffset));
    DmWrite(recordDBEntry, (UInt32)(&packed->noteOffset),
            &noteOffset, sizeof(noteOffset));
}


static void UnpackRecord (LibPackedDBRecord *packed,
                          LibDBRecordType *record)
{
    Int16   index;
    UInt16  flags;
    char    *p;


    record->status = packed->status;
    flags = packed->flags.allBits;
    p = &packed->firstField;
```

```
    for (index = 0; index < libFieldsCount; index++) {
        if (GetBitMacro(flags, index) != 0) {
            record->fields[index] = p;
            p += StrLen(p) + 1;
        }
        else {
            record->fields[index] = NULL;
        }
    }
}
```

The **PackRecord** function has a much harder job than **UnpackRecord**, since **PackRecord** must calculate the offsets of the last name, first name, year, and note fields and write these values into the packed record structure. Also, **PackRecord** must set and clear the appropriate flags in its `flags` field to indicate which text fields contain data and which are empty.

The **UnpackRecord** function needs only to look through the `flags` field of a packed record to determine which fields have data, and then iterate over the pile of strings starting at the `firstField` member of the packed record, separating the strings into the `fields` array of an unpacked record.

These two packing functions, as well as other functions in other parts of Librarian, make use of the following macros to easily retrieve, set, and clear flag bits in bit fields:

```
#define BitAtPosition(pos)                ((UInt16)1 << (pos))
#define GetBitMacro(bitfield, index)      ((bitfield) & \
                                           BitAtPosition(index))
#define SetBitMacro(bitfield, index)      ((bitfield) |= \
                                           BitAtPosition(index))
#define RemoveBitMacro(bitfield, index)   ((bitfield) &= \
                                          ~BitAtPosition(index))
```

Comparing Records

Because every application stores different kinds of data, a single Palm OS function for sorting databases is impractical. Instead, each application provides its own callback function for comparing records, which the system uses to sort the records into the proper order. Various Palm OS functions, detailed later in this chapter, use this callback to sort the database and to find the proper location to insert new records.

The application-defined callback for comparing two records in a database should have the following prototype:

```
typedef Int16 DmComparF (void* rec1, void* rec2, Int16 other,
    SortRecordInfoPtr rec1SortInfo,
    SortRecordInfoPtr rec2SortInfo, MemHandle appInfoH)
```

The first two parameters, `rec1` and `rec2`, are pointers to the two records that should be compared. Along with the information contained in each record, the system also provides other information that may be useful for sorting the records. The `other` parameter holds a value that is specific to your application. You may use `other` for any extra integer information you want to pass along when using the other Palm OS functions that require the callback function. For example, if your application allows for more than one way of sorting records, you can use the `other` parameter to specify which sort order should be assumed when comparing records. The Address Book uses this kind of scheme to allow sorting by last name or by company.

The next two parameters contain a `SortRecordInfoType` structure for each of the two records. Within the Palm OS header file `DataMgr.h`, `SortRecordInfoType` is defined as follows:

```
typedef struct {
    UInt8  attributes;
    UInt8  uniqueID[3];
} SortRecordInfoType;
```

In `SortRecordInfoType`, the `attributes` field contains the attributes for a particular record, which include the category of the record and all its status flags, such as whether or not the record is marked private. The `uniqueID` parameter contains the record's unique ID within the database.

The information in the `rec1SortInfo` and `rec2SortInfo` parameters to **DmComparF** may be useful for sorting records that might otherwise be identical according to your application's normal sorting criteria. Along with all the other information provided to **DmComparF**, the system also supplies a handle to the database's application info block in the `appInfoH` parameter.

For most applications, the `rec1SortInfo`, `rec2SortInfo`, and `appInfoH` parameters are overkill; just having pointers to the two records and the `other` parameter usually supplies more than enough information to compare two records. Still, the other three parameters might be useful if your application needs to perform more unusual sorting tasks, such as sorting records by their categories.

To indicate how the two records compare, the **DmComparF** callback must return a signed integer value `n`, where `n` is one of the following values:

- `n` < `0` if the first record should come before the second record
- `n` > `0` if the first record should come after the second record
- `0` if the two records can occupy exactly the same place in the sort order

Notice that these return values are exactly the same as the return values of the standard C function strcmp and its Palm OS cousin, StrCompare. This fact should make it easier for you to remember what to return when writing a callback comparison function.

As an example, the following implementation of **DmComparF** makes a simple string comparison between two records' `name` fields. Note that the `other`, `rec1SortInfo`, `rec2SortInfo`, and `appInfoH` parameters are completely unused in this function:

```
Int16 MyCompareFunc (void* rec1, void* rec2, Int16 other,
    SortRecordInfoPtr rec1SortInfo,
    SortRecordInfoPtr rec2SortInfo, MemHandle appInfoH)
{
    return StrCompare(rec1->name, rec2->name);
}
```

Unlike this basic example, the Librarian sample application's **LibComparePacked Records** function is quite complex. There are multiple sort orders possible in the Librarian application, and each one changes which of Librarian's record fields should be compared, and in what order, to determine the relative sort values of two records. Also, empty fields are possible in Librarian records, which further complicates matters. Listing 13-1 shows the **LibComparePackedRecords** function.

Listing 13-1: **Librarian's LibComparePackedRecords function**

```
static Int16 LibComparePackedRecords (LibPackedDBRecord *r1,
    LibPackedDBRecord *r2, Int16 showInList,
    SortRecordInfoPtr info1, SortRecordInfoPtr info2,
    MemHandle appInfoH)
{
    UInt16  whichKey1, whichKey2;
    char    *key1, *key2;
    Int16   result;


    // Records that don't contain data in the primary sort
    // field for the current sort order should be sorted before
    // records that do contain data. For example, in
    // libShowAuthorTitle sort order, any record containing
    // author data should come after a record without an
    // author.
    switch (showInList) {
        case libShowAuthorTitle:
            // Does r1 have empty author data?
            if ( (! r1->flags.bits.lastName) &&
                 (! r1->flags.bits.firstName) ) {
                // Does r2 have empty author data?
                if ( (! r2->flags.bits.lastName) &&
                     (! r2->flags.bits.firstName) )
                    // Neither r1 nor r2 contains author data,
                    // so LibComparePackedRecords needs to
                    // compare the records field by field to
                    // determine sort order.
                    break;
```

Continued

Listing 13-1 *(continued)*

```
                // r1 has no author data, r2 does have author
                // data. Therefore, r1 < r2.
                else {
                    result = -1;
                    return result;
                }
            }
            else {
                // r1 has author data, r2 does not have author
                // data. Therefore, r1 > r2.
                if ( (! r2->flags.bits.lastName) &&
                     (! r2->flags.bits.firstName) ) {
                    result = 1;
                    return result;
                }
            }
            break;

        case libShowTitleAuthor:
        case libShowTitleOnly:
            // Does r1 have empty title data?
            if (! r1->flags.bits.title) {
                // Does r2 have empty title data?
                if (! r2->flags.bits.title)
                    // Neither r1 nor r2 contains title data,
                    // so LibComparePackedRecords needs to
                    // compare the records field by field to
                    // determine sort order.
                    break;
                // r1 has no title data, r2 does have title
                // data. Therefore, r1 < r2.
                else {
                    result = -1;
                    return result;
                }
            }
            else {
                // r1 has title data, r2 does not have title
                // data. Therefore, r1 > r2.
                if (! r2->flags.bits.title) {
                    result = 1;
                    return result;
                }
            }
            break;

        default:
            break;
    }
```

```
    // Both records contain primary key data, or both records
    // have empty primary key data. Either way,
    // LibComparePackedRecords must now compare the two records
    // field by field to determine sort order.
    whichKey1 = 1;
    whichKey2 = 1;

    do {
        LibFindKey(r1, &key1, &whichKey1, showInList);
        LibFindKey(r2, &key2, &whichKey2, showInList);

        // A key with NULL loses the StrCompare.
        if (key1 == NULL) {
            // If both are NULL then return them as equal.
            if (key2 == NULL) {
                result = 0;
                return result;
            }
            else
                result = -1;
        }
        else if (key2 == NULL)
            result = 1;
        else {
            result = StrCaselessCompare(key1, key2);
            if (result == 0)
                result = StrCompare(key1, key2);
        }

    } while (! result);

    return result;
}
```

The first thing **LibComparePackedRecords** tries to determine is whether the primary key field for each record contains any data. The primary key is the field that has first priority when **LibComparePackedRecords** tries to compare two records. To determine which field is the primary key, **LibComparePackedRecords** looks at the value of its `showInList` parameter.

The **LibComparePackedRecords** function sorts empty fields before those that contain data so they will appear at the top of the list. If one of the records has an empty primary key field and the other record's primary key contains data, **LibComparePackedRecords** will have enough data to compare the records without even looking at the contents of their fields, and it will return the appropriate value and then exit.

If both records have primary key data, or if both records have empty primary keys, **LibComparePackedRecords** compares the records field by field to determine which record should come first. To accomplish this task, **LibComparePackedRecords** calls a helper function, **LibFindKey**, within a `do . . . while` loop to get the offsets of the appropriate strings within each record. The **LibFindKey** function is shown in Listing 13-2.

Listing 13-2: **Librarian's LibFindKey function**

```
static void LibFindKey (LibPackedDBRecord *record, char **key,
                        UInt16 *whichKey, Int16 showInList)
{
    LibDBRecordFlags  fieldFlags;


    fieldFlags.allBits = record->flags.allBits;

    ErrFatalDisplayIf(*whichKey == 0 || *whichKey == 6,
                      "Bad sort key");

    switch (showInList) {
        case libShowAuthorTitle:
            if (*whichKey == 1 && fieldFlags.bits.lastName) {
                *whichKey = 2;
                goto returnLastNameKey;
            }
            if (*whichKey <= 2 && fieldFlags.bits.firstName) {
                *whichKey = 3;
                goto returnFirstNameKey;
            }
            if (*whichKey <= 3 && fieldFlags.bits.title) {
                *whichKey = 4;
                goto returnTitleKey;
            }
            if (*whichKey <= 4 && fieldFlags.bits.year) {
                *whichKey = 5;
                goto returnYearKey;
            }
            break;

        case libShowTitleAuthor:
        case libShowTitleOnly:
            if (*whichKey == 1 && fieldFlags.bits.title) {
                *whichKey = 2;
                goto returnTitleKey;
            }
            if (*whichKey <= 2 && fieldFlags.bits.lastName) {
                *whichKey = 3;
                goto returnLastNameKey;
            }
```

```
            if (*whichKey <= 3 && fieldFlags.bits.firstName) {
                *whichKey = 4;
                goto returnFirstNameKey;
            }
            if (*whichKey <= 4 && fieldFlags.bits.year) {
                *whichKey = 5;
                goto returnYearKey;
            }
            break;

        default:
            break;
    }

    // All possible fields have been tried.
    *whichKey = 7;
    *key = NULL;
    return;

returnTitleKey:
    *key = &record->firstField;
    return;

returnLastNameKey:
    *key = (char *) &record->firstField +
        record->lastNameOffset;
    return;

returnFirstNameKey:
    *key = (char *) &record->firstField +
        record->firstNameOffset;
    return;

returnYearKey:
    *key = (char *) &record->firstField + record->yearOffset;
    return;
}
```

If Librarian's current sort order is `libShowAuthorTitle`, **LibFindKey** returns fields from a record in the following order: author's last name, author's first name, title of the book, year of publication. If the sort order is `libShowTitleAuthor` or `libShowTitleOnly`, **LibFindKey** uses this order instead: title of the book, author's last name, author's first name, year of publication. The **LibFindKey** function looks for the first field in this order that contains data and returns a pointer to that field's string via the `key` parameter. If **LibFindKey** cannot find any key field that contains data, it returns `NULL` in the `key` parameter. The function also advances the `whichKey` parameter to an appropriate value so that the next time **LibComparePackedRecords** calls **LibFindKey**, the search for a valid key field can start after the fields that have already been tried.

As an example of how this works, assume that `whichKey` has a value of `1` and that `showInList` is `libShowAuthorTitle`. With these values, **LibFindKey** looks for data in the record's `lastName` field. If it finds data there, the function returns a pointer to the beginning of the `lastName` string in the `key` parameter and advances `whichKey` to `2`; if `lastName` is empty, **LibFindKey** looks next in the record's `firstName` field, followed by `title`, followed by `year`, returning the first field in which it finds data, or `NULL` if they are all empty.

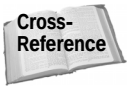
Once **LibComparePackedRecords** has isolated a key field in each record that contains data, it calls **StrCaselessCompare**, passing the strings returned from **LibFindKey**. If these two strings are unequal, **LibComparePackedRecords** returns the value returned by **StrCaselessCompare**. Otherwise, **LibComparePackedRecords** continues its `do ... while` loop, comparing the next key field in the first record to the next key field in the second record until either **StrCaselessCompare** returns a non-zero value, or all of the key fields in both records have been exhausted; in either case, the two records are equal.

Finding Records

Use the **DmFindSortPosition** function to find where a record belongs in a sorted database. Given a record that is currently unattached from a database, **DmFindSortPosition** performs a binary search to locate the record's proper position in the database. The prototype for **DmFindSortPosition** looks like this:

```
UInt16 DmFindSortPosition (DmOpenRef dbP, void* newRecord,
    SortRecordInfoPtr newRecordInfo, DmComparF *compar,
    Int16 other)
```

The `dbP` parameter is a reference to an open database, which your application can retrieve with the **DmOpenDatabaseByTypeCreator** or **DmOpenDatabase** functions.



For more information about opening databases, see Chapter 12, "Storing and Retrieving Data."

You should pass a pointer to a database record, with all its appropriate key fields filled in, via the `newRecord` parameter. The `newRecordInfo` parameter may be used to pass in extra information about the record.

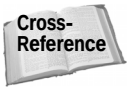
You usually will not need to use `newRecordInfo`. Simply pass the value `NULL` for this parameter if you do not wish to specify any extra record information.

The `compar` parameter is a pointer to your application's record comparison callback, and the `other` parameter allows you to send extra information to that callback function via its own `other` parameter.

Because Librarian must call **DmFindSortPosition** in a similar fashion from several locations throughout its code, Librarian has a **LibFindSortPosition** function that wraps **DmFindSortPosition**:

```
static UInt16 LibFindSortPosition (DmOpenRef db,
    LibPackedDBRecord *record)
{
    UInt8  showInList;
    LibAppInfoType  *appInfo;


    // Retrieve the current sort order from Librarian's
    // application info block.
    appInfo = MemHandleLock(LibGetAppInfo(db));
    showInList = appInfo->showInList;
    MemPtrUnlock(appInfo);

    return DmFindSortPosition(db, (MemPtr) record, NULL,
        (DmComparF *) &LibComparePackedRecords,
        (UInt16) showInList);
}
```

The **LibFindSortPosition** function reduces the number of parameters needed to find the sort position of a record to two: an open database reference, and a pointer to the record itself. Since Librarian never needs to pass extra record information to the **DmFindSortPosition** function, **LibFindSortPosition** just passes `NULL` for the `newRecordInfo` parameter. Likewise, the record comparison callback never changes, so **LibFindSortPosition** just passes the address of **LibComparePacked Records** for the `compar` parameter. The current sort order of Librarian's database does change, so **LibFindSortPosition** function looks up the current sort order in Librarian's application info block and passes it as the special value in the `other` parameter of **DmFindSortPosition**.



More details about using DmFindSortPosition, and Librarian's LibFindSort Position wrapper function, may be found later in this chapter's sections on creating and modifying records.

If you already know the unique ID of the record you want to find, you can use the **DmFindRecordByID** function to return the record's index in the database:

```
Err DmFindRecordByID (DmOpenRef dbP, UInt32 uniqueID,
                      UInt16* indexP)
```

The **DmFindRecordByID** function takes three arguments: an open database reference, the unique ID of the desired record, and a pointer to a variable that receives the index of the requested record. If for some reason **DmFindRecordByID** is unable to locate a record with the given unique ID, the function returns an error code. A successful search for the unique ID results in a `0` return value.

One last function for finding records is **DmSearchRecord**. This function looks through all open record databases for a record with a given handle and returns an open database reference and the index of the record in that database if it finds the record. The **DmSearchRecord** function has the following prototype:

```
UInt16 DmSearchRecord (MemHandle recH, DmOpenRef* dbPP)
```

The pointer to the database where the record is found is returned in the `dbPP` parameter, and the function's return value contains the index of the found record. If **DmSearchRecord** is unable to find a record, it returns `-1`, and `dbPP` is `NULL`.

Creating Records

There are two ways to create a new record in a Palm OS database. The first method allocates space for a new record with **DmNewRecord** and then writes data to the new record. The second method involves creating the record in its own memory chunk, and then attaching that chunk to the database with **DmAttachRecord**. Both methods are perfectly valid; each has its uses.

Creating records with DmNewRecord

Using **DmNewRecord** to create records works well if your application needs to view or edit the new record immediately, since **DmNewRecord** sets the *busy bit* on the record it creates. The busy bit is a flag that indicates to the data manager that a record is currently open and should be left alone. A busy record may not be opened by another application until its busy bit is cleared with the **DmReleaseRecord** function.

The **DmNewRecord** function takes three parameters: an open database reference, a pointer to the desired index for the new record, and the size of the record in bytes. If **DmNewRecord** successfully creates a new record, it returns a handle to the new record. The prototype for **DmNewRecord** looks like this:

```
MemHandle DmNewRecord (DmOpenRef dbP, UInt16* atP, UInt32 size)
```

When you call **DmNewRecord**, the `atP` parameter should point to a variable containing the index where you would like to insert the record; when **DmNewRecord** returns, it replaces the contents of what `atP` points to with the actual index of the newly created record. Record indices range from `0` to the total number of records minus one. If you specify `0` as the index for the new record, **DmNewRecord** adds the record at the beginning of the database:

```
MemHandle  newRecordH;
UInt16     index = 0;

newRecordH = DmNewRecord(gDB, &index, size);
```

If the index number is greater than the number of records in the database, **DmNewRecord** adds the record to the end of the database. You can ensure that a record is appended to the end of the database by using the constant `dmMaxRecordIndex`:

```
UInt16  index = dmMaxRecordIndex;

newRecordH = DmNewRecord(gDB, &index, size);
```

After the call above, `index` contains the actual index of the new record.

Usually, you should not add records to the end of the database, because that is where the data manager keeps deleted and archived records. Adding records to the end of the database can confuse record-sorting functions like DmFindSortPosition, which assumes that deleted records always have a higher index than undeleted records.

Most of the time, you will want to add a new record at its proper sort position in the database. Use the **DmFindSortPosition** function to find where the new record belongs, and then create the new record using **DmNewRecord**:

```
UInt16        index;
MyRecordType  newRecord;
MemHandle     newRecordH;
MyRecordType  *newRecordP;


// Initialize the fields of the newRecord structure.

index = DmFindSortPosition(gDB, &newRecord, NULL,
                           (DmComparF *) MyCompareFunc, NULL);
newRecordH = DmNewRecord(gDB, &index, sizeof(newRecord));
newRecordP = MemHandleLock(newRecordH);
DmWrite(newRecordP, 0, &newRecord, sizeof(newRecord));
MemHandleUnlock(newRecordH);
DmReleaseRecord(gDB, index, true);
```

This example goes one step further than previous examples and actually writes the new record's data into the database with **DmWrite**. The **DmWrite** function has four parameters: a pointer to a locked chunk of storage memory, the offset from the beginning of that memory where **DmWrite** should start writing data, a pointer to the data to write, and the size of the data in bytes. In the foregoing example, the **DmWrite** function writes the entire `newRecord` structure to memory.

After you are finished reading from and writing to a record created with **DmNewRecord**, call **DmReleaseRecord** to clear the busy bit on the new record. The **DmReleaseRecord** function has three parameters: an open database reference, the index of the record to release, and a `Boolean` value to indicate whether the record should be marked dirty or not. Passing a `true` value for the **DmReleaseRecord** function's last parameter sets the record's *dirty bit*, and a `false` value leaves the dirty bit alone.

A record's dirty bit is important during a HotSync operation. The HotSync Manager uses the dirty bit to determine whether a record has changed or not. Be sure your application sets the dirty bit if it changes a record; not setting the dirty bit can cause an application's conduit to improperly synchronize a record, since the conduit then has no indication that the record has changed.

Creating records with DmAttachRecord

The second method for creating a new database record first creates the record in its own independent storage memory chunk and then attaches that chunk to a database using the **DmAttachRecord** function. This method works well if your application does not view or edit the new record immediately, since **DmAttach Record** does not set the busy bit on the new record. Likewise, it is not necessary to remember to call **DmReleaseRecord** after creating a new record using the **DmAttachRecord** function.

To start creating a new record with **DmAttachRecord**, you must first allocate a chunk of storage memory and fill it with the new record's data. Use the **DmNewHandle** function to accomplish this task:

```
MyRecordType  newRecord;
MemHandle     newRecordH;
MyRecordType  *newRecordP;
UInt16        index;
Err           error;

// Initialize the fields of the newRecord structure.

newRecordH = DmNewHandle(gDB, sizeof(newRecord));
newRecordP = MemHandleLock(recordH);
DmWrite(newRecordP, 0, &newRecord, sizeof(newRecord));
```

After the code above executes, you have a free-floating chunk of memory containing the new record. To attach this chunk to the database, first find where the record belongs using **DmFindSortPosition**, and then attach the record at that position using **DmAttachRecord**:

```
index = DmFindSortPosition(gDB, &newRecord, NULL,
                           (DmComparF *) MyCompareFunc, NULL);
MemHandleUnlock(recordH);
error = DmAttachRecord(gDB, &index, newRecordH, NULL);

// If all went well, index now contains the actual index where
// the record was inserted into the database.

if (error)
    MemHandleFree(newRecordH);
```

The **DmAttachRecord** function has four parameters: an open database reference, a pointer to a variable containing the desired index for the new record, the handle of the new record, and a pointer to another record handle. The last parameter of **DmAttachRecord** may be used to replace an existing record in the database. If you pass a pointer to a handle for the last parameter, **DmAttachRecord** replaces the record at the requested index with the new record and returns a handle to the old record via this pointer; see the "Modifying Records" section later in this chapter for an example. A `NULL` value for the fourth parameter tells **DmAttachRecord** to insert the record instead of replacing an existing one, in much the same way that **DmNewRecord** works. Also similar to **DmNewRecord** is the way **DmAttachRecord** deals with its index parameter; **DmAttachRecord** returns the actual index of the new record via its index parameter.

The Librarian sample application uses the **DmAttachRecord** method to add new records to its database, and it wraps the whole new record creation process in the function **LibNewRecord**:

```
Err LibNewRecord (DmOpenRef db, LibDBRecordType *record,
                  UInt16 *index)
{
    MemHandle  recordH;
    Err        error;
    LibPackedDBRecord  *packed;
    UInt16     newIndex;


    // Allocate a chunk large enough to hold the new packed
    // record.
    recordH = DmNewHandle(db, LibUnpackedSize(record));
    if (recordH == NULL)
        return dmErrMemError;

    // Copy the data from the unpacked record to the packed
    // one.
    packed = MemHandleLock(recordH);
    PackRecord(record, packed);

    // Get the index of the new record.
    newIndex = LibFindSortPosition(db, packed);
    MemPtrUnlock(packed);

    // Attach new record in place and return the index of the
    // new record in the index parameter.
    error = DmAttachRecord(db, &newIndex, recordH, 0);
    if (error)
        MemHandleFree(recordH);
    else
        *index = newIndex;

    return error;
}
```

Deleting Records

Deleting records from a Palm OS database requires a bit of finesse, because the data manager expects to find deleted and archived records at the end of the database. In an application with a conduit, deleted records need to stick around on the handheld until the next HotSync operation, at which point the application's conduit can delete the corresponding records from the desktop version of the application's database. Likewise, archived records need to remain on the handheld until a HotSync operation can allow the conduit to properly archive a record on the desktop before entirely removing it from the handheld.

Because of these deletion and archival requirements, the Palm OS provides three functions for deleting records from a database, appropriate for different circumstances: **DmRemoveRecord**, **DmDeleteRecord**, and **DmArchiveRecord**.

The first record deletion function, **DmRemoveRecord**, actually deletes a record outright, without giving the application's conduit a chance to look at the deleted record. This function is appropriate when the user creates a record on the handheld but then immediately deletes it. In this case, there is no corresponding record on the desktop, so there is no need to keep the record at the end of the database until the next HotSync operation. The **DmRemoveRecord** function needs only an open database reference and the index of the record to remove:

```
Err error = DmRemoveRecord(gDB, index);
```

The **DmDeleteRecord** function frees the memory chunk associated with a record's data but marks the record as deleted in the database header. A similar function, **DmArchiveRecord**, frees a record's data chunk and sets the record's archive bit in the database header. Just as with **DmRemoveRecord**, these two functions require an open database reference and the index of the record to delete or archive.

You must be sure your application moves deleted and archived records to the end of the database itself using the **DmMoveRecord** function; the system does not do this important step for you. The following code takes care of deleting or archiving a record and moving it to the end of the database:

```
if (gArchive)
    DmArchiveRecord(gDB, index);
else
    DmDeleteRecord(gDB, index);
DmMoveRecord(gDB, index, DmNumRecords(gDB));
```

The last parameter of the **DmMoveRecord** indicates the index where a record should be moved to, which causes the indices of all the records following this insertion point to increase by one. You may pass a value one greater than the index of the last record to move a record to the end of the database. The example above uses the function **DmNumRecords** to determine the number of records in the database and uses that function's return value as the index to which a deleted or archived record should be moved (recall that record indices are zero-based).

An accompanying function, **DmDetachRecord**, allows you to deliberately orphan a database record by removing its entry in the database header but leaving its data chunk intact. The prototype for **DmDetachRecord** looks like this:

```
Err DmDetachRecord (DmOpenRef dbP, UInt16 index,
    MemHandle* oldHP)
```

The first two parameters to **DmDetachRecord** take an open database reference and the index of the record you would like to detach. If you supply a pointer to a memory handle in the third parameter, **DmDetachRecord** returns the handle of the detached record in this pointer. The **DmDetachRecord** function is a good way to start moving a record from one database to another. Use **DmDetachRecord** to cut the record from the first database, and then attach it to the second database using **DmAttachRecord**.

Librarian encapsulates the code it needs to delete records in its **DeleteRecord** function, shown below:

```
static void DeleteRecord (Boolean archive)
{
    // Show the prior record. This provides context for the
    // user, as it shows where the record was, and it allows a
    // return to the same location in the database if the user
    // is working through the records sequentially. If there
    // isn't a prior record, show the following record. If
    // there isn't a following record, don't show a record at
    // all.
    gListFormSelectThisRecord = gCurrentRecord;
    if (! SeekRecord(&gListFormSelectThisRecord, 1,
        dmSeekBackward))
        if (! SeekRecord(&gListFormSelectThisRecord, 1,
            dmSeekForward))
            gListFormSelectThisRecord = noRecord;

    // Delete or archive the record.
    if (archive)
        DmArchiveRecord(gLibDB, gCurrentRecord);
    else
        DmDeleteRecord(gLibDB, gCurrentRecord);

    // Deleted records are stored at the end of the database.
    DmMoveRecord(gLibDB, gCurrentRecord, DmNumRecords(gLibDB));

    // Since we just moved the gCurrentRecord to the end, the
    // gListFormSelectThisRecord may need to be moved up one
    // position.
    if (gListFormSelectThisRecord >= gCurrentRecord &&
        gListFormSelectThisRecord != noRecord)
        gListFormSelectThisRecord--;

    // Use whatever record we found to select.
    gCurrentRecord = gListFormSelectThisRecord;
}
```

Librarian keeps track of the current record via a global variable, `gCurrentRecord`. When the user deletes the current record, **DeleteRecord** ensures that `gCurrentRecord` is set to an appropriate value, because deleting a record makes it invalid for display purposes. Likewise, Librarian also keeps track of which record in its List view should be highlighted using the global `gListFormSelectThisRecord`; **DeleteRecord** also ensures that this global variable is updated properly. The **SeekRecord** function used in **DeleteRecord** is described later in this chapter, in the section "Categorizing Records."

Reading Records

If your application needs to read values from a record without writing to it, the data manager allows the application to lock a handle to the record without marking it busy by using the **DmQueryRecord** function. Calling **DmQueryRecord** looks like this:

```
MemHandle recordH = DmQueryRecord(gDB, index);
MyRecordType *recordP = MemHandleLock(recordH);
// Read values from recordP here.
MemHandleUnlock(recordH);
```

Modifying Records

Just as there is more than one way to create records, there is more than one way to open a record so that your application may write to it. The first requires **DmGetRecord** to retrieve a handle to the record, marking the record busy in the process. The second method allocates a completely new memory chunk for the changed record with **DmNewHandle**, copies the old values from the record to the new chunk with **DmQueryRecord**, modifies the new chunk's values, and then replaces the original record with the new one using **DmAttachRecord**. A third method, useful for text data displayed in fields in a table, relies on the field manager's **FldSetText** function to allow the user to edit the data in place.

Modifying records with DmGetRecord

The **DmGetRecord** function marks a record as busy so that no other applications can mess with the record while your application is modifying it. Just as when you're creating a new record with **DmNewRecord**, you must call **DmReleaseRecord** to clear the record's busy bit after you are done making changes, and to set the record's dirty bit if necessary. Also, because all memory in the storage area is protected, you may write to it using the **DmWrite** function only, or **DmSet** to set a memory range to a particular value.

Changing a record with **DmGetRecord** requires the following steps:

1. Create a temporary record structure and copy the original record into it.
2. Change the temporary record structure's fields to their new values.
3. Open the record with **DmGetRecord**.

4. Copy the changes from the temporary record to the actual record using **DmWrite**.
5. Check to see if changes to the record have altered its sort position within the database. If so, release the record and mark it dirty with **DmReleaseRecord**, and then move the record to its proper index with **DmMoveRecord**.
6. If the record did not change position, release it and mark it dirty with **DmReleaseRecord**.

The following function takes an open database reference and a pointer to the index of a record and changes the record according to the steps outlined above:

```
Err ChangeRecord (DmOpenRef db, UInt16 *index)
{
    MemHandle     recordH;
    MyRecordType  tempRecord;
    MyRecordType  *record;
    MyRecordType  *cmp;
    UInt16        attributes;
    Boolean       move = true;
    Int16         i;

    recordH = DmGetRecord(db, *index);
    if (recordH == NULL)
        return DmGetLastErr();
    record = MemHandleLock(recordH);

    // Copy the values from the actual record to a temporary
    // record.
    tempRecord = *record;

    // Modify the temporary record here.
    tempRecord.field = newValue;

    // Copy the modified temporary record into the actual
    // storage space of the real record.
    DmWrite(record, 0, &tempRecord, sizeof(tempRecord));

    // Determine if the record is in the proper sort order.
    if (*index > 0) {
        // Compare this record to the record before it.
        cmp = MemHandleLock(DmQueryRecord(db, *index - 1));
        move = (MyCompareFunc(cmp, record, 0, NULL, NULL,
                              NULL) > 0);
        MemPtrUnlock(cmp);
    } else {
        move = false;
    }

    if (*index + 1 < DmNumRecords(db)) {
        // Be sure not to move the record beyond the deleted
        // records at the end of the database.
```

```
            DmRecordInfo(db, *index + 1, &attributes, NULL, NULL);
            if (! (attributes & dmRecAttrDelete)) {
                // Compare this record to the record after it.
                cmp = MemHandleLock(DmQueryRecord(db, *index + 1));
                move = (! move) && (MyCompareFunc(record,
                                    cmp, 0, NULL, NULL, NULL) > 0);
                MemPtrUnlock(cmp);
            }
        }

        if (move) {
            // The record isn't in the right position, so move it.
            i = DmFindSortPosition(db, record, NULL,
                                   &MyCompareFunc, 0);

            // Unlock and release the record before moving it.
            MemHandleUnlock(recordH);
            DmReleaseRecord(db, index, true);

            DmMoveRecord(db, *index, i);
            if (i > *index)
                i--;
            *index = i;  // Return new record database position.
        } else {
            MemHandleUnlock(recordH);
            DmReleaseRecord(db, index, true);
        }

        return 0;
    }
```

If you have changed any key fields that your application uses to sort records, use **DmMoveRecord** to put the changed record into its proper sort order within the database. By calling the application-defined **MyCompareFunc** callback function directly, the **ChangeRecord** function determines whether the modified record should be moved. First, the example compares the changed record with the record before it, which is the record located at `*index - 1`. Then, **ChangeRecord** compares the modified record with the record following it, at `*index + 1`. If the record is out of place, **ChangeRecord** finds the record's proper sort position using **DmFindSortPosition**, and then moves the record to the new location with **DmMoveRecord**. Afterward, **ChangeRecord** modifies its `index` parameter to reflect the altered record's new index.

You can also set individual fields in a record without having to copy the entire record structure by using the standard C `offsetof` macro, which returns the offset of a field within a structure. This approach is a bit more efficient when you need to modify only a single value in a record:

```
DmWrite(recordP, offsetof(MyRecordType, field),
        &newValue, sizeof(newValue));
```

In fact, this method may be required for writing large records. Because stack space is very limited in the Palm OS (less than 4KB on some hardware), copying an entire large record into a temporary structure may not be an option. In this case, you can use the offset to write the record's data a small piece at a time.

Modifying records with DmAttachRecord

A slightly different approach to modifying records involves the **DmAttachRecord** function. Instead of modifying an existing record, this method allocates a brand new chunk of storage memory to contain the changed record. Here are the steps to follow when modifying a record using **DmAttachRecord**:

1. Create a temporary record structure and copy the original record into it.
2. Change the temporary record structure's fields to their new values.
3. Allocate a chunk of memory with **DmNewHandle** to hold the changed record.
4. Copy the temporary record into the new chunk of memory. You now have an orphaned record, floating freely in storage RAM.
5. Check to see if changes to the record have altered its sort position within the database. If so, move the original record to the correct index with **DmMoveRecord**.
6. Replace the original record with the new memory chunk using **DmAttachRecord**.
7. Dispose of the original record, which has now become an orphaned chunk of memory.

The **ChangeRecord2** function below performs exactly the same function as the **ChangeRecord** function from the previous section, but using the **DmAttachRecord** method outlined above instead of **DmGetRecord**:

```
Err ChangeRecord2 (DmOpenRef db, UInt16 *index)
{
    Err           result;
    MemHandle     recordH, changedRecordH, oldH;
    MyRecordType  tempRecord;
    MyRecordType  *record;
    MyRecordType  *changedRecord;
    MyRecordType  *cmp;
    UInt16        attributes;
    Boolean       move = true;
    Int16         i;

    recordH = DmQueryRecord(db, *index);
    record = MemHandleLock(recordH);

    // Copy the values from the actual record to a temporary
    // record.
    tempRecord = *record;
```

```
// The original record has been copied and is no longer
// needed.
MemHandleUnlock(recordH);

// Modify the temporary record here.
tempRecord.field = newValue;

// Allocate a chunk for the changed record.
changedRecordH = DmNewHandle(db, sizeof(tempRecord));
if (changedRecordH == NULL) {
    MemHandleUnlock(recordH);
    return dmErrMemError;
}
changedRecord = MemHandleLock(changedRecordH);

// Copy the modified temporary record into the new
// memory chunk.
DmWrite(changedRecord, 0, &tempRecord, sizeof(tempRecord));

// Make sure the record is in the proper sort order.
if (*index > 0) {
    // Compare this record to the record before it.
    cmp = MemHandleLock(DmQueryRecord(db, *index - 1));
    move = (MyCompareFunc(cmp, changedRecord, 0, NULL,
                          NULL, NULL) > 0);
    MemPtrUnlock(cmp);
} else {
    move = false;
}

if (*index + 1 < DmNumRecords(db)) {
    // Be sure not to move the record beyond the deleted
    // records at the end of the database.
    DmRecordInfo(db, *index + 1, &attributes, NULL, NULL);
    if (! (attributes & dmRecAttrDelete)) {
        // Compare this record to the record after it.
        cmp = MemHandleLock(DmQueryRecord(db, *index + 1));
        move = (! move) && (MyCompareFunc(changedRecord,
                            cmp, 0, NULL, NULL, NULL) > 0);
        MemPtrUnlock(cmp);
    }
}

if (move) {
    // The record isn't in the right position, so move it.
    i = DmFindSortPosition(db, changedRecord);
    DmMoveRecord(db, *index, i);
    if (i > *index)
        i--;
    *index = i;  // Return new record database position.
}
```

```
    // Replace the original record, now located in its proper
    // sort order, with the memory chunk containing the changed
    // record.
    result = DmAttachRecord(db, index, changedRecordH, &oldH);
    MemHandleUnlock(changedRecordH);
    if (result) return result;

    // The original record is now a detached orphan, so its
    // memory may be freed.
    MemHandleFree(oldH);
    return 0;
}
```

Since the method above never marks the record to change as busy with **DmGetRecord**, and the **DmAttachRecord** function sets the dirty bit on a newly attached record, there is no need to call **DmReleaseRecord** when done to clear the busy bit or set the dirty bit. The **DmReleaseRecord** call in **ChangeRecord2** also returns a handle, `oldH`, to the original record; this example does not need to do anything with the original record once it has been replaced, and so **ChangeRecord2** disposes of the handle with **MemHandleFree**.

The DmAttachRecord function is very handy for cutting and pasting between two databases.

The Librarian sample application uses the **DmAttachRecord** method for committing changes to a record in its database. Librarian's **LibChangeRecord** function, shown below, takes care of all the necessary work:

```
Err LibChangeRecord (DmOpenRef db, UInt16 *index,
    LibDBRecordType *record, LibDBRecordFlags changedFields)
{
    LibDBRecordType  src;
    MemHandle  srcH;
    Err        result;
    MemHandle  recordH = 0;
    MemHandle  oldH;
    Int16      i;
    UInt32     changes = changedFields.allBits;
    Int16      showInList;
    LibAppInfoType  *appInfo;
    Boolean    move = true;
    UInt16     attributes;
    LibPackedDBRecord*  cmp;
    LibPackedDBRecord*  packed;


    // LibChangeRecord does not assume that record is
    // completely valid, so it retrieves a valid pointer to the
    // record.
    if ((result = LibGetRecord(db, *index, &src, &srcH)) != 0)
        return result;
```

```
// Apply the changes to the valid record.
src.status = record->status;
for (i = 0; i < libFieldsCount; i++)  {
    // If the flag is set, point to the string, otherwise
    // point to NULL.
    if (GetBitMacro(changes, i) != 0) {
        src.fields[i] = record->fields[i];
        RemoveBitMacro(changes, i);
    }
    if (changes == 0)
        break;      // no more changes
}

// Make a new chunk with the correct size.
recordH = DmNewHandle(db, LibUnpackedSize(&src));
if (recordH == NULL) {
    MemHandleUnlock(srcH);
    return dmErrMemError;
}
packed = MemHandleLock(recordH);

// Copy the data from the unpacked record to the packed
// record.
PackRecord(&src, packed);

// The original record is copied and no longer needed.
MemHandleUnlock(srcH);

// Check if any of the key fields have changed. If they
// have not changed, this record is already in its proper
// place in the database, and LibChangeRecord can skip
// re-sorting the record.
if ((changedFields.allBits & sortKeyFieldBits) == 0)
    move = false;

// Make sure *index - 1 < *index < *index + 1; if so, the
// record is already in sorted order. Deleted records are
// stored at the end of the database, so LibChangeRecord
// must also make sure not to sort this record past the end
// of any deleted records.
if (move) {
    appInfo = MemHandleLock(LibGetAppInfo(db));
    showInList = appInfo->showInList;
    MemPtrUnlock(appInfo);

    if (*index > 0) {
        // Compare this record to the record before it.
        cmp = MemHandleLock(DmQueryRecord(db, *index - 1));
        move = (LibComparePackedRecords(cmp, packed,
                showInList, NULL, NULL, NULL) > 0);
        MemPtrUnlock(cmp);
    } else {
        move = false;
    }
```

```
            if (*index + 1 < DmNumRecords(db)) {
                // Be sure not to move the record beyond the
                // deleted records at the end of the database.
                DmRecordInfo(db, *index + 1, &attributes, NULL,
                             NULL);
                if (! (attributes & dmRecAttrDelete)) {
                    // Compare this record to the record after it.
                    cmp = MemHandleLock(DmQueryRecord(db,
                                        *index + 1));
                    move = (! move) &&
                        (LibComparePackedRecords(packed, cmp,
                         showInList, NULL, NULL, NULL) > 0);
                    MemPtrUnlock(cmp);
                }
            }
        }

        if (move) {
            // The record isn't in the right position, so move it.
            i = LibFindSortPosition(db, packed);
            DmMoveRecord(db, *index, i);
            if (i > *index)
                i--;
            *index = i;  // Return new record database position.
        }

        // Attach the new record to the old index, which preserves
        // the category and record ID.
        result = DmAttachRecord(db, index, recordH, &oldH);
        MemPtrUnlock(packed);
        if (result) return result;

        MemHandleFree(oldH);
        return 0;
    }
```

Note that because **LibChangeRecord** receives a `LibDBRecordFlags` structure (`changedFields`) that indicates what fields in the record have changed, **LibChangeRecord** saves itself a lot of work by looking in `changedFields` for changed key fields before checking whether the record is in the right position or not. If none of the key fields that Librarian uses to sort its data have been changed, then the record is still in its proper sort order and does not need to be moved. The **sortKeyFieldBits** macro that **LibChangeRecord** uses to make this decision is defined as follows in `librarianDB.c`:

```
#define BitAtPosition(pos)  ((UInt16)1 << (pos))
#define sortKeyFieldBits  (BitAtPosition(libFieldTitle) | \
                           BitAtPosition(libFieldLastName) | \
                           BitAtPosition(libFieldFirstName) | \
                           BitAtPosition(libFieldYear) )
```

Editing records in place

A field object can be used to edit a string in place within a record. In-place editing is particularly useful in large text fields that may contain a lot of data, such as the Note view in the various ROM applications. Less dynamic memory is required to edit a large field in place than to copy all the data from the field to a database record.

With a little bit of setup, the field manager takes care of resizing the handle containing the string as the user edits the text in the field. This method even works for string values that are stored within a larger structure in an application's records; you simply need to pass an offset from the start of the record where the string data begins.

To illustrate editing in place, consider the following record structure:

```
typedef struct {
    Int16  data1;
    Int32  data2;
    Char   stringData[20];
} MyRecordType;
```

The text string stored in `stringData` is a null-terminated string and may be of any length, not just 20 characters. The following code sets up a text field for in-place editing of the `stringData` field:

```
FormType   *form;
FieldType  *field;
MemHandle  recordH;
MemHandle  oldTextH;
MyRecordType  *record;

form = FrmGetActiveForm();
field = FrmGetObjectPtr(form, FrmGetObjectIndex(form,
                                                MyField));
oldTextH = FldGetTextHandle(field);

// Dispose of the old handle to prevent a memory leak.
if (oldTextH)
    MemHandleFree(oldTextH);

recordH = DmGetRecord(gDB, index);
record = MemHandleLock(recordH);
FldSetText(field, recordH, offsetof(MyRecordType, stringData),
    StrLen(record.stringData) + 1);

MemHandleUnlock(recordH);
```

Now the `MyField` text field is set up for in-place editing of a record's `stringData` field. You can actually dispense with calling **MemHandleFree** to free up the memory occupied by the field's old text handle if you are setting up in-place editing on a form that has just been opened, because a text field that has just been initialized does not have a text handle allocated for it yet; just in case, though, it's never a bad idea to dispose of the old handle, as this will prevent a memory leak if the field did, indeed, have a text handle before your application called **FldGetTextHandle**.

Once editing is finished, you must perform a few cleanup tasks. Normally, you would call something similar to the following code when closing the form that contains the text field you are working with:

```
Boolean  dirty = FldDirty(field);

if (dirty)
    FldCompactText(field);
FldSetTextHandle(field, NULL);
DmReleaseRecord(gDB, index, dirty);
```

The first thing you should do is to compact the handle containing the text if the field has been modified. Since the field manager resizes the text handle several bytes at a time instead of one byte at a time, there might be more space allocated for the string than is actually required. Call **FldCompactText** to trim the handle down to the proper size so you are not wasting storage space with unnecessary empty bytes.

Once you have compacted the text handle, disconnect the handle from the field by calling **FldSetTextHandle** with `NULL` as its second parameter. The system frees the memory allocated for a field's text handle when disposing of the field. Since the text handle in this case is actually the data stored in the application's database, freeing this memory would erase the data from this particular record. Disconnecting the text handle from the field with **FldSetTextHandle** prevents this data loss.

Finally, you must release the record so the system can clear its busy bit. The **DmReleaseRecord** function serves this purpose here, also setting the dirty bit for the record, if necessary.

The in-place editing technique cannot connect more than one text field to multiple strings in a database record. If two or more fields were hooked up to the same record, they might both try to resize the handle, and the field manager is not set up to deal with that situation.

Librarian uses the in-place editing technique in its Note view to allow in-place editing of a large field. The **NoteViewLoadRecord** function takes care of setting up in-place editing in the Note view:

```
static void NoteViewLoadRecord (void)
{
    FieldType  *field;
    LibPackedDBRecord  *packed;
    MemHandle  packedH;
    Char       *ptr;
    UInt16     offset;


    // Get a pointer to the memo field.
    field = GetObjectPtr(NoteField);

    // Set the font used in the memo field.
    FldSetFont(field, gNoteFont);

    // Retrieve the note field from the current database
    // record. Librarian calls CreateNote before getting to
    // NoteViewLoadRecord, which guarantees that the note field
    // already exists.
    packedH = DmQueryRecord(gLibDB, gCurrentRecord);
    ErrFatalDisplayIf((! packedH), "Bad record");
    packed = MemHandleLock(packedH);

    // Set a pointer to the location of the note field, using
    // the note field offset stored in the packed database
    // record.
    ptr = &packed->firstField;
    ptr += packed->noteOffset;

    // Calculate the offset of the note field from the front of
    // the packed database record, not from the beginning of
    // the first field, since FldSetText wants the offset from
    // the start of the record's memory chunk.
    offset = ptr - (char *)packed;

    // Set the note field text to the contents of the note
    // field.
    FldSetText(field, packedH, offset, StrLen(ptr) + 1);

    MemHandleUnlock(packedH);
}
```

Notice that **NoteViewLoadRecord** does not use the **offsetof** macro to determine the offset of the record's note field. Since Librarian uses variable-length records, and there are a number of strings of different lengths that may or may not exist before the note field in a record, a simple **offsetof** call will not return the correct location of the note field. Instead, **NoteViewLoadRecord** uses the record structure's `noteOffset` field to determine where the note field begins.

The **NoteViewSave** function takes care of in-place editing cleanup in Librarian:

```
static void NoteViewSave (void)
{
    FieldType  *field;
    UInt16     length;


    field = GetObjectPtr(NoteField);

    // If the field wasn't modified then don't do anything.
    if (FldDirty(field)) {
        // Release any free space in the note field.
        FldCompactText(field);
        DirtyRecord(gCurrentRecord);
    }

    length = FldGetTextLength(field);

    // Clear the handle value in the field, otherwise the
    // handle will be free when the form is disposed of.
    // This call also unlocks the handle that contains the
    // note string.
    FldSetTextHandle(field, 0);

    // Empty fields are not allowed because they cause
    // problems.
    if (length == 0)
        DeleteNote();
}
```

Because of Librarian's record structure, an empty string does not work well in one of its fields, since it would still contain a single terminating null character. Other routines in Librarian expect that an empty field will take up zero space in a record, and leaving the trailing null of an empty string in the record would cause problems later, so **NoteViewSave** deletes the note entirely with the **DeleteNote** function if the note's text field is empty.

Sorting Records

The Palm OS provides two functions for sorting all the records in a database: **DmInsertionSort** and **DmQuickSort**. As its name implies, **DmInsertionSort** performs an insertion sort of a database's records, which works its way through the database one record at a time, comparing each record with the preceding record and inserting it into its proper sorted position if it is less than the preceding record. The **DmInsertionSort** algorithm is very fast on a database that is already mostly sorted, or on a small database containing about 20 or fewer records. In a situation where your application has moved a single record out of place, **DmInsertionSort** would be ideal to place the database back in order. To call **DmInsertionSort**, pass

an open database reference, a pointer to a **DmComparF** callback comparison function, and any special value the callback might need to perform its job:

```
Err error = DmInsertionSort(gDB, &MyCompareFunc, other);
```

The **DmQuickSort** function sorts records using a quicksort algorithm, which sorts the database by partitioning the records repeatedly. Use **DmQuickSort** when you don't know how sorted the database is. For example, changing the sort order of a database from sorting by name to sorting by date (a different field in the database record) is a good time to use **DmQuickSort**. Call **DmQuickSort** the same way you call **DmInsertionSort**:

```
Err error = DmQuickSort(gDB, &MyCompareFunc, other);
```

Keep in mind that the sort performed by **DmInsertionSort** is stable, whereas **DmQuickSort** performs an unstable sort. In a stable sort, two records that compare as the same value maintain their relative positions in the database; an unstable sort may cause equal records to change positions. Also, **DmQuickSort** can work with a maximum of 16K records, whereas **DmInsertionSort** can handle up to 32K records.

Retrieving and Modifying Record Information

Each record in a database has a number of properties related to it that are stored in the record list in the database's header. You can retrieve this information using the **DmRecordInfo** function:

```
UInt16   attributes;
UInt16   uniqueID;
LocalID  chunkID;
Err      error;

error = DmRecordInfo(db, index, &attributes, &uniqueID,
                     &chunkID);
```

The **DmRecordInfo** function returns the record's unique ID, as well as the LocalID of the chunk where the record's data resides. Usually, the best way to modify a record is to retrieve a handle to it using **DmGetRecord** or **DmQueryRecord**, but the unique ID and LocalID are available through **DmRecordInfo** should you need them.

You can pass `NULL` for the `attributes`, `uniqueID`, or `chunkID` parameters of DmRecordInfo if you want to ignore any of these values.

The `attributes` for a record include flags for the record's deleted, dirty, busy, and secret bits, along with half a byte containing the record's category. The Palm OS header file `DataMgr.h` provides the following handy constants for accessing the flags and masking the category in this bit field:

```
#define dmRecAttrCategoryMask  0x0F
#define dmUnfiledCategory      0
```

```
#define dmRecAttrDelete         0x80
#define dmRecAttrDirty          0x40
#define dmRecAttrBusy           0x20
#define dmRecAttrSecret         0x10
```

Even though the DmRecordInfo function returns a pointer to an unsigned 16-bit integer containing a record's attributes, current implementations of the Palm OS use only half of this space (one byte) to store the attributes. As with any implementation-specific details of the Palm OS, you should not rely on the size of specific structures used by the system. Code defensively; always use the Palm OS API functions and the appropriate constants in the Palm OS headers when dealing with record attributes.

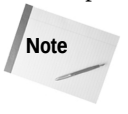
For example, the following code retrieves the current deletion status of a record:

```
UInt16  attributes;

DmRecordInfo(db, index, &attributes, NULL, NULL);
Boolean deleted = attributes & dmRecAttrDelete;
```

You can set record information with the **DmSetRecordInfo** function. Everything available through **DmRecordInfo** may be set, except for the LocalID of the record's data chunk:

```
Err error = DmSetRecordInfo(db, index, &attributes, &uniqueID);
```

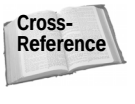
Passing `NULL` for the `attributes` or `uniqueID` parameters leaves that particular piece of record information alone.



As a general rule, leave the unique ID of a record alone. The data manager automatically creates a unique ID when you create a record with DmNewRecord, so applications usually do not have to change a record's unique ID.

The following example sets a record's secret bit, which controls whether or not a record is considered private:

```
UInt16 attributes;

DmRecordInfo(db, index, &attributes, NULL, NULL);
attributes |= dmRecAttrSecret;
DmSetRecordInfo(db, index, &attributes, NULL);
```



See the sections "Setting a record's category," "Selecting and modifying categories," and "Implementing Private Records" later in this chapter for more examples of how to use DmRecordInfo and DmSetRecordInfo.

Categorizing Records

Every record in a Palm OS database has an attributes field, which contains half a byte of category information about the record. This mechanism allows you to provide user-customizable categories for classifying records, in the same way that the Address Book, To Do List, and Memo Pad built-in applications do. The Palm OS provides a category manager to make managing categories easier.

Initializing categories

In order to provide some default categories in your application, such as the "Business," "Personal," and "Unfiled" categories that are common in the built-in applications, you will need to initialize those categories in the application info block. The best place to store these default category names is in an app info string resource, which you create with either Constructor or PilRC, depending on your development environment.

Librarian has the default categories "Fiction," "Nonfiction," and "Unfiled." Figure 13-1 shows Librarian's category app info string, as defined in Constructor. Notice that "Unfiled" is listed first. Any categories that you do not want the user to edit or remove must appear first in the app info string.

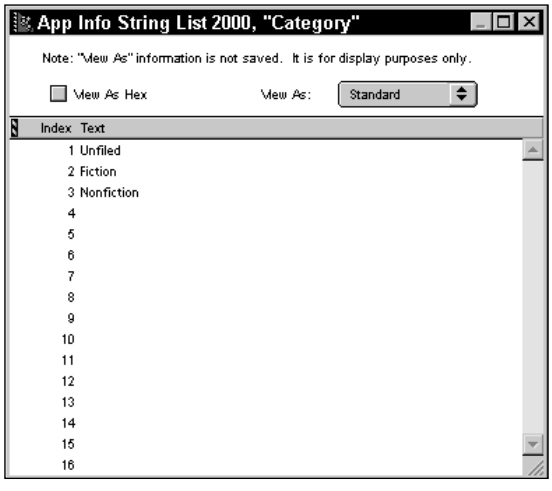


Figure 13-1: Librarian's category app info string in Constructor

The PilRC definition for Librarian's categories looks like this:

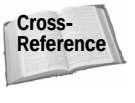
```
CATEGORIES ID CategoryAppInfoStr
    "Unfiled"
    "Fiction"
    "Nonfiction"
    ""
    ""
    ""
    ""
    ""
    ""
    ""
    ""
    ""
    ""
    ""
    ""
```

Once you have default categories defined in your application's resources, you can initialize the application info block with these default names using the **CategoryInitialize** function, which has the following prototype:

```
void CategoryInitialize (AppInfoPtr appInfoP,
    UInt16 localizedAppInfoStrID)
```

The **CategoryInitialize** function needs a pointer to an application info block and the resource ID of an app info string containing the default category names. Librarian calls **CategoryIntialize** from its **LibAppInfoInit** function when it first creates a new application info block for the program, like this:

```
LibAppInfoType  *appInfo;


// Code to retrieve appInfo omitted.

CategoryInitialize((AppInfoPtr) appInfo, CategoryAppInfoStr);
```



For the complete story on initializing a new application info block for a database, see Chapter 12, "Storing and Retrieving Data."

Finding applications within a category

The function **DmSeekRecordInCategory** allows an application to search for records in a database, restricting the records returned to those that match a particular category. This function is particularly useful when an application displays records in a table or list, in conjunction with a pop-up list for selecting categories. Figure 13-2 shows the Librarian's category pop-up list in action. Selecting a category from the list changes which category is displayed in Librarian's List view.

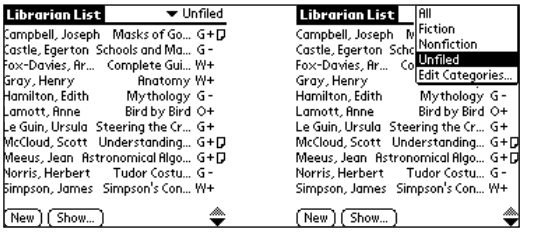


Figure 13-2: Left: Librarian's category pop-up trigger, in the upper-right corner of the screen, shows which category is currently displayed in the table. Right: The category pop-up list in Librarian allows subsets of Librarian's records to be displayed.

To restrict record retrieval to a specific category, call the **DmSeekRecordInCategory** function when filling a table or list, passing the desired category to **DmSeekRecordInCategory**, which has the following prototype:

```
Err DmSeekRecordInCategory (DmOpenRef dbP, UInt16* indexP,
    Int16 offset, Int16 direction, UInt16 category)
```

The first parameter to **DmSeekRecordInCategory** is an open database reference. Specify a category to restrict the search to using the `category` parameter, or pass the constant `dmAllCategories` to specify all records.

The `offset` parameter allows you to skip a number of records that match the requested category before returning a record. Starting at `index`, **DmSeekRecordInCategory** looks through the records in the direction specified by `direction` (either of the constants `dmSeekForward` or `dmSeekBackward`). When **DmSeekRecordInCategory** finds a record in the appropriate category, it decrements `offset` and continues searching for the next matching record until `offset` equals `0` and another match is found, at which point the function returns the index of the found record in the `index` parameter.

If **DmSeekRecordInCategory** successfully finds an appropriate record, it returns `0`; otherwise, it will return `dmErrIndexOutOfRange` or `dmErrSeekFailed` to indicate why it was unable to find a matching record.

Depending on whether or not the starting record at `indexP` is in the current category, you need to specify different values for `offset` to get the next matching record. If the record at `indexP` is in the specified category, an `offset` of `0` will return the current record. To find the next record in the category, you must call **DmSeekRecordInCategory** with an `offset` of `1` to skip over the current record.

As an example, the following code iterates over all the records in a database belonging to a specific category:

```
UInt16  i;
UInt16  recordNum;

recordNum = 0;

// Find the first record in the category.
```

```
if (DmSeekRecordInCategory(db, &recordNum, 0, dmSeekForward,
                           category)) {

    // Do something with the first record at index recordNum.

    for (i = 1; i < DmNumRecordsInCategory(db, category);
         i++) {
        // Use offset == 1 to skip over current record.
        if (! DmSeekRecordInCategory(db, &recordNum, 1,
                                     dmSeekForward, category))
            break;

        // Do something with the record at index recordNum.

        recordNum++;
    }
}
```

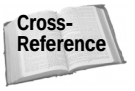
The example above first calls **DmSeekRecordInCategory** with an `offset` of `0` to ensure that if the record at index `0` in the database is in the correct category, the record is returned properly. Then, the code iterates over the rest of the records matching `category`, using **DmNumRecordsInCategory** to return the number of records in the category. Subsequent calls to **DmSeekRecordInCategory** in the body of the `for` loop use an `offset` of `1` to skip over the current record, since the current record in this loop will always match the desired category.

The DmNumRecordsInCategory function must examine all the records in the database; in a large database, this can take some time. Use DmNumRecordsIn Category only on small databases or in situations where speed is not critical.

Librarian uses the **DmSeekRecordInCategory** function extensively, as will any application that implements scrolling tables filled with records from a database. To make calling this function easier, Librarian wraps it in the **SeekRecord** function:

```
static Boolean SeekRecord (UInt16 *index, Int16 offset,
                           Int16 direction)
{
    DmSeekRecordInCategory(gLibDB, index, offset, direction,
                           gCurrentCategory);
    if (DmGetLastErr()) return (false);

    return (true);
}
```

The **SeekRecord** function cuts the number of parameters required for a **DmSeekRecordInCategory** call down to three, since it always looks in the global `gLibDB` variable for the database reference and it relies on the global `gCurrentCategory` to supply the proper category to search through. Also, **SeekRecord** returns a simple `Boolean` value to indicate whether or not a matching record was found.



See Chapter 11, "Programming Tables," for some examples of the SeekRecord function in action.

You can also use **DmQueryNextInCategory** to find functions in a specific category. The **DmQueryNextInCategory** function returns a handle to the next record in the database in the specified category:

```
recordH = DmQueryNextInCategory(db, &index, category);
```

Since **DmQueryNextInCategory** does not have an `offset` parameter like **DmSeekRecordInCategory**, you must increment the index parameter to skip over a known record in the correct category to get the next record in the category. Also, **DmQueryNextInCategory** does not allow you to specify a direction; it searches only forward through the database. The handle returned by this function is read-only; **DmQueryNextInCategory** does not set the busy bit in the record it finds.

Another useful function is **DmPositionInCategory**, which returns the position of a record within its own category, using a zero-based indexing system:

```
UInt16 position = DmPositionInCategory(db, index, category);
```

For example, if a record is the fourth record in the category passed to **DmPosition InCategory**, the function returns the value `3`.

Setting a record's category

To set the category for a particular record, use the following code:

```
DmRecordInfo(db, index, &attributes, NULL, NULL);
attributes &= ~dmRecAttrCategoryMask;
attributes |= category;
attributes |= dmRecAttrDirty;
DmSetRecordInfo(gLibDB, gCurrentRecord, &attr, NULL);
```

The example above sets the category of the record at `index` to the category specified by the `category` variable. Setting the dirty bit ensures that the HotSync Manager will deal properly with the record at synchronization time.

Selecting and modifying categories

When the user taps a category pop-up trigger, your application should display a list of available categories from which the user may select a new category. To match the behavior of category triggers in the built-in applications, the pop-up list should have an "Edit Categories..." item at the bottom of the list that launches the system category editing dialog box, pictured in Figure 13-3. From the editing dialog box the user may add, remove, and rename categories in the current application.

Figure 13-3: The system category editing dialog box

The **CategorySelect** function makes displaying a category list with appropriate behavior a fairly painless process. Call **CategorySelect** in response to a `ctlSelectEvent` triggered by the category pop-up trigger; it has the following prototype:

```
Boolean CategorySelect (DmOpenRef db, const FormType *frm,
    UInt16 ctlID, UInt16 lstID, Boolean title,
    UInt16 *categoryP, char *categoryName,
    UInt8 numUneditableCategories, UInt32 editingStrID)
```

You must supply an open database reference in the first parameter to **CategorySelect** so the function can retrieve category names from the database's application info block. The `frm`, `ctlID`, and `lstID` are pointers to the form, pop-up trigger, and list objects that **CategorySelect** should use when displaying the category list.

Pass `true` for the value of the `title` parameter if the category trigger is located on the title line of a form; if `title` is `true`, **CategorySelect** adds an "All" choice to the top of the pop-up list. Pass a `title` value of `false` if the trigger is intended to select the category for a single record, instead of for a list of records, where an "All" list item is not appropriate. A good example of this second kind of category trigger is in Librarian's Details dialog box, pictured in Figure 13-4. The category trigger in this dialog box changes the category of the current record instead of changing the current display category.

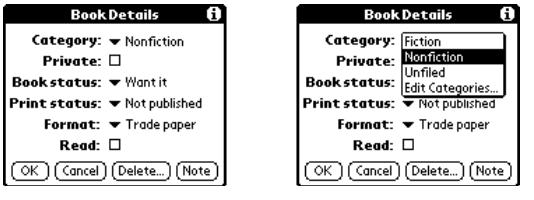


Figure 13-4: The category trigger in Librarian's Details dialog box, both closed (left) and open (right)

In the categoryP parameter, supply a pointer to a variable containing the category that should be selected in the pop-up list. When **CategorySelect** returns, it modifies the contents of the categoryP pointer to match the new category selected by the user, if any. Likewise, the categoryName parameter should point to a string containing the name of the selected category, and **CategorySelect** changes this string to the name of the new selected category.

You can retrieve the name of a category with the **CategoryGetName** function, which requires an open database reference, the category's index, and a pointer to a string to fill with the category name:

```
char  categoryName[dmCategoryLength];

CategoryGetName(db, category, categoryName);
```

The **CategorySelect** function's numUneditableCategories allows you to specify how many uneditable categories the application's category list contains. Usually, there is only one, the "Unfiled" category, but you can make applications with more uneditable categories if you want certain categories to always appear in the list.

Finally, the editingStrID parameter allows you to specify the resource ID of a string that appears at the bottom of the category list and in the title of the category editing dialog box. By default, this string is "Edit Categories..."; you can specify this default string by passing 0, or the constant categoryDefaultEditCategoryString, defined in the Palm OS header file Category.h. You may also use the constant categoryHideEditCategory to prevent the user from editing categories entirely; this constant removes the editing item from the bottom of the category pop-up list. If you want to customize the string that appears here, pass in the ID of your own string resource, and it will appear in the pop-up list and as the title of the editing dialog box.

Be sure not to make your custom string resource longer than dmCategoryLength (15 characters, plus a terminating null).

When the user selects the "Edit Categories..." list item (or its customized equivalent), **CategorySelect** calls the **CategoryEdit** function to launch the system category editing dialog box, shown earlier in Figure 13-3. You can call the **CategoryEdit** function directly from your application's code to launch the editing dialog box:

```
Boolean result = CategoryEdit(db, category, titleStrID,
    numUneditableCategories);
```

Both **CategorySelect** and **CategoryEdit** return a Boolean value to indicate that the user made some major change to the categories while it was running. The return value is true if any of the following occurred:

- ✦ The user renamed a category.
- ✦ The user deleted a category.

- The user renamed a category with the name of another category, merging those two categories.

If your application checks the return value from **CategorySelect** or **CategoryEdit**, it can react appropriately to the changes in the categories, reloading records into a table or list if the current form displays a list of records, or re-categorizing the current record as appropriate in a details dialog box.

The category selection and editing process has evolved as new versions of the Palm OS have been released. In Palm OS version 2.0, the `numUneditableCategories` parameter to CategoryEdit is unavailable, and Palm OS 1.0 does not allow customization of the category editing dialog box because CategoryEdit on version 1.0 does not have a `titleStrID` parameter. Palm OS 3.0 and later contain CategoryEditV20 and CategoryEditV10 functions to provide backward compatibility for applications written to support earlier versions of the operating system.

Likewise, CategorySelect is different between versions 1.0 and 2.0 of the Palm OS, with both the `numUneditableCategories` and `titleStrID` parameters in version 2.0. Even though `numUneditableCategories` is not available in the 2.0 version of CategoryEdit, the 2.0 CategorySelect does support this option. Starting with Palm OS 2.0, a backward-compatible CategorySelectV10 function is available to support applications that must maintain compatibility with Palm OS 1.0.

You can also programmatically set a category's name with the **CategorySetName** function, which may be called as follows:

```
CategorySetName(db, category, newNameString);
```

Pass a null-terminated string in the `newNameString` parameter to change the specified category's name. Your application may also use the **CategorySetName** function to delete a category; pass `NULL` for the new name instead of a string, and **CategorySetName** deletes the category instead of renaming it.

One more function, **DmMoveCategory**, allows your application to move all the records from one of its categories to another category:

```
Boolean  dirty = true;

Err error = DmMoveCategory(db, toCategory, fromCategory,
                           dirty);
```

The last parameter to **DmMoveCategory** controls whether records re-categorized by this function should be marked dirty or not; a `true` value marks them dirty, while a `false` value leaves the records' dirty bits alone.

Cycling through categories

The built-in applications that support categories, as well as the Librarian sample application, allow the user to cycle through the categories in the application by repeatedly pressing the hardware key assigned to launch the application. For example, if you press the Memo Pad hardware button while the Memo Pad application is already open and displaying its List view, Memo Pad changes its display to the next available category. When it gets to the end of the list of categories, Memo Pad displays all the categories, then goes on to the first category in the list again.

To implement this behavior in your application, you need to handle the `keyDownEvent` and look for hard key presses and, in response, call the **CategoryGetNext** function to retrieve the next available category. The following bit of code from Librarian's **ListFormHandleEvent** function handles a hard key press while the application is already running:

```
case keyDownEvent:
    if (TxtCharIsHardKey(event->data.keyDown.modifiers,
                         event->data.keyDown.chr)) {
        if (! (event->data.keyDown.modifiers &
               poweredOnKeyMask)) {
            ListFormNextCategory();
            handled = true;
        }
    }
```

The **ListFormNextCategory** function called above looks like this:

```
static void ListFormNextCategory (void)
{
    UInt16       category;
    TableType    *table;
    ControlType  *ctl;


    category = CategoryGetNext(gLibDB, gCurrentCategory);

    if (category != gCurrentCategory) {
        if (category == dmAllCategories)
            gShowAllCategories = true;
        else
            gShowAllCategories = false;

        ChangeCategory(category);

        // Set the label of the category trigger.
        ctl = GetObjectPtr(ListCategoryPopTrigger);
        CategoryGetName(gLibDB, gCurrentCategory,
                        gCategoryName);
        CategorySetTriggerLabel(ctl, gCategoryName);
```

```
            // Display the new category.
            ListFormLoadTable();
            table = GetObjectPtr(ListTable);
            TblEraseTable(table);
            TblDrawTable(table);

            // By changing the category the current record is
            // lost.
            gCurrentRecord = noRecord;
        }
    }
```

The first thing **ListFormNextCategory** does is to call **CategoryGetNext**, passing the Librarian open database reference and the current category, which Librarian keeps track of through the global `gCurrentCategory` variable.

The CategoryGetNext function's behavior is dependent on what version of the Palm OS is running. In Palm OS 1.0, CategoryGetNext cycles through the special "All" category, all of the named categories in alphabetical order, and the "Unfiled" category, then starts again with the "All" category. Starting with Palm OS 2.0, CategoryGetNext skips over categories that do not contain any records and the "Unfiled" category.

After retrieving the new category, **ListFormNextCategory** calls a small utility function, **ChangeCategory**, to update some Librarian global variables in response to a category change:

```
static void ChangeCategory (UInt16 category)
{
    gCurrentCategory = category;
    gTopVisibleRecord = 0;
}
```

Then **ListFormNextCategory** sets the label of the List view's category pop-up trigger, using the Palm OS function **CategorySetTriggerLabel**, which is far more convenient than having to deal with the **CtlSetLabel** function, since **CategorySetTriggerLabel** takes care of all the details normally required for setting a control label.

Finally, **ListFormNextCategory** reloads the List form's table to reflect the change of category.

Deleting all records in a category

The **DmDeleteCategory** function deletes all the records in a given category. Use the function as follows:

```
Err error = DmDeleteCategory(db, category);
```

If no error occurs while deleting the records, **DmDeleteCategory** returns `0`.

Despite its name, DmDeleteCategory does nothing to modify the name of the category given to it; DmDeleteCategory merely deletes the records within a category, leaving its name intact. If you want to delete a category's name, call the CategorySetName function and pass it a `NULL` for the string you would like to use to rename the category:

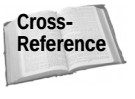
```
CategorySetName(db, category, NULL);
```

Implementing Private Records

The Palm OS provides a facility to enable the user to mark records in applications across the handheld as private, so that they may be shown only when the user enters the correct password in the device's Security application. To implement private records in your application, you must pay attention to the system-wide private records preference and program your application to respond appropriately, depending on what the current private record status is on the handheld.

Private records are marked with a secret bit only in the record's entry in the database header; the Palm OS does not take any steps to encrypt private records, and private information may easily be read by someone synchronizing the device and opening a database's backed up PDB file on the desktop in a text editor. If you need real security in a Palm OS application, you will have to program your own encryption.

You can retrieve the current private records status of the device by querying the system preferences with the **PrefGetPreference** function. Most applications should retrieve the system's current private record status in their **StartApplication** functions and assign the privacy status to a global variable for later use throughout the application. For example, Librarian stores the system private records status in the global variable `gPrivateRecordStatus`. Then the application should open the database in the appropriate mode; if private records are not hidden, use the `dmModeShowSecret` mode, which allows the various database functions to display hidden records. Without the `dmModeShowSecret` mode turned on, the **DmOpenDatabaseByTypeCreator** and **DmOpenDatabase** functions prevent access to records marked private.



For more information about retrieving the private record status, see the "Reading and Setting System Preferences" section in Chapter 12, "Storing and Retrieving Data." See the "Opening Databases" section in the same chapter to learn how to set the mode when opening a database.

Once you have the database opened in the appropriate mode, functions like **DmSeekRecordInCategory**, **DmQueryNextInCategory**, and **DmNumRecords InCategory** skip over private records when the database has not been opened in `dmModeShowSecret` mode.

To change a record's privacy status, use the **DmRecordInfo** and **DmSetRecordInfo** to retrieve and set the record's secret bit. The following code sets or clears the secret bit as appropriate:

```
DmRecordInfo(db, index, &attributes, NULL, NULL);
if (secret)
    attributes |= dmRecAttrSecret;
else
    attributes &= ~dmRecAttrSecret;
DmSetRecordInfo(db, index, &attributes, NULL);
```

Deleting all private records

If for some reason you should wish to delete all the private records in a database, call the **DmRemoveSecretRecords** function and pass it an open database reference. This function exists primarily for system use; if a user forgets the password assigned to show private records in the Security application and taps that application's Forgotten Password... button to reset the password, the Security application deletes all the private records on the device, using this function on every database.

Because the user may delete private records in this manner, it is a bad idea to use the secret bit for any purpose other than to mark a record as private, such as using the secret bit to flag a record as read-only. If the user resets the system password, all the "read-only" records in your application would suddenly disappear.

Be careful how you use DmRemoveSecretRecords if you use it at all. It can cause a lot of damage very quickly.

Resizing Records

The **DmResizeRecord** function allows you to change the size of a record's data chunk. This function comes in handy if your application's records are of variable length. To resize a record, call **DmResizeRecord** as follows:

```
MemHandle newHandle = DmResizeRecord(db, index, newSize);
```

The `newSize` parameter should contain the new size of the record, in bytes. If the heap that currently contains the record is not large enough to resize the record, **DmResizeRecord** reallocates the record in a different head on the same card. If this happens, the handle to the record's data chunk changes, so be sure to use the handle returned by the **DmResizeRecord** function after resizing a record, since the original handle may be invalid. If for some reason **DmResizeRecord** is unable to allocate enough space for the record, it returns `NULL`; in this case, call **DmGetLastErr** to retrieve an error code that indicates what went wrong.

Working with Resources

Resource databases have a slightly different structure from record databases', but many of the techniques and functions for handling resources are very similar to those used to work with records. The primary difference to keep in mind is that, unlike a record, each resource has a *type* assigned to it. Resource types are four-byte identifiers, similar in appearance to creator IDs or database types. Table 13-1 lists the common resource types available in the Palm OS.

Table 13-1
Palm OS Resource Types

Identifier	*Description*
`cnty`	Country-dependent information, such as date format and measurement system used in a particular country
`FONT`	Custom font
`MBAR`	Menu bar
`MENU`	Individual menu within a menu bar
`silk`	Information about the silk-screened area at the bottom of the screen, such as the locations of the buttons and the key codes they send when tapped
`tAIB`	Application icon, either the small icon or the large icon displayed in the system launcher application
`taif`	Application icon family; may contain multiple icons
`tAIN`	Application icon name (the name that appears in the system launcher application)
`tAIS`	App info string
`Talt`	Alert
`Tbmp`	Bitmap image, with support for up to 256 colors
`tbmf`	Bitmap family; may contain multiple bitmap images
`tBTN`	Command button
`tCBX`	Check box
`tFBM`	Form bitmap
`tFLD`	Text field
`tFRM`	Form
`tgbn`	Graphic button
`tGDT`	Gadget

Identifier	*Description*
tgpb	Graphic push button
tgrb	Graphic repeating button
tGSI	Graffiti shift indicator
tint	Integer constant
tLBL	Label
tLST	List
tPBN	Push button
tPUL	Pop-up list
tPUT	Pop-up trigger
tREP	Repeating button
tSCL	Scroll bar
tSLT	Selector trigger
tSTL	String list
tSTR	String
tTBL	Table
tver	Application version string (appears in the launcher's Info dialog box)

The Palm OS header file `UIResources.h` defines several useful constants for specifying resource types:

```
#define ainRsc              'tAIN'
#define alertRscType        'Talt'
#define appInfoStringsRsc   'tAIS'
#define bitmapRsc           'Tbmp'
#define constantRscType     'tint'
#define formRscType         'tFRM'
#define iconType            'tAIB'
#define MenuRscType         'MBAR'
#define silkscreenRscType   'silk'
#define strListRscType      'tSTL'
#define strRsc              'tSTR'
#define verRsc              'tver'
```

Also, unlike records, each individual resource has its own resource ID, which should be unique in the database for all resources of a particular type. For example, having a `tFRM` resource and a `tAIB` resource in a single database that both have a resource ID of `1000` is perfectly acceptable; two `tBTN` resources sharing a resource ID of `1000` in the same database is not permitted.

Finding Resources

Given the combination of resource type and resource ID, which uniquely identifies a resource within a database, searching for a particular resource is a more direct process than searching for a record. Finding the index of an individual resource in a database may be accomplished with the **DmFindResource** function, which has the following prototype:

```
UInt16 DmFindResource (DmOpenRef dbP, DmResType resType,
    DmResID resID, MemHandle resH)
```

There are two ways to use **DmFindResource**:

- ✦ Pass **DmFindResource** the type and resource ID of the desired resource.
- ✦ Pass **DmFindResource** a locked pointer to the resource.

If you have the type and resource ID of the resource you want to find, call **DmFindResource** like this:

```
index = DmFindResource(db, resourceType, resourceID, NULL);
```

If you already have a locked pointer to the resource, call **DmFindResource** like this:

```
index = DmFindResource(db, 0, 0, resourceH);
```

Passing a non-`NULL` handle to **DmFindResource** for its `resH` parameter causes the function to ignore the values passed in its `resType` and `resID` parameters.

If you need to iterate over all the resources in a database that match a particular type, use the **DmFindResourceType** function:

```
UInt16 DmFindResourceType (DmOpenRef dbP, DmResType resType,
                           UInt16 typeIndex)
```

The **DmFindResourceType** function starts at the index you give it via the `typeIndex` parameter and returns the index of the first resource it finds that matches the given `resType`. If **DmFindResourceType** encounters an error, it returns `-1`; otherwise, the function returns the index of the resource that it found.

As an example, the following code iterates through a database and counts the total number of bitmap resources (`Tbmp`):

```
UInt16  count = 0;
UInt16  i = 0;
UInt16  index;

while ( (index = DmFindResourceType(db, bitmapRsc, i) != -1) {

    // Do something with the resource at index here.
```

```
        count++;
        i = index + 1;
    }
```

If you need a broader search than **DmFindResource** or **DmFindResourceType** can provide, note that the **DmSearchResource** allows you to look through all the open resources databases on the handheld for a resource of a given type and resource ID—or if you do not have that information, you can provide **DmSearchResource** with a pointer to a locked resource instead. Not only does **DmSearchResource** return the index of the found record, it also returns a reference to the database containing the resource.

The **DmSearchResource** function operates in much the same way as **DmFind Resource**, but it is not restricted to a single database. To find a resource with a given type and resource ID, use the following code:

```
DmOpenRef  db;

index = DmSearchResource(resourceType, resourceID, NULL, &db)
```

If you already have a pointer to a locked database resource, pass it to **DmSearch Resource**, and the function will ignore its type and resource ID parameters:

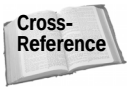
```
index = DmSearchResource(0, 0, resourceH, &db);
```

Along with finding specific resources, the Palm OS also provides the **DmNum Resources** function to count the total number of resources in a given resource database:

```
UInt16 numResources = DmNumResources(db);
```

Creating Resources

Most of the time, you create resources using Constructor or PilRC, and the CodeWarrior or GNU build tools to assemble these resources into a resource database (your application) for you. However, if you need to create a resource at run time, or if you want to define your own type of resource, the Palm OS offers **DmNewResource** for creation of resources.



Note that creating new resources is different from dynamically creating user interface elements at run time. For information about how to implement dynamic user interface, see Chapter 20, “Odds and Ends.”

The **DmNewResource** function has the following prototype:

```
MemHandle DmNewResource (DmOpenRef dbP, DmResType resType,
                         DmResID resID, UInt32 size)
```

You need to supply **DmNewResource** with an open database reference where you want to create the resource, a type, a resource ID, and the size of the new resource in bytes. Once you have allocated space for the new resource, you may write information to it using the **DmWrite** function much as you would write to a record; see the "Creating Records" section earlier in this chapter.

Once you are done writing data to the new resource, you need to call **DmRelease Resource** to signal to the system that the resource is no longer in use. Resources do not have a busy bit like records, but the system still keeps track of which resources are currently needed by an application, and it is important to program your application to release a resource once it is done modifying it or reading data from it. The **DmReleaseResource** function has the following prototype:

```
Err DmReleaseResource (MemHandle resourceH)
```

A different strategy for adding resources to a database involves allocating a new chunk of memory for the resource, and then attaching that chunk to the database using the **DmAttachResource** function, which has the following prototype:

```
Err DmAttachResource (DmOpenRef dbP, MemHandle newH,
    DmResType resType, DmResID resID)
```

Attaching a resource to a database with **DmAttachResource** is almost identical to attaching a record to a database using **DmAttachRecord**. The biggest difference is that resources do not have a particular sort order within a database but rather a unique type and resource ID combination, and that therefore the index where **DmAttachResource** attaches a new resource is completely irrelevant. This means that **DmAttachResource** cannot be used to replace an existing resource in a database, only to insert a new one.

If you do want to swap out a resource for a new copy, detach the original with DmDetachResource (see below) and then use DmAttachResource to connect the new resource to the database.

Deleting Resources

The **DmRemoveResource** function serves to remove a resource from a database and dispose of its memory chunk. **DmRemoveResource** has the following prototype:

```
Err DmRemoveResource (DmOpenRef dbP, UInt16 index)
```

Unlike with records, an application's conduit does not need to bother with deleting or archiving a desktop copy of a resource, so there are no corresponding "**DmDeleteResource**" or "**DmArchiveResource**" functions. Likewise, you do not need to worry about moving a deleted resource to the end of its database; once you call **DmRemoveResource**, the resource is gone.

You can also use the **DmDetachResource** function to orphan a resource from its database, usually in preparation for pasting the resource into a different database. The **DmDetachResource** function has the following prototype:

```
Err DmDetachResource (DmOpenRef dbP, UInt16 index,
                      MemHandle* oldHP)
```

The **DmDetachResource** function returns the handle of the resource's data chunk in its `oldHP` parameter. You should either use this handle to attach the resource to another database, or dispose of the chunk with the **MemHandleFree** function:

```
MemHandle  lonelyUnwantedResourceH;

DmDetachResource(db, index, &lonelyUnwantedResourceH);
MemHandleFree(lonelyUnwantedResourceH);
```

Reading Resources

While most resources represent user interface elements with their own sets of functions for interaction with a program, you must read from some resources directly in order to use them in your application. In particular, strings (`tSTR`), app info strings (`tAIS`), and bitmaps (`Tbmp`) require some special handling if they are to be used in an application.

Before reading data from a resource, you need to obtain a handle to the resource. The **DmGetResource** function searches through all open databases for a resource of a given type and resource ID, returning a handle to the resource if it finds a match. If **DmGetResource** cannot find an appropriate resource, it returns `NULL`; in this case, call **DmGetLastErr** to get an error code explaining the failure.

Once you have retrieved a handle to the resource, lock the handle with **MemHandleLock** to obtain a pointer to the data in the resource. Using this pointer, you can read data from the resource. Once you are done with the resource, unlock its handle with **MemHandleUnlock** and then release the resource with **DmReleaseResource**. Releasing the resource is an important step; if you fail to release the resource, it will be unavailable to later **DmGetResource** calls, because the system will think the resource is still in use.

The following example retrieves a handle to a string resource with a resource ID of `1000`, reads it into a string variable, and then releases the resource:

```
MemHandle  resourceH;
Char       *resource;
Char       *string[30];
```

```
resourceH = DmGetResource(strRsc, 1000);
resource = MemHandleLock(resourceH);
StrCopy(string, resource);
MemHandleUnlock(resourceH);
DmReleaseResource(resourceH);
```

If you already have the index of a resource, you can retrieve a handle to the resource with the **DmGetResourceIndex** function:

```
resourceH = DmGetResource(db, index);
```

You may also restrict the search for a matching type and resource ID to the most recently opened database by using the **DmGet1Resource** function instead of **DmGetResource**:

```
resourceH = DmGet1Resource(strRsc, 1000);
```

Remember to release the resource with **DmReleaseResource** after retrieving a resource handle with either **DmGetResourceIndex** or **DmGet1Resource**.

The Librarian sample application uses **DmGetResource** to retrieve strings for its Record view, pictured in Figure 13-5. Librarian displays status for a book using specific strings, such as "Got this book" or "Unread." These strings were created in Constructor or PilRC, and they are stored in app info string (type `tAIS`) resources in the application.

Figure 13-5: Librarian's Record view contains status strings (lower half of the screen, after the ISBN) that begin their lives as app info string resources.

For the sake of expediency, Librarian retrieves these strings from their resources only once, when it initializes its application info block, storing the strings in the application info block itself. It takes fewer steps to recall the strings from the application info block than to read them from the application's resources every time they are needed, since resources must be retrieved, locked, unlocked, and released in order for an application to access their contents.

Librarian's **LibGetAppInfoStr** function does all the dirty work of retrieving the app info strings and storing them in Librarian's application info block:

```
static void LibGetAppInfoStr (LibAppInfoType *appInfo,
    Int16 resourceID, Int16 stringCount, UInt32 arrayAddress)
```

```
{
    MemHandle  rscH, stringArrayH;
    char       *rsc;
    char       **stringArray;
    Int16      i;
    UInt32     offset;


    // Retrieve the strings from a string list in Librarian's
    // resources.
    rscH = DmGetResource(appInfoStringsRsc, resourceID);
    rsc = MemHandleLock(rscH);
    stringArrayH = SysFormPointerArrayToStrings(rsc,
                                                stringCount);
    stringArray = MemHandleLock(stringArrayH);

    // Set the initial offset at the beginning of the array in
    // the application info block.
    offset = arrayAddress - (UInt32) appInfo;

    // Copy each string into the application info block.
    for (i = 0; i < stringCount; i++) {
        if (stringArray[i][0] != '\0')
            DmStrCopy(appInfo, offset, stringArray[i]);
        // Increment the offset to the next array member.
        offset += sizeof(libLabel);
    }

    MemPtrFree(stringArray);
    MemPtrUnlock(rsc);
    DmReleaseResource(rscH);
}
```

The **SysFormPointerArrayToStrings** function is a handy utility that converts a packed block of strings, such as the format used in an app info string resource, into an array of pointers to strings, suitable for use in a list, or in this case, for inclusion in an array in Librarian's application info block.

Retrieving and Modifying Resource Information

You can retrieve and set certain pieces of information about a resource with the **DmResourceInfo** and **DmSetResourceInfo** functions, which behave similarly to the **DmRecordInfo** and **DmSetRecordInfo** functions for records. The prototypes for these two resource information functions look like this:

DmResourceInfo

```
Err DmResourceInfo (DmOpenRef dbP, UInt16 index,
    DmResType* resTypeP, DmResID* resIDP,
    LocalID* chunkLocalIDP)
```

DmSetResourceInfo

```
Err DmSetResourceInfo (DmOpenRef dbP, UInt16 index,
    DmResType* resTypeP, DmResID* resIDP)
```

If you pass `NULL` for the `resTypeP`, `resIDP`, or `chunkLocalIDP` pointers, these functions will avoid retrieving or setting the piece of information represented by the `NULL` pointer. Note that you cannot change the LocalID of the resource's data chunk.

Resizing Resources

If you should need to resize a resource, use the **DmResizeResource** function to reallocate space for the resource:

```
MemHandle newResourceH = DmResizeResource(resourceH, newSize);
```

Just like **DmResizeRecord**, **DmResizeResource** may need to move the resource's data to a different memory heap, invalidating the original resource handle. After resizing a resource, be sure to use the handle returned by **DmResizeRecord** for continued access to that resource's data.

Implementing the Global Find Facility

Any application that stores text data can benefit from the Palm OS global find feature. Activated by means of the silk-screened Find button, the Find dialog box, pictured in Figure 13-6, allows the user to search through any applications that support global find for a particular text string.

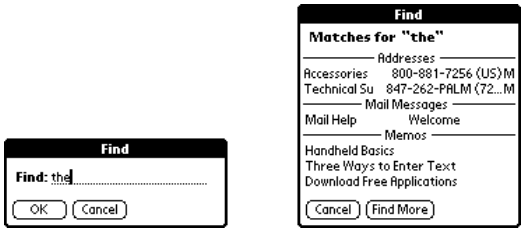


Figure 13-6: The global Find dialog box (left) and the Find results dialog box (right)

To implement the global find feature, an application should handle three launch codes: `sysAppLaunchCmdSaveData`, `sysAppLaunchCmdFind`, and `sysAppLaunchCmdGoto`.

The following list outlines the sequence of events that occur when the user enters text in the Find dialog box and taps the dialog box's OK button:

1. The system sends a `sysAppLaunchCmdSaveData` launch code to each application on the device. This launch code allows an application that supports find to save its own unsaved data if the application happens to be open when the find operation begins.
2. The system sends a `sysAppLaunchCmdFind` launch code to each application. Any program that supports the global find responds to this launch code by querying its database for the search text supplied by the user, returning a list of records that contain the appropriate text. The find dialog box displays a list of matching records in all applications that support the find feature.
3. If the user selects a matching record from the list, the system sends a `sysAppLaunchCmdGoto` launch code to the application that owns the selected record. The owning application can then respond to the launch code by displaying the appropriate record from its database.

Keep in mind that in a multi-segment application, any function referenced from PilotMain, including any functions used to implement the global find facility, must be located in the same code segment as PilotMain. For more information about multi-segment applications, turn to Chapter 20, "Odds and Ends."

Handling sysAppLaunchCmdSaveData

If a find-aware application is currently open when the user performs a global find, it needs to save any unsaved data, because the find operation can display a different record from the record that is currently displayed, or even open a different application. This sudden switch to a new record can cause data loss if the application has a record open for editing.

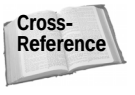
Fortunately, the `sysAppLaunchCmdSaveData` launch code that the system sends before a `sysAppLaunchCmdFind` code provides an easy way to make sure that everything is saved in the application. The simplest thing to do in response to `sysAppLaunchCmdSaveData` is to call the **FrmSaveAllForms** function, which sends a `frmSaveEvent` to all the open forms in an application. Then, in the event handler for any form that might have unsaved data when a global find occurs, respond to the `frmSaveEvent` by saving unsaved data.

As an example, here is the portion of the Librarian application's **PilotMain** routine that takes care of an incoming `sysAppLaunchCmdSaveData` launch code:

```
switch (cmd) {
    case sysAppLaunchCmdSaveData:
        FrmSaveAllForms();
        break;
    // Other launch codes omitted
}
```

The only form in Librarian that might contain unsaved data when a global find occurs is the Edit form, which might have one unsaved text field within its table. Librarian's **EditFormHandleEvent** function takes care of the `frmSaveEvent` by releasing the focus from whichever field in the Edit form is currently being edited:

```
switch (event->eType) {
    case frmSaveEvent:
        table = GetObjectPtr(EditTable);
        TblReleaseFocus(table);
        break;
    // Other events omitted
}
```



For more details about how Librarian handles text editing in its Edit form, see Chapter 11, "Programming Tables."

Handling sysAppLaunchCmdFind

When handling a `sysAppLaunchCmdFind` launch code, an application should perform the following actions:

1. Open the application's database.
2. Draw an appropriate header string, using the **FindDrawHeader** function, in the Find results dialog box to differentiate matching records in the current application from those found in other applications.
3. Search through the application's records for matching text using the **FindStrInStr** function.
4. For each successful match, call **FindSaveMatch** to let the system know that a match has been found so the system can update its own internal data about the progress of the global find operation. Also, retrieve the screen location of the next line in the Find results dialog box with **FindGetLineBounds** and draw an appropriate string in the dialog box to identify the record that was found.
5. Close the application's database.

The Librarian sample application's **PilotMain** function passes handling of the `sysAppLaunchCmdFind` launch code to another function called **Search**. The **PilotMain** function also casts the parameter block received with the launch code to a `FindParamsType` for use by the various find functions in the **Search** routine:

```
UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
    switch (cmd) {
        case sysAppLaunchCmdFind:
            Search( (FindParamsType *) cmdPBP);
            break;

        // Other launch codes omitted
```

```
        }

        return 0;
    }
```

Listing 13-3 shows Librarian's **Search** function. The listing includes the numbered steps listed above for reference.

Listing 13-3: **Librarian's Search function**

```
static void Search (FindParamsPtr params)
{
    LibAppInfoType  *appInfo;
    LibDBRecordType  record;
    MemHandle  recordH;
    Boolean  done, match;
    DmOpenRef  db;
    DmSearchStateType  searchState;
    Err      error;
    Char     *header;
    MemHandle  headerStringH;
    RectangleType  r;
    LocalID  dbID;
    UInt16   cardNo = 0;
    UInt16   recordNum;
    UInt16   i;
    UInt16   pos;
    Char     *noAuthor = NULL;
    Char     *noTitle = NULL;


    // Find the Librarian database.
    error = DmGetNextDatabaseByTypeCreator(true, &searchState,
        libDBType, libCreatorID, true, &cardNo, &dbID);
    if (error) {
        params->more = false;
        return;
    }
```

Step 1

```
    // Open the Librarian database.
    db = DmOpenDatabase(cardNo, dbID, params->dbAccesMode);
    if (! db) {
        params->more = false;
        return;
    }
```

Continued

Listing 13-3 *(continued)*

Step 2

```
// Display the heading line.
headerStringH = DmGetResource(strRsc, FindHeaderString);
header = MemHandleLock(headerStringH);
done = FindDrawHeader(params, header);
MemHandleUnlock(headerStringH);
DmReleaseResource(headerStringH);
if (done)
    goto Exit;

// Search the description and note fields for the "find"
// string.
recordNum = params->recordNum;
while (true) {
    // Applications may take a long time to finish a find,
    // so it is a good idea to allow the user to interrupt
    // the find at any time. This allows the user to
    // immediately go to a displayed record by tapping on
    // it, even before the global find finishes filling
    // the screen, or to cancel the find entirely by
    // tapping the Stop button. To accomplish this, check
    // to see if an event is pending, and stop the find if
    // there is an event. This call slows down the
    // search, so it should only be performed every
    // sixteen records instead of at each and every
    // record. If that 16th record is marked secret, and
    // the system is currently set to hide private
    // records, the check does not occur, because
    // DmQueryNextInCategory respects the database access
    // mode used earlier to open the database.
    if ((recordNum & 0x000f) == 0 &&  // every 16th record
        EvtSysEventAvail(true)) {
        // Stop the search process.
        params->more = true;
        break;
    }

    recordH = DmQueryNextInCategory(db, &recordNum,
                                    dmAllCategories);

    // Stop searching if there are no more records.
    if (! recordH) {
        params->more = false;
        break;
    }
```

Step 3

```
        // Search all the fields of the Librarian record.
        LibGetRecord(db, recordNum, &record, &recordH);
        match = false;
        for (i = 0; i < libFieldsCount; i++) {
            if (record.fields[i]) {
                match = FindStrInStr(record.fields[i],
                                     params->strToFind, &pos);
                if (match)
                    break;
            }
        }
```

Step 4

```
        if (match) {
            done = FindSaveMatch(params, recordNum, pos, i, 0,
                                 cardNo, dbID);
            if (done) {
                MemHandleUnlock(recordH);
                break;
            }

            // Get the bounds of the region where we will draw
            // the results.
            FindGetLineBounds(params, &r);

            appInfo = MemHandleLock(LibGetAppInfo(db));

            // Display the title of the description.
            FntSetFont(stdFont);
            DrawRecordName(&record, &r, appInfo->showInList,
                           &noAuthor, &noTitle);

            MemPtrUnlock(appInfo);

            params->lineNumber++;
        }

        MemHandleUnlock(recordH);
        recordNum++;
    }

    // Unlock handles to unnamed items.
    if (noAuthor != NULL)
        MemPtrUnlock(noAuthor);
    if (noTitle != NULL)
        MemPtrUnlock(noTitle);
```

Continued

Listing 13-3 *(continued)*

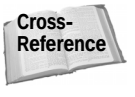
Step 5

```
Exit:
    DmCloseDatabase(db);
}
```

The search performed by the **FindStrInStr** function is case-insensitive. Another thing to keep in mind is that **FindStrInStr** matches only the beginnings of words. For example, a search for "bar" will match the word "bark" but not the word "embarrassed."

Handling sysAppLaunchCmdGoto

An application supporting global find that receives a `sysAppLaunchCmdGoto` launch code should display the record requested by the system. This can be a bit tricky, since the application that is handling the `sysAppLaunchCmdGoto` launch code may or may not already be running.



The `sysAppLaunchCmdGoto` launch code is also integral to the process of beaming records from one handheld to another. See the "Displaying Beamed Records" section of Chapter 14, "Beaming Data by Infrared," for more details.

An application can tell if it was launched by the `sysAppLaunchCmdGoto` launch code, or if it is already running, by checking its **PilotMain** function's `launchFlags` parameter for the `sysAppLaunchFlagNewGlobals` flag, which indicates that the launch code comes with its own brand new set of global variables. A new set of global variables means that the application was not already running.

If the application is already running, you must take care to close any open dialog boxes in the application before switching to whatever view displays the requested record. If the application is not already running, you need to take care of some of the same responsibilities that the `sysAppLaunchCmdNormalLaunch` launch code takes care of, namely running the application's **StartApplication** function before displaying the found record, and then calling the application's **EventLoop** and **StopApplication** functions. A `sysAppLaunchCmdGoto` launch is essentially a second entry point into the application.

As an example of how to handle `sysAppLaunchCmdGoto`, here are the appropriate parts of Librarian's **PilotMain** function:

```
UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
    Err      error;
    Boolean  launched;
```

```
    switch (cmd) {
        case sysAppLaunchCmdGoTo:
            launched = launchFlags &
                sysAppLaunchFlagNewGlobals;

            if (launched)  {
                error = StartApplication();
                if (error)
                    return (error);
            }

            GoToItem( (GoToParamsType *) cmdPBP, launched);

            if (launched) {
                EventLoop();
                StopApplication();
            }
            break;

        // Other launch codes omitted
    }

    return 0;
}
```

Librarian's **PilotMain** function defers most of the processing of the `sysAppLaunchCmdGoto` launch code to another function, **GoToItem**, by first casting the launch code's parameter block to a `GoToParamsType` and then passing the parameter block to **GoToItem**. The **GoToItem** function looks like this:

```
static void GoToItem (GoToParamsPtr goToParams,
                      Boolean launchingApp)
{
    UInt16   formID;
    UInt16   recordNum;
    UInt16   attr;
    UInt32   uniqueID;
    EventType  event;
    UInt32   romVersion;


    recordNum = goToParams->recordNum;
    DmRecordInfo(gLibDB, recordNum, &attr, &uniqueID, NULL);

    // Change the current category if necessary.
    if (gCurrentCategory != dmAllCategories) {
        gCurrentCategory = attr & dmRecAttrCategoryMask;
    }

    // If the application is already running, close all the
    // open forms. If the record currently displayed is
    // blank, it will be deleted, which knocks all the record
    // indices off by one. Use the found record's unique ID
```

```
        // to find the record index again once all the forms are
        // closed.
        if (! launchingApp) {
            FrmCloseAllForms();
            DmFindRecordByID(gLibDB, uniqueID, &recordNum);
        }

        // Set global variable to keep track of the current record.
        gCurrentRecord = recordNum;

        // Set gPriorFormID so the Note view returns to the List
        // view.
        gPriorFormID = ListForm;

        if (goToParams->matchFieldNum == libFieldNote) {
            // If running on Palm OS 3.5 or above, use the
            // NewNoteView form; otherwise, stick with the
            // original NoteView.
            FtrGet(sysFtrCreator, sysFtrNumROMVersion,
                   &romVersion);
            if (romVersion >=
                sysMakeROMVersion(3,5,0,sysROMStageRelease,0))
                formID = NewNoteView;
            else
                formID = NoteView;
        }
        else {
            formID = RecordForm;
        }

        MemSet(&event, sizeof(EventType), 0);

        // Send an event to load the form.
        event.eType = frmLoadEvent;
        event.data.frmLoad.formID = formID;
        EvtAddEventToQueue(&event);

        // Send an event to go to a form and select the matching
        // text.
        event.eType = frmGotoEvent;
        event.data.frmGoto.formID = formID;
        event.data.frmGoto.recordNum = recordNum;
        event.data.frmGoto.matchPos = goToParams->matchPos;
        event.data.frmGoto.matchLen = goToParams->searchStrLen;
        event.data.frmGoto.matchFieldNum =
            goToParams->matchFieldNum;
        EvtAddEventToQueue(&event);
    }
```

The **GoToItem** function first retrieves the record information for the found record by calling **DmRecordInfo**. Both the record's attributes and unique ID are interesting to **GoToItem**. The attributes allow **GoToItem** to change the current category if the

category the record belongs to is not the category currently displayed in Librarian, and the unique ID allows **GoToItem** to find the record again after calling **FrmCloseAllForms**. If Librarian is already running, and it is currently displaying an empty record in its Edit view, that record is deleted as soon as **FrmCloseAllForms** closes the Edit form; deleting the record bumps all the record indices in the database by one, invalidating the index provided by the **GoToItem** function's `goToParams` parameter. Passing the unique ID of the record to **DmFindRecordByID** allows **GoToItem** to recover the correct record index.

If the found text is part of the note field, **GoToItem** launches Librarian's Note view; otherwise, **GoToItem** launches the Record view. To launch the appropriate form, **GoToItem** must queue a `frmLoadEvent` for the form, and then a `frmGotoEvent` to actually go to that form. The `frmGotoEvent` also contains data about the position and length of the found text so the form can highlight the text in its display.

The last part of implementing `sysAppLaunchCmdGoto` involves handling the `frmGotoEvent` in the forms that might pop up in response to a `sysAppLaunchCmdGoto` launch code. In Librarian, both the Record and Note views handle `frmGotoEvent`. Here is the appropriate section of the **NoteViewHandleEvent** function, to demonstrate how Librarian highlights the found text in the Note view:

```
case frmGotoEvent:
    form = FrmGetActiveForm();
    gCurrentRecord = event->data.frmGoto.recordNum;
    NoteViewInit(form);
    field = GetObjectPtr(NoteField);
    FldSetScrollPosition(field, event->data.frmGoto.matchPos);
    FldSetSelection(field, event->data.frmGoto.matchPos,
                    event->data.frmGoto.matchPos +
                    event->data.frmGoto.matchLen);
    NoteViewDrawTitle(form);
    NoteViewUpdateScrollBar();
    FrmSetFocus(form, FrmGetObjectIndex(form, NoteField));
    handled = true;
    break;
```

Summary

This chapter showed you how to read and write records and resources, as well as how to implement the Palm OS global find facility. After reading this chapter, you should understand the following:

- Most Palm OS record databases keep their records in sorted order to permit rapid population of lists and tables from record data.
- You should implement a callback comparison function for your record database for the system to call when you use the **DmFindSortPosition**, **DmInsertionSort**, or **DmQuickSort** functions.

- You can find a record, or the position it should occupy, using the **DmFind SortPosition**, **DmFindRecordByID**, and **DmSearchRecord** functions.
- An application may create records with the **DmNewRecord** function, or by allocating a new memory chunk and attaching it to the database with **DmAttachRecord**.
- Deleted and archived records remain at the end of a database's record list, so that an application's conduit can update the desktop version of the application's database.
- The **DmRecordInfo** and **DmSetRecordInfo** functions allow you to retrieve and set attributes and other properties of database records.
- After initializing default category names with **CategoryIntialize**, you can implement the standard Palm OS category selection list by calling **CategorySelect** and **CategoryEdit**.
- All implementation of private records is up to your application; the system takes care only of keeping track of the global security setting and a secret bit on each record.
- Resources differ from records, in that each resource has a type and a resource ID, making it unnecessary to keep resources in a sorted order within a database.
- To implement the system global find facility, you must handle three application launch codes: `sysAppLaunchCmdSaveData`, `sysAppLaunchCmdFind`, and `sysAppLaunchCmdGoto`.

✦ ✦ ✦

CHAPTER 14

Beaming Data by Infrared

In This Chapter

Understanding the Exchange Manager

Registering a data type

Beaming records and categories

Receiving records and categories

Beaming applications

Customizing the beam acceptance dialog box

Understanding the IR Library

First introduced in Palm OS 3.0 on the Palm III, infrared (IR) beaming allows data exchange between two Palm OS handhelds, or between a Palm OS handheld and another IR-capable device, without one's having to attach cables to either device. All of the infrared transfer capabilities of the Palm OS are based on industry-standard protocols set by the Infrared Data Association (IrDA), which allows a Palm OS handheld to "talk" to many other devices, not just those running the Palm OS.

The Palm OS offers two general levels of IR beaming support:

- The *Exchange Manager*, which is a simple high-level interface for exchanging data with a variety of other devices and protocols
- The *IR Library*, which gives an application a direct interface with the underlying hardware and software protocols that run IR communications on a Palm OS handheld

The Exchange Manager can easily handle most of the data exchange required to pass data back and forth between two devices, and this chapter will concentrate on using the Exchange Manager. At the end of this chapter, you can find an introduction to using the IR Library for those situations where some hard-core low-level communications hacking is the only way get complex data transfer to work. However, the Palm OS Exchange Manager is much easier to use, and it should suffice for almost any infrared transfer application.

Using the Exchange Manager

The Exchange Manager provides a simple interface between a Palm OS application and the Put operation of the IrDA Data standard's Object Exchange (OBEX) layer. OBEX is designed

to allow quick and easy transfer of an arbitrary "thing" from one device to another. This "thing" can be any arbitrary data object, and the Put operation in OBEX can transmit such a data object, either as a quick, all-at-once transfer, or as an extended transfer that takes place over a longer period of time.

Complete specifications for the IrDA OBEX layer are available on the Web at `http://www.irda.org`.

Central to using the Exchange Manager is the `ExgSocketType` structure. Defined in the Palm OS header `ExgMgr.h`, `ExgSocketType` holds information about the connection and the type of data being transferred. When sending data, an application needs to fill in the appropriate fields in this structure. Likewise, when receiving data, an application can retrieve information about an incoming data stream and its contents from the socket structure.

The term "socket," as used by the Exchange Manager, has nothing to do with socket communications programming.

The `ExgSocketType` structure looks like this:

```
typedef struct ExgSocketType {
    UInt16  libraryRef;
    UInt32  socketRef;
    UInt32  target;
    UInt32  count;
    UInt32  length;
    UInt32  time;
    UInt32  appData;
    UInt32  goToCreator;
    ExgGoToType  goToParams;
    UInt16  localMode:1;
    UInt16  packetMode:1;
    UInt16  noGoTo:1;
    UInt16  noStatus:1;
    UInt16  reserved:12;
    Char    *description;
    Char    *type;
    Char    *name;
}
```

Table 14-1 describes the fields in the `ExgSocketType` structure.

Table 14-1
The ExgSocketType Structure

Field	*Description*
`libraryRef`	Internal field used by the Exchange Manager to identify the exchange library that is in use. You should leave this field alone.
`socketRef`	Identifier for the connection itself, used internally by the Exchange Manager. You should also leave this field alone.
`target`	Creator ID of the application that the data is being sent to.
`count`	Number of data objects in this connection. This value is usually `1`, even if the application is sending multiple records
`length`	Total length in bytes of all the objects being transferred. This field is optional.
`time`	Time when the object was last modified. This field is optional.
`appData`	Application-specific data.
`goToCreator`	Creator ID of the application to launch after the transfer is complete.
`goToParams`	Structure containing information about where to go if `goToCreator` is specified.
`localMode`	If set to `1`, transfer takes place only with the local machine; if set to `0`, transfer is enabled with a remote device.
`packetMode`	If set to `1`, use connectionless packet mode (OBEX Ultra mode) for the transfer. This field defaults to `1`, and you should usually leave this value alone.
`noGoTo`	If set to `1`, the Exchange Manager does not launch the application specified in `goToCreator` after completion of the transfer. This field only works when `localMode` is `1`.
`noStatus`	If set to `1`, the Exchange Manager does not display any transfer status dialog boxes.
`reserved`	Reserved for future use.
`description`	Pointer to a string containing a text description of the data object that is being transferred. This description is displayed to the user on both the sending and receiving ends of the transfer.
`type`	Pointer to a string describing the MIME type of the data object. This field is optional.
`name`	Pointer to a string holding the name of the data object, usually a file name. This field is optional.

The ExgGoToType structure in the ExgSocketType structure's goToParams field is also defined in ExgMgr.h, and it contains information about the record to display once the transfer has been completed. Here is what the ExgGoToType structure looks like:

```
typedef struct {
    UInt16   dbCardNo;
    LocalID  dbID;
    UInt16   recordNum;
    UInt32   uniqueID;
    UInt32   matchCustom;
} ExgGoToType;
```

Table 14-2 briefly describes the fields in the ExgGoToType structure.

Table 14-2
The ExgGoToType Structure

Field	*Description*
dbCardNo	Card number containing the database that holds the record.
dbID	LocalID of the database that holds the record.
recordNum	Index of the record to display.
uniqueID	Position within the record of the data that should be displayed. Note that this is not the same as a record's unique ID.
matchCustom	Custom information for use by the application.

The rest of this chapter gives more detail about how to use the ExgSocketType and ExgGoToType structures.

Every application that is to receive beamed data must register the type (or types) of data it can handle with the Exchange Manager, using the **ExgRegisterData** function. When the Exchange Manager detects incoming data that can be handled by one of the programs on the handheld, the manager sends three launch codes to the registered application so it can receive the data:

- sysAppLaunchCmdAskUser, to present a customized dialog box to the user for accepting or rejecting incoming data
- sysAppLaunchCmdReceiveData, to actually receive the incoming data
- sysAppLaunchCmdGoTo, to display the newly received record

Registering a Data Type

If you wish to write an application that communicates only with another copy of itself on another Palm OS handheld, you can simply specify the application's creator ID in the `target` field of the `ExgSocketType` structure, and the Exchange Manager will send the appropriate launch codes to that application on the receiving handheld. However, this scenario is somewhat limited; the Exchange Manager was built with enough flexibility to allow transfer between the Palm OS and completely different applications, operating systems, and devices. In order to allow a Palm OS application to receive data from so many different sources, the application must register with the Exchange Manager the types of data it can handle.

At least one of three pieces of identifying information about a data object is required to allow an application to receive it:

- The creator ID of the specific Palm OS application that will receive the data
- A file name identifying the data, usually with some kind of file extension (for example, `.txt` for text files)
- A Multipurpose Internet Mail Extensions (MIME) data type (for example, `text/html` for HTML documents)

An application should call **ExgRegisterData** immediately after it has been installed on the handheld to let the Exchange Manager know what kinds of data the program would like to receive. The prototype for **ExgRegisterData** looks like this:

```
Err ExgRegisterData(const UInt32 creatorID, const UInt16 id,
                    const Char * const dataTypesP)
```

The `creatorID` parameter specifies the application to register to handle a data type. Use the `id` parameter to define exactly what kind of data type to register. The two possible values for `id` are the constants `exgRegExtensionID` for specifying a file extension, or `exgRegTypeID` for specifying a MIME data type.

Pass a pointer to a null-terminated string containing the file extensions or MIME types to register in the `dataTypesP` parameter. You can specify multiple file extensions or MIME types in the same string by delimiting types with tab characters.

For example, the following **ExgRegisterData** call registers an application to handle files with a `.txt` file extension:

```
Err error = ExgRegisterData(myCreatorID, exgRegExtensionID,
                            "txt");
```

This example registers an application to handle HTML documents, which have a MIME type of `text/html`:

```
Err error = ExgRegisterData(myCreatorID, exgRegTypeID,
                            "text/html");
```

The Internet Assigned Numbers Authority (IANA) maintains the central registry of MIME types. A complete list of MIME types is available on the Web at `http://www.isi.edu/in-notes/iana/assignments/media-types/media-types`.

The best time to register an application for handling beamed data is right after the application has been installed to the handheld. If an application were to call **Exg RegisterData** only in its **StartApplication** routine, the program would be able to respond to incoming data transmissions only after it has been run once. Instead, you can call **ExgRegisterData** in response to the `sysAppLaunchCmdSyncNotify` launch code. The system sends this launch code to an application whose databases have been changed during the most recent HotSync operation. Installing the application itself counts as a change in the application's database, so an application can catch the `sysAppLaunchCmdSyncNotify` launch code and use it to register itself with the Exchange Manager.

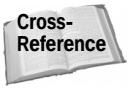
As an example, the following relevant parts of the Librarian application's **PilotMain** function call another Librarian function, **LibRegisterData**, to register Librarian to handle incoming data with a `.lib` file extension:

```
UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
    switch (cmd) {
        case sysAppLaunchCmdSyncNotify:
            LibRegisterData();
            break;
    }

    return 0;
}
```

The **LibRegisterData** function and the `libFileExtension` constant it uses look like this:

```
#define libFileExtension  "lib"


void LibRegisterData(void)
{
    UInt32  romVersion;


    // Beaming is only available on Palm OS 3.0 and later.
    FtrGet(sysFtrCreator, sysFtrNumROMVersion, &romVersion);
    if (romVersion >=
        sysMakeROMVersion(3,0,0,sysROMStageRelease,0))
        ExgRegisterData(libCreatorID, exgRegExtensionID,
                        libFileExtension);
}
```

Because beaming and the Exchange Manager are available only on Palm OS 3.0 and later, Librarian registers itself only if it is installed on an appropriate version of the operating system. Also, Librarian uses two different sets of menus, one with beaming commands, for 3.0 and later, and one without, for running Librarian on Palm OS 2.0, so that beaming functions appear in Librarian's interface only if the program is running on a device that supports beaming.



Take a look at the "Programming Menus" section of Chapter 9, "Programming User Interface Elements," for details about removing items from menus.

Sending Data

Sending data through the Exchange Manager is a four-step process:

1. Initialize an `ExgSocketType` structure.
2. Call **ExgPut** to begin the transfer.
3. Call **ExgSend** from within a loop to send the data.
4. Call **ExgDisconnect** to end the transfer.

Initializing an ExgSocketType structure

Every beam operation requires an `ExgSocketType` structure to define how the transfer should take place. Before using the socket structure, it is important to set it to zero with the **MemSet** function; random bits left over from declaring the `ExgSocketType` structure can cause unpredictable (and undesirable) results when it comes time to actually beam data. The following lines of code declare an exchange socket structure and wipe it clean:

```
ExgSocketType  exgSocket;

MemSet(&exgSocket, sizeof(exgSocket), 0);
```

Once the socket structure is zeroed, you should set appropriate fields in the structure to control how the beaming operation should proceed. The first field you should initialize is the `description` field, which is a pointer to a string that describes the data that is about to be sent. This description string appears in the beaming dialog boxes on both the sending and receiving devices. Figure 14-1 shows beaming dialog boxes as used by the Librarian sample application when beaming the "Nonfiction" category.

When beaming an individual record, it is customary to use part of the record's data for the description field:

```
exgSocket.description = myRecord->name
```

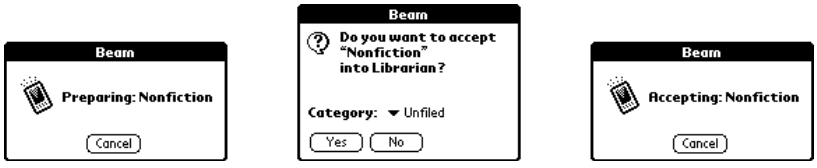


Figure 14-1: Beaming dialog boxes display a description string that describes the data to be transferred. Pictured here are the preparation dialog box (left), a dialog box to prompt the user to accept the data (middle), and the accepting dialog box (right), all of which display the description "Nonfiction."

Librarian's **LibBeamRecord** function, shown in Listing 14-1, goes to considerable lengths to assemble an appropriate string for the description, because not all of the fields in a Librarian record may contain data. Librarian uses a book record's title by default; records that lack a title have a description beginning with "a book by" and ending with the author's name; records with neither title nor author simply use the string "a book." Assembling a good description string may be as simple or complex a process as you want. Ideally, the description should give the user enough information to decide whether to accept or reject the incoming record.

Listing 14-1: **Librarian's LibBeamRecord function**

```
void LibBeamRecord (DmOpenRef db, Int16 recordNum)
{
    LibDBRecordType  record;
    MemHandle  recordH;
    LibPackedDBRecord  *packed;
    MemHandle  packedH;
    MemHandle  descH;
    UInt16     descSize = 0;
    Coord      descWidth, ignoreHeight;
    Boolean    descFit;
    UInt16     newDescSize;
    MemHandle  prefixH;
    Char       *prefix;
    MemHandle  nameH;
    Err        error;
    ExgSocketType  exgSocket;


    // Initialize the exchange socket structure to zero.
    MemSet(&exgSocket, sizeof(exgSocket), 0);

    // Assemble a description of the record to send. This
    // description is displayed by the system send and receive
```

```
// dialog boxes on both the sending and receiving devices.
error = LibGetRecord(db, recordNum, &record, &recordH);
ErrNonFatalDisplayIf(error, "Can't get record");

if (RecordContainsData(&record)) {
    // Use the title of the book, if it exists. If the
    // book record is untitled, use the author's name.
    // Failing that, fall back to a generic string stored
    // in Librarian's resources.
    descH = NULL;
    exgSocket.description = NULL;
    if (record.fields[libFieldTitle]) {
        // Use title of book for the description.
        descSize = StrLen(record.fields[libFieldTitle]) +
            sizeOf7BitChar('\0');
        descH = MemHandleNew(descSize);
        if (descH) {
            exgSocket.description = MemHandleLock(descH);
            StrCopy(exgSocket.description,
                    record.fields[libFieldTitle]);
        }

    } else if (record.fields[libFieldFirstName] ||
               record.fields[libFieldLastName]) {
        // Use "a book by <author>" for the description.
        prefixH = DmGetResource(strRsc,
                                UntitledBeamString);
        prefix = (Char *) MemHandleLock(prefixH);
        descSize = StrLen(prefix);

        if (record.fields[libFieldFirstName] &&
            record.fields[libFieldLastName])
            descSize += sizeOf7BitChar(' ') +
                        sizeOf7BitChar('\0');
        else
            descSize += sizeOf7BitChar('\0');

        if (record.fields[libFieldFirstName])
            descSize +=
                StrLen(record.fields[libFieldFirstName]);

        if (record.fields[libFieldLastName])
            descSize +=
                StrLen(record.fields[libFieldLastName]);

        descH = MemHandleNew(descSize);
        exgSocket.description = MemHandleLock(descH);
        StrCopy(exgSocket.description, prefix);
        MemHandleUnlock(prefixH);
        DmReleaseResource(prefixH);
```

Continued

Listing 14-1 *(continued)*

```
        if (record.fields[libFieldFirstName]) {
            StrCat(exgSocket.description,
                   record.fields[libFieldFirstName]);
            if (record.fields[libFieldLastName])
                StrCat(exgSocket.description, " ");
        }

        if (record.fields[libFieldLastName])
            StrCat(exgSocket.description,
                   record.fields[libFieldLastName]);

    } else {
        // Use "a book" for the description.
        prefixH = DmGetResource(strRsc,
                                NoAuthorBeamString);
        prefix = (Char *) MemHandleLock(prefixH);
        descSize = StrLen(prefix) + sizeOf7BitChar('\0');

        descH = MemHandleNew(descSize);
        exgSocket.description = MemHandleLock(descH);
        StrCopy(exgSocket.description, prefix);
        MemHandleUnlock(prefixH);
        DmReleaseResource(prefixH);
    }

    // Truncate the description if it's too long.
    if (descSize > 0) {
        newDescSize = descSize;
        WinGetDisplayExtent(&descWidth, &ignoreHeight);
        FntCharsInWidth(exgSocket.description, &descWidth,
                        (Int16 *) &newDescSize, &descFit);

        if (newDescSize > 0) {
            if (newDescSize != descSize) {
                exgSocket.description[newDescSize] =
                    nullChr;
                MemHandleUnlock(descH);
                MemHandleResize(descH, newDescSize +
                                sizeOf7BitChar('\0'));
                exgSocket.description =
                    MemHandleLock(descH);
            }
        } else {
            MemHandleFree(descH);
        }
        descSize = newDescSize;
    }
```

```
        // Create a filename from the description.
        nameH = MemHandleNew(descSize + sizeOf7BitChar('.') +
                             libFileExtensionLength +
                             sizeOf7BitChar('\0'));
        exgSocket.name = MemHandleLock(nameH);
        StrCopy(exgSocket.name, exgSocket.description);
        StrCat(exgSocket.name, ".");
        StrCat(exgSocket.name, libFileExtension);

        exgSocket.target = libCreatorID;

        // Beam the record.
        error = ExgPut(&exgSocket);
        if (! error) {
            packedH = DmQueryRecord(db, recordNum);
            packed = MemHandleLock(packedH);
            error = BeamData(&exgSocket, packed,
                             MemHandleSize(packedH));
            MemHandleUnlock(packedH);
        }

        ExgDisconnect(&exgSocket, error);

        if (nameH) {
            MemHandleUnlock(nameH);
            MemHandleFree(nameH);
        }

        if (descH) {
            MemHandleUnlock(descH);
            MemHandleFree(descH);
        }

    } else {
        FrmAlert(NoDataToBeamAlert);
    }

    MemHandleUnlock(recordH);
}
```

After setting the description, the sending application needs to set at least one of the following three fields in the socket structure:

- `target`
- `name`
- `type`

Setting `target` to the creator ID of a specific application (usually the same as the sending application's creator ID) tells the Exchange Manager to send the appropriate launch codes to that application so it can receive the data. This is perfectly sufficient for an application that needs to send data only to another copy of the same application that is already installed on another Palm OS handheld. However, setting `target` does prevent any other application from receiving the data on the other end of the transfer, so if your application needs to be able to send data to other programs, you are better off setting the `name` or `type` fields instead.

If you want to describe the outgoing data using a particular file extension, set the `name` field in the socket structure to point to a string containing an appropriate file name. For example, the following code sets the `name` field to send a file with a `.txt` extension:

```
exgSocket.name = "myRecord.txt";
```

Librarian uses the `description` of a record for its file name, tacking a `.lib` file extension onto the end of the string to form the `name`:

```
nameH = MemHandleNew(descSize + sizeOf7BitChar('.') +
                     libFileExtensionLength +
                     sizeOf7BitChar('\0'));
exgSocket.name = MemHandleLock(nameH);
StrCopy(exgSocket.name, exgSocket.description);
StrCat(exgSocket.name, ".");
StrCat(exgSocket.name, libFileExtension);
```

The `sizeOf7BitChar` macro that appears throughout LibBeamRecord is defined in the Palm OS header file `CharAttr.h` as follows:

```
#define  sizeOf7BitChar(c)  1
```

At first glance, this macro may seem a waste of typing time. However, it helps a great deal when one is trying to write self-documenting code. Many uses of `sizeof` result in a size of 2 bytes (`sizeof('\0')`, for example), because `sizeof` treats many character constants as `int` values instead of `char` values. The `sizeOf7BitChar` macro is safer to use than `sizeof` when assembling string values, and provides a clearer picture of what is going on in the code than something like the following:

```
nameH = MemHandleNew(descSize + 1 +
                     libFileExtensionLength + 1);
```

Instead of specifying a file name, you may also specify a MIME type by pointing the socket's `type` parameter at a string describing the appropriate MIME type. The following example sets a socket's `type` to an HTML text document:

```
exgSocket.type = "text/html";
```

Once you have a `description` for the data to be transferred, and a `target`, `name`, or `type`, you may proceed to initiate the data transfer. However, there are some optional fields in the socket structure you may wish to consider setting:

- `length`. If you have calculated the length of the data to transfer in advance, setting `length` to equal the total length of the data in bytes can preempt the transfer if the receiving device does not have enough memory to receive the data object. Instead of having to transfer a significant part of a large piece of data before failing because of memory constraints, the receiving device can look at the `length` before transfer even begins and determine whether there is enough space for the object. By using the `length` field, you can prevent the user from having to wait a long time for an unsuccessful data transfer.
- `localMode`. Setting `localMode` to `1` prevents the application from sending the data out via the handheld's infrared port, and instead loops the data back to the local device. This is primarily useful when you are debugging the beaming features of an application, because it means you can debug most of the beaming process using only one handheld, or just the Palm OS Emulator.
- `noGoTo`. Setting `noGoTo` to `1` prevents the receiving application from displaying the record that was just transferred. This setting works only when `localMode` is also set to `1`. You can use `noGoTo` to export data to another application on the local handheld, without actually launching the other program.

The built-in Memo Pad application is registered to handle data with a file extension of `.txt`. If you want to export text data to the Memo Pad, set the socket's `name` to a file name with a `.txt` extension, and then set `localMode` to `1`. If you want to export the data and remain in the current application, you should also set `noGoTo` to `1` instead of jumping straight to the Memo Pad.

Beginning a transfer with ExgPut

The **ExgPut** function initiates a data transfer. Pass a pointer to the exchange socket structure as an argument to **ExgPut**:

```
Err error = ExgPut(&exgSocket);
```

If **ExgPut** does not return an error, the connection was successful, and you must follow up with a call to either **ExgSend** or **ExgDisconnect**. The **ExgPut** function displays a dialog box, pictured in Figure 14-2, to let the user know that the application is busy assembling data to export.

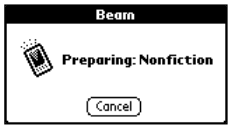


Figure 14-2: The ExgPut function displays this dialog box while the application is busy putting together data for a beaming operation.

Sending data with ExgSend

The **ExgSend** function does the actual work necessary to send data across the connection formed by **ExgPut**. Here is the prototype for **ExgSend**:

```
UInt32 ExgSend(ExgSocketPtr socketP, const void * const bufP,
               const UInt32 bufLen, Err *err)
```

The `socketP` parameter is a pointer to the exchange socket structure to use for the transfer. The `bufP` parameter should point to the start of the data to transmit, and `bufLen` should contain the length of the data in that buffer, in bytes. You can check for errors in the transfer process by passing a pointer to an `Err` type variable in the `err` parameter.

The **ExgSend** function returns the number of bytes of data that it successfully sent. You must call **ExgSend** in a loop, changing the `bufP` and `bufLen` parameters to reflect the remaining data in the buffer at each call. The **ExgSend** function may not send all the data that you instruct it to send because of connection problems between the two devices, instead requiring multiple attempts to send the data.

Wrapping **ExgSend** in its own routine is the easiest way to use it. As an example of one approach to looping over **ExgSend**, here is the **BeamData** function from Librarian:

```
static Err BeamData (ExgSocketType *exgSocket, void *buffer,
                     UInt32 bytes)
{
    Err  error = 0;


    while ( (! error) && (bytes > 0) ) {
        UInt32  bytesSent = ExgSend(exgSocket, buffer, bytes,
                                    &error);
        bytes -= bytesSent;
        buffer = ( (Char *) buffer) + bytesSent;
    }

    return error;
}
```

To make your data transfers more efficient, try to call ExgSend as few times as possible. If possible, allocate a large buffer and send all of the buffer's data at once.

Ending a transfer with ExgDisconnect

After you have finished sending data with **ExgSend**, call **ExgDisconnect** to sever the connection:

```
ExgDisconnect(&exgSocket, error);
```

The first argument to **ExgDisconnect** is a pointer to the exchange socket used for the transfer. For the second argument, you should pass whatever error value has been returned by calls to **ExgPut** and **ExgSend**, which lets the Exchange Manager know what kind of cleanup it needs to perform and which dialog boxes it should display to the user to indicate either a completed transfer or one that failed because of some sort of interruption.

Beaming multiple records

It might be useful to allow an application to send many records all at once. For example, the built-in applications that support categories, and Librarian, all present the option to beam an entire category of records. There are a number of ways to beam multiple records, but the primary thing to remember when designing data transfer for multiple records is that the application must be able to tell multiple records from single records and receive them properly. Librarian uses a very simple system to differentiate single-record beaming from category beaming. Single records have a file extension of `.lib`, and beamed categories have an extension of `.lbc`.

Librarian's **LibBeamCategory** function is responsible for beaming a whole category full of records. Other than the data that **LibBeamCategory** sends, the function differs very little from **LibBeamRecord**. The **LibBeamCategory** function sends the following information:

- A `UInt16` value containing the total number of records in the transfer
- For each record in the category:
  - A `UInt16` value containing the length of the record in bytes
  - The record's actual data

Listing 14-2 presents the **LibBeamCategory** function in its entirety.

Listing 14-2: **Librarian's LibBeamCategory function**

```
void LibBeamCategory (DmOpenRef db, UInt16 category)
{
    Err      error;
    Char     desc[dmCategoryLength];
    UInt16   index;
    Boolean  foundAtLeastOneRecord;
    ExgSocketType  exgSocket;
    UInt16   mode;
    LocalID  dbID;
    UInt16   cardNo;
    Boolean  databaseReopened;
    UInt16   numRecords;
```

Continued

Listing 14-2 *(continued)*

```
LibPackedDBRecord  *packed;
MemHandle  packedH;


// If the database is currently opened to show private
// records, reopen it with private records hidden to
// prevent private records from being accidentally sent
// along with a large batch of normal records. Private
// records should only be sent one at a time.
DmOpenDatabaseInfo(db, &dbID, NULL, &mode, &cardNo, NULL);
if (mode & dmModeShowSecret) {
    db = DmOpenDatabase(cardNo, dbID, dmModeReadOnly);
    databaseReopened = true;
}
else
    databaseReopened = false;

// Verify that there is at least one record in the
// category. It's possible to just use
// DmNumRecordsInCategory for this purpose, but that
// function has to look over the entire database, which
// can be slow if there are many records. This technique
// is quicker, since it stops searching at the first
// successful match.
index = 0;
foundAtLeastOneRecord = false;
while (true) {
    if (DmSeekRecordInCategory(db, &index, 0,
                               dmSeekForward,
                               category) != 0)
        break;

    foundAtLeastOneRecord =
        (DmQueryRecord(db, index) != 0);
    if (foundAtLeastOneRecord)
        break;

    index++;
}

// If at least one record exists in the category, beam the
// category.
if (foundAtLeastOneRecord) {
    // Initialize the exchange socket structure to zero.
    MemSet(&exgSocket, sizeof(exgSocket), 0);

    // Assemble a description of the record to send. This
```

```
// description is displayed by the system send and
// receive dialog boxes on both the sending and
// receiving devices.
CategoryGetName(db, category, desc);
exgSocket.description = desc;

// Create a filename from the description.
exgSocket.name = MemPtrNew(StrLen(desc) +
                    sizeOf7BitChar('.') +
                    sizeof(libCategoryExtension) +
                    sizeOf7BitChar('\0') );
if (exgSocket.name) {
    StrCopy(exgSocket.name, desc);
    StrCat(exgSocket.name, ".");
    StrCat(exgSocket.name, libCategoryExtension);
}

exgSocket.target = libCreatorID;

// Initiate transfer.
error = ExgPut(&exgSocket);
if (! error) {
    // Now use DmNumRecordsInCategory to get the
    // number of records to beam, since it's certain
    // at this point that the category will be beamed.
    numRecords = DmNumRecordsInCategory(db, category);

    // Send the number of records across first.
    error = BeamData(&exgSocket, &numRecords,
                    sizeof(numRecords));

    index = dmMaxRecordIndex;
    while ( (! error) && (numRecords-- > 0) ) {
        UInt16  seekOffset = 1;
        UInt16  recordSize;


        // Be sure to check the last record instead of
        // skipping over it.
        if (index == dmMaxRecordIndex)
            seekOffset = 0;

        error = DmSeekRecordInCategory(db, &index,
            seekOffset, dmSeekBackward, category);
        if (! error) {
            packedH = DmQueryRecord(db, index);
            ErrNonFatalDisplayIf(! packedH,
                "Couldn't query record.");
```

Continued

Listing 14-2 *(continued)*

```
                    // Send the size of the record.
                    recordSize = MemHandleSize(packedH);
                    error = BeamData(&exgSocket, &recordSize,
                                     sizeof(recordSize));

                    // Send the record itself.
                    if (! error) {
                        packed = MemHandleLock(packedH);
                        error = BeamData(&exgSocket, packed,
                                         recordSize);
                        MemHandleUnlock(packedH);
                    }
                }
            }
        }

        ExgDisconnect(&exgSocket, error);

        // Free the filename string to prevent a memory leak.
        MemPtrFree(exgSocket.name);

    } else {
        FrmAlert(NoDataToBeamAlert);
    }

    if (databaseReopened)
        DmCloseDatabase(db);
}
```

Customizing the Beam Acceptance Dialog Box

The first launch code that an application registered to receive beamed data receives is `sysAppLaunchCmdAskUser`. If the application simply ignores this launch code, no harm is done; the Exchange Manager will present a default dialog box prompting the user to either accept or reject the incoming data. This dialog box is pictured in Figure 14-3.

Figure 14-3: This default dialog box prompts the user to accept or reject incoming beamed data.

Quite often, the default dialog box is perfectly fine; most applications do not require any more than a simple "yes" or "no" when they ask whether or not beamed data should be accepted. However, if you would like to do something more interesting with the incoming data, such as assign it to a specific category, the application needs to handle `sysAppLaunchCmdAskUser` and present a custom dialog box to the user. Librarian, when running on Palm OS 3.5 or later, displays such a custom dialog box, pictured in Figure 14-4, which allows the user to move incoming data to a specific category.

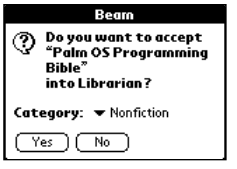


Figure 14-4: Librarian, along with the built-in applications in Palm OS 3.5, uses a custom dialog box to allow the user to assign a category to incoming records.

A `sysAppLaunchCmdAskUser` launch code comes with a parameter block that is a pointer to an `ExgAskParamType` structure. This structure is defined as follows in `ExgMgr.h`:

```
typedef struct {
    ExgSocketPtr  socketP;
    ExgAskResultType  result;
    UInt8  reserved;
} ExgAskParamType;
```

The `ExgAskParamType` structure contains a pointer to the exchange structure used to transfer the data. You should set the `result` field of the `ExgAskParamType` structure to one of the following values, depending on how you want the Exchange Manager to proceed:

- `exgAskDialog` presents the default dialog box to the user
- `exgAskOk` pretends that the user tapped on the OK button in the default dialog box
- `exgAskCancel` pretends that the user tapped on the Cancel button in the default dialog box

You do not need to provide your own replacement for the default beam acceptance dialog box at all, and instead simply use the `sysAppLaunchCmdAskUser` as an opportunity to accept or reject incoming data based on criteria other than user input. This kind of scenario could work for constant, low-bandwidth infrared traffic between two copies of a game that is played via IR, although for efficiency's sake, you are probably better off using the IR Library to provide this kind of "always on" connection between two handhelds.

Setting the `result` takes care of the basic function of the default dialog box, but what about adding other return values, such as category selection? For this, you have the `appData` field of the socket structure contained within the launch code's parameter block. The `appData` field is a `UInt32` value that may contain any data you wish. When the application handles the `sysAppLaunchCmdReceiveData` launch code (see below), it may then take action based on the value of the `appData` field.

Setting the category of incoming records

Starting with Palm OS 3.5, the Exchange Manager provides the **ExgDoDialog** function, which makes it easy to display a default beam acceptance dialog box containing a category selector. The 3.5 versions of the built-in applications use this dialog box, and so does Librarian. Here is the relevant section of Librarian's **PilotMain** function as an example of how to prompt the user for a category:

```
UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
    Err        error;
    DmOpenRef  db;


    switch (cmd) {
        case sysAppLaunchCmdExgAskUser:
            // If Librarian is not already running, open its
            // database.
            if (! (launchFlags & sysAppLaunchFlagSubCall)) {
                error = LibGetDatabase(&db, dmModeReadWrite);
            } else {
                db = gLibDB;
            }

            if (db != NULL) {
                CustomBeamDialog(db,
                                 (ExgAskParamType *) cmdPBP);
                if (! (launchFlags & sysAppLaunchFlagSubCall))
                    error = DmCloseDatabase(db);
            }
            break;

            // Other launch codes omitted
    }

    return 0;
}
```

If Librarian is not already running, which **PilotMain** determines by looking at the launch code's flags for a `sysAppLaunchFlagSubCall` flag, **PilotMain** opens Librarian's database. Access to the database is required, because the database's application info block contains all the strings for Librarian's category names.

Whether the database is open or closed, **PilotMain** then throws the ball to **CustomBeamDialog**, listed below, to handle most of the details:

```
Err CustomBeamDialog (DmOpenRef db, ExgAskParamPtr askInfo)
{
    ExgDialogInfoType  exgInfo;
    Err      error;
    Boolean  result;
    UInt32   romVersion;


    // The custom category-enabled dialog box is available only
    // on Palm OS 3.5 or later.
    FtrGet(sysFtrCreator, sysFtrNumROMVersion, &romVersion);
    if (romVersion <
        sysMakeROMVersion(3,5,0,sysROMStageRelease,0))
        return 1;

    // Set the default category to Unfiled.
    exgInfo.categoryIndex = dmUnfiledCategory;

    // Store the database reference for use by the event
    // handler.
    exgInfo.db = db;

    // Display the custom dialog box.
    result = ExgDoDialog(askInfo->socketP, &exgInfo, &error);

    if (! error && result) {
        // Accept the data; pretend that the user tapped OK.
        askInfo->result = exgAskOk;

        // Stuff the category index into the appData field.
        askInfo->socketP->appData = exgInfo.categoryIndex;
    } else {
        // Reject the data; pretend that the user tapped
        // Cancel.
        askInfo->result = exgAskCancel;
    }

    return error;
}
```

The **CustomBeamDialog** function first checks the version of the operating system. Because **ExgDoDialog** and the custom dialog box resource it uses do not exist before Palm OS 3.5, **CustomBeamDialog** returns immediately on earlier versions, allowing the Exchange Manager to display the default dialog box without any category selection capability.

After checking the Palm OS version, **CustomBeamDialog** declares and initializes `exgInfo`, an `ExgDialogInfoType` structure, with the default category that should appear in the custom dialog box and an open reference to Librarian's database so the dialog box can display the names of the categories in its pop-up list. The `ExgDialogInfoType` structure, declared in `ExgMgr.h`, looks like this:

```
typedef struct {
    UInt16  version;
    DmOpenRef  db;
    UInt16  categoryIndex;
} ExgDialogInfoType;
```

The `version` field of `ExgDialogInfoType` represents the version of the structure itself. This value is for possible future changes to the structure; for now, just use a value of `0` for `version`.

Once `exgInfo` is initialized, **CustomBeamDialog** calls the **ExgDoDialog** function, passing in a pointer to the exchange socket structure for the transfer, a pointer to `exgInfo`, and a pointer to an `Err` value to receive any errors encountered while displaying the dialog box. The **ExgDoDialog** function returns a `Boolean` result: `true` if the user tapped on the dialog box's OK button, `false` if the user tapped on Cancel. If the return result was `true`, and no errors occurred, **CustomBeamDialog** sets the `result` field of the launch code's parameters to mimic a tap on OK in the default beam dialog box. Also, **CustomBeamDialog** fills the socket structure's `appData` field with the category selected in the custom dialog box, which is stored in the `categoryIndex` field of `exgInfo`. A `false` return value from **ExgDoDialog** causes **CustomBeamDialog** to act as if the user tapped on Cancel in the default dialog box.

Receiving Data

When the Exchange Manager receives data that is targeted to a particular application's creator ID, or that has a file extension or MIME type registered by an application, the manager sends a `sysAppLaunchCmdRecieveData` launch code to the appropriate application. Handling this launch code allows an application to receive beamed data.

The application handling `sysAppLaunchCmdReceiveData` may or may not be running when it receives the launch code, so it must open its database if it is not already the active application. Librarian takes care of this step in its **PilotMain** routine, and then hands off processing of the incoming data to another function:

```
UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
    case sysAppLaunchCmdExgReceiveData:
        // If Librarian is not already running, open its
        // database.
        if (! (launchFlags & sysAppLaunchFlagSubCall)) {
```

```
            error = LibGetDatabase(&db, dmModeReadWrite);
        } else {
            db = gLibDB;
            FrmSaveAllForms();
        }

        if (db != NULL) {
            error = LibReceiveData(db,
                                   (ExgSocketType *) cmdPBP);
            if (! (launchFlags & sysAppLaunchFlagSubCall))
                error = DmCloseDatabase(db);
        }
        break;
}

return 0;
}
```

The function that takes care of receiving data in Librarian is **LibReceiveData**, which is shown in Listing 14-3.

Listing 14-3: **Librarian's LibReceiveData function**

```
Err LibReceiveData (DmOpenRef db, ExgSocketType *exgSocketP)
{
    Err     error;
    UInt16  index = 0;
    Char    *startOfExtension;
    Boolean  singleRecord = false;
    UInt16  numRecords;
    UInt16  recordSize;


    // Determine whether the input stream contains a single
    // record or an entire category by looking at the file
    // extension.
    if (exgSocketP->name) {
        startOfExtension = exgSocketP->name +
            StrLen(exgSocketP->name) - libFileExtensionLength;
        if (StrCaselessCompare(startOfExtension,
                               libFileExtension) == 0)
            singleRecord = true;
    }

    // Accept connection from remote device.
    error = ExgAccept(exgSocketP);

    // Import records from data stream.
    if (! error) {
```

Continued

Listing 14-3 *(continued)*

```
        if (singleRecord) {
            // Import a single record.
            error = LibImportRecord(db, exgSocketP,
                                    libEntireStream, &index);
        } else {
            // Import a whole category, starting with the
            // number of records in the transfer.
            ExgReceive(exgSocketP, &numRecords,
                       sizeof(numRecords), &error);
            while ( (! error) && (numRecords-- > 0) ) {
                // Retrieve the size of the next record.
                ExgReceive(exgSocketP, &recordSize,
                           sizeof(recordSize), &error);

                // Import the record.
                if (! error)
                    error = LibImportRecord(db, exgSocketP,
                        recordSize, &index);
            }
        }

        // Disconnect the transfer.
        ExgDisconnect(exgSocketP, error);
    }

    // Set the socket structure's goTo information so the
    // system can fire off a sysAppLaunchCmdGoTo launch code
    // to open the newly transmitted record in Librarian.
    if (! error) {
        DmRecordInfo(db, index, NULL,
                     &exgSocketP->goToParams.uniqueID, NULL);
        DmOpenDatabaseInfo(db, &exgSocketP->goToParams.dbID,
            NULL, NULL, &exgSocketP->goToParams.dbCardNo,
            NULL);
        exgSocketP->goToParams.recordNum = index;
        exgSocketP->goToCreator = libCreatorID;
    }

    return error;
}
```

Receiving records is a four-step process:

1. Call **ExgAccept** to accept the connection.
2. Call **ExgReceive** within a loop to retrieve the data.

3. Call **ExgDisconnect** to end the transfer.

4. Set up `goTo` parameters to display the transferred record.

Accepting a connection with ExgAccept

The **ExgAccept** function is very similar to **ExgPut**: it merely opens a connection, and you must call **ExgReceive** or **ExgDisconnect** next to either retrieve data or close the connection. Pass a pointer to the exchange socket structure to **ExgAccept**:

```
Err error = ExgAccept(&exgSocket);
```

Receiving data with ExgReceive

If **ExgAccept** did not return an error, your application may now start receiving data using the **ExgReceive** function. Just as with **ExgSend**, you must call **ExgReceive** repeatedly from within a loop to retrieve all the incoming data. The prototype for **ExgReceive** is nearly identical to that for **ExgSend**:

```
UInt32 ExgReceive(ExgSocketPtr socketP, void *bufP,
                  const UInt32 bufLen, Err *err)
```

The **ExgReceive** function returns the number of bytes actually read from the incoming data stream. The easiest way to use **ExgReceive** is from within another function that contains the loop necessary to make **ExgReceive** work. Librarian uses the **LibImportRecord** function for this purpose:

```
Err LibImportRecord (DmOpenRef db, ExgSocketType *exgSocketP,
                     UInt32 bytes, UInt16 *indexP)
{
    Char     buffer[libImportBufferSize];
    Err      error;
    UInt16   index = 0;
    UInt16   insertIndex;
    UInt32   bytesReceived;
    MemHandle  recordH = NULL;
    Char     *record;
    UInt32   recordSize = 0;
    MemHandle  packedH;
    LibPackedDBRecord  *packed;
    Boolean  allocated = false;
    UInt16   category;


    do {
        UInt32  bytesToRead = min(bytes, sizeof(buffer));


        bytesReceived = ExgReceive(exgSocketP, buffer,
```

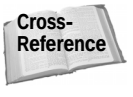
```
                                    bytesToRead, &error);
        bytes -= bytesReceived;
        if (! error) {
            if (! recordH)
                recordH = DmNewRecord(db, &index,
                                      bytesReceived);
            else
                recordH = DmResizeRecord(db, index,
                    recordSize + bytesReceived);

            if (! recordH) {
                error = DmGetLastErr();
                break;
            }

            allocated = true;
            record = MemHandleLock(recordH);
            error = DmWrite(record, recordSize, buffer,
                            bytesReceived);
            MemHandleUnlock(recordH);

            recordSize += bytesReceived;
        }

    } while ( (! error) && (bytesReceived > 0) &&
              (bytes > 0) );

    if (recordH) {
        DmReleaseRecord(db, index, true);

        // Grab the category for the new record from the
        // socket's appData field.
        category = exgSocketP->appData;

        // Put the record in the proper category.
        if (category) {
            UInt16  attr;


            // Get the record's attributes.
            error = DmRecordInfo(db, index, &attr, NULL,
                                 NULL);

            // Set the category and mark the record dirty.
            if ((attr & dmRecAttrCategoryMask) != category) {
                attr &= ~dmRecAttrCategoryMask;
                attr |= category | dmRecAttrDirty;
                error = DmSetRecordInfo(db, index, &attr,
                                        NULL);
            }
        }
```

```
            // Move the record to its proper sort position.
            packedH = DmQueryRecord(db, index);
            packed = MemHandleLock(packedH);
            insertIndex = LibFindSortPosition(db, packed);
            error = DmMoveRecord(db, index, insertIndex);
            if (! error)
                index = insertIndex - 1;
            MemHandleUnlock(packedH);
        }

        if (error && allocated)
            DmRemoveRecord(db, index);

        *indexP = index;
        return error;
    }
```

The **LibImportRecord** function begins by creating a new record with **DmNew Record**; then it progressively calls **ExgReceive** to fill the new record, resizing the record as necessary with **DmResizeRecord**. Once the record has been successfully retrieved, **LibImportRecord** checks the `appData` field of the exchange socket structure to see if the user assigned a new category to the record using the custom beam acceptance dialog box. Because the Unfiled category has a value of `0`, **LibImport Record** will take no action when the `appData` field equals `0`, resulting in the record landing in the Unfiled category. Otherwise, **LibImportRecord** assigns the record to the category specified in `appData`.

Making many calls to ExgReceive using small buffers is better than allocating a really large buffer to receive incoming data, because the application receiving the data is not necessarily the currently running application. The active application is already occupying a great deal of dynamic memory with its own variables, so there might not be a lot of space available for ExgReceive to do its work. If you have verified that your application is indeed the active application, you can get away with allocating a large buffer, but in general, you should stick with smaller buffers when receiving data than you use when sending data.

One more step remains. As it currently stands, the newly created record is the first record in the database (index equal to `0`), which is not necessarily where the record belongs. The **LibImportRecord** function retrieves a handle to the new record with **DmQueryRecord**, and then finds its proper sort position by calling Librarian's **LibFindSortPosition** function, which is just a wrapper for the Palm OS function **DmFindSortPosition**. Once the record's real location is determined, **LibImport Record** moves the record to the correct location with **DmMoveRecord**.



See Chapter 13, "Manipulating Records," for more information about creating, resizing, sorting, and moving records.

Ending a transfer with ExgDisconnect

After receiving all the incoming data, your application should end the transfer by calling **ExgDisconnect**, passing it a pointer to the exchange socket structure and the error value, if any, received from any **ExgAccept** or **ExgReceive** calls:

```
ExgDisconnect(&exgSocket, error);
```

Setting up goTo parameters

After receiving a record, your application should modify the socket structure's `goToParams` and `goToCreator` fields so the Exchange Manager knows which record to display, and in what application it should display it. Librarian's **LibReceiveData** routine takes care of this task with the following lines of code:

```
DmRecordInfo(db, index, NULL,
             &exgSocketP->goToParams.uniqueID, NULL);
DmOpenDatabaseInfo(db, &exgSocketP->goToParams.dbID,
    NULL, NULL, &exgSocketP->goToParams.dbCardNo, NULL);
exgSocketP->goToParams.recordNum = index;
exgSocketP->goToCreator = libCreatorID;
```

Receiving multiple records

Librarian can tell whether it is receiving a single record or a category by the file extension of the incoming data object (see "Beaming multiple records" earlier in this chapter). If a single record is on its way in, the **LibReceiveData** function just calls **LibImportRecord** to loop over the data stream and extract a record from it.

If **LibReceiveData** determines that a whole category is inbound, it calls **ExgReceive** to pull the two-byte number of records included in the transfer into the `numRecords` variable:

```
ExgReceive(exgSocketP, &numRecords, sizeof(numRecords),
           &error);
```

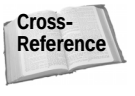
Retrieving the number of incoming records allows **LibReceiveData** to set up a loop for retrieving individual records from the data stream. In each iteration of the loop, **LibReceiveData** calls **ExgReceive** to grab the two-byte record size number that precedes each record's actual data, and then passes that size to **LibImportRecord** to retrieve the record itself:

```
ExgReceive(exgSocketP, &numRecords,
           sizeof(numRecords), &error);
while ( (! error) && (numRecords-- > 0) ) {
    // Retrieve the size of the next record.
    ExgReceive(exgSocketP, &recordSize, sizeof(recordSize),
               &error);
```

```
    // Import the record.
    if (! error)
        error = LibImportRecord(db, exgSocketP, recordSize,
                                &index);
}
```

Displaying Beamed Records

The final step in receiving a beamed record is displaying that record to the user. In order to accomplish this task, the Exchange Manager sends a `sysAppLaunchCmdGoTo` launch code to the application, based on the `goToParams` and `goToCreator` fields in the transfer's socket structure, which specifies the record to display. The `sysAppLaunchCmdGoTo` launch code is actually shared between the Exchange Manager and the global find facility, which both use it to display specific data within an application.



See the "Implementing the Global Find Facility" section of Chapter 13, "Manipulating Records," for more information about how to handle the `sysAppLaunchCmdGoTo` launch code.

Unlike the find facility, the Exchange Manager generally does not need to highlight specific information within a record that it displays, and so the `matchPos` and `matchFieldNum` fields in the `sysAppLaunchCmdGoTo` launch code's parameter block are not set. However, you may pass an application-defined value to the launch code parameter block's `matchCustom` field by setting the `matchCustom` field in the socket's `goToParams` when you are handling the `sysAppLaunchCmdReceiveData` launch code:

```
UInt32  myCustomValue;

// Set myCustomValue here.

exgSocketP->goToParams.matchCustom = myCustomValue;
```

Debugging Beaming

Other than setting the exchange socket's `localMode` field to `1`, there are a couple of other tricks that can make debugging beaming operations much easier. Both of these methods are special developer Graffiti shortcuts that you can make by writing the Graffiti shortcut stroke in the Graffiti area of a Palm OS handheld or the Palm OS Emulator, followed by a period (two dots) and a letter. The shortcuts are pictured in Figure 14-5.

Figure 14-5: Helpful beam debugging shortcuts: toggle beam loopback (left), and serial port IR (right)

A Graffiti shortcut character, followed by a period and the letter *t*, toggles `localMode` on and off.

In an application that supports beaming records, you can use the *t* shortcut as a quick and dirty record copier. Turn on local beaming, and then beam to yourself the record you wish to copy. Just accept the record back into the application, and you now have a duplicate record. Just be sure to make the *t* shortcut a second time when you're done to enable beaming to other devices again.

A Graffiti shortcut character, followed by a period and the letter *s*, sends infrared data to the handheld's serial port. This is primarily useful when you are debugging using POSE, which does not actually have an IR port but can emulate a serial connection using the desktop machine's own serial ports.

Beaming Applications and Databases

The system application launcher allows you to beam applications between Palm OS handhelds. You can also implement application beaming from within your own application, as well as perform a few tricks that the launcher cannot do, such as beaming any arbitrary database on the handheld, regardless of whether it is an application or not.

Beaming databases is similar to beaming any other type of data with the Exchange Manager. Two extra functions are required to make the whole process work: **ExgDBWrite**, and a callback function that **ExgDBWrite** uses to actually send the database's bytes across the connection.

The **ExgDBWrite** function converts a database from its internal format on the handheld to its equivalent `.prc` or `.pdb` file format on the desktop, and then uses a callback function to perform some kind of write operation with the converted database information. Most commonly, this write operation will be used to send the database through the Exchange Manager, but this function might also be used for other purposes, such as to make a backup copy of a database on the handheld. The **ExgDBWrite** function has the following prototype:

```
Err ExgDBWrite (ExgDBWriteProcPtr writeProcP, void* userDataP,
    const char* nameP, LocalID dbID, UInt16 cardNo)
```

The first parameter to **ExgDBWrite** is a pointer to the callback function that will be responsible for sending the actual data. This callback must have the following prototype:

```
Err ExgDBWriteProc (const void *dataP, UInt32 *sizeP,
                    void *userDataP)
```

Next, the **ExgDBWrite** function has a `userDataP` parameter, where you may pass application-defined data to **ExgDBWrite**. This data is also passed along to the callback function's `userDataP` parameter. When using **ExgDBWrite** to beam a database, you should pass a pointer to the exchange socket structure in the `userDataP` parameter.

If `nameP` is not `NULL`, **ExgDBWrite** looks for a database that has a name matching the string that `nameP` points to. You may alternatively specify a value for `dbID` if you know the LocalID of the database to send. In either case, you need to use the `cardNo` parameter to specify the card number where the database resides.

The following simple function takes the name of a database as an argument and sends the corresponding database to another handheld using the Exchange Manager:

```
Err SendDatabase (Char *dbName)
{
    ExgSocketType  exgSocket;
    Err   error;
    Char  name[36];  // max length for a database name (32),
                     // plus a period and three-letter file
                     // extension


    // Initialize the socket structure.
    MemSet(&exgSocket, sizeof(exgSocket), 0);
    StrCopy(name, dbName);
    StrCat(name, ".prc");

    exgSocket.description = dbName;
    exgSocket.name = name;

    // Make a connection to send the database.
    error = ExgPut(&exgSocket);
    if (! error) {
        error = ExgDBWrite(WriteDatabase, &exgSocket, dbName,
                           NULL, 0);
        error = ExgDisconnect(&exgSocket, error);
    }
    return error;
}
```

The operating system is registered to handle the file extensions `.prc` (application database), `.pdb` (record database), and `.pqa` (Palm Query Application). If you specify any of these file extensions for the socket's file name, the system on a receiving Palm OS handheld will automatically try to receive the database that you have sent.

The **WriteDatabase** callback passed to **ExgDBWrite** in the **SendDatabase** function is very similar to Librarian's **BeamData** function but does not need to call **ExgSend** in a loop, because **ExgDBWrite** already calls the callback function multiple times until it sends the entire database or times out. Here is what the **WriteDatabase** function should look like:

```
Err WriteDatabase (const void *buffer, UInt32 *bytes,
                   void *exgSocket)
{
    Err  error;


    ExgSend( (ExgSocketType *) exgSocket, buffer, bytes,
            &error);

    return error;
}
```

Receiving beamed databases

Normally, you do not have to do anything to receive an incoming database if it was sent with the `.prc`, `.pdb`, or `.pqa` file extensions; the system automatically takes care of receiving databases with these extensions. However, if you want an application to receive a database and subject it to special processing, you can use **ExgDBRead** to receive a database with a different file extension. The **ExgDBRead** function has the following prototype:

```
Err ExgDBRead (ExgDBReadProcPtr readProcP,
    ExgDBDeleteProcPtr deleteProcP, void* userDataP,
    LocalID* dbIDP, UInt16 cardNo, Boolean* needResetP,
    Boolean keepDates)
```

The first two parameters to **ExgDBRead** are pointers to callback functions. The first callback, specified by `readProcP`, is the receiving equivalent of the send callback function used by **ExgDBWrite**. The **ExgDBRead** callback calls this callback repeatedly until the entire database has been received or **ExgDBRead** times out. The reading callback function must have the following prototype:

```
Err ReadProc (void* dataP, UInt32* sizeP, void* userDataP)
```

After the read callback function is a delete callback. The **ExgDBRead** function calls the function specified by `deleteProcP` if a database already exists with the same name as the incoming database. The delete callback function is responsible for deleting, renaming, moving, or otherwise dealing with the name conflict. If the delete callback successfully deals with the situation, it returns `true`; otherwise, the callback returns `false`. The delete callback function must have the following prototype:

```
Boolean DeleteProc (const char* nameP, UInt16 version,
    UInt16 cardNo, LocalID dbID, void* userDataP)
```

Understanding the IR Library

The Exchange Manager is a very flexible and easy-to-use interface to the Palm OS infrared transfer facility. However, if you have an application that requires very fine control of the Palm OS infrared system, you will need to use the IR Library, which provides you with access to the protocols underlying the Exchange Manager.

The IR Library is a shared library of functions. In order to access the library, you must first get a reference to it using the **SysLibFind** function:

```
UInt16  refNum;

Err error = SysLibFind(irLibName, &refNum);
```

Many of the functions in the IR Library require the library reference as an argument. For example, the **IrOpen** function (which opens the IR Library, allocates global memory for the IR stack, and reserves system resources for use by the library) may be called as follows to open the library at the maximum negotiated connection speed:

```
UInt32  options = 0;

options |= irOpenOptSpeed115200;
Err error = IrOpen(refNum, options);
```

IR Library functions implement all the required IrDA Data protocols as outlined by the Infrared Data Association, as well as the optional Object Exchange (OBEX) layer. Figure 14-6 shows the hardware and protocol layers of the IrDA stack. Protocol layers toward the top of the diagram are built on top of layers at the bottom of the diagram.

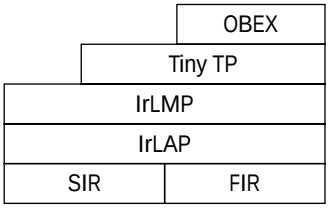


Figure 14-6: The IrDA hardware and protocol stack

The SIR and FIR layer is the hardware portion of the IrDA stack. The SIR (Serial Infrared) layer supports speeds up to 115,200 kbps, and the FIR (Fast Infrared) layer supports speeds up to 4 Mbps. The IrLAP (Infrared Link Access Protocol) layer provides a data pipe between different IrDA devices; the IrLMP (Infrared Link Management Protocol) layer manages multiple IrLAP sessions; and the TinyTP (Tiny Transport Protocol) layer provides a lightweight transfer protocol for higher-level IrDA layers.

OBEX is an optional layer that provides object exchange facilities for beaming typed pieces of data between devices. Much of OBEX is very similar to the Hypertext Transport Protocol (HTTP), only scaled down to provide a bridge between resource-heavy HTTP servers and smaller devices using IrDA. OBEX is intended to make the final hop between the Web and a small device with limited resources. The Exchange Manager is the Palm OS implementation of OBEX, and as such, there are no separate API functions to access the OBEX layer.

The details of programming IrDA are complex enough to warrant their own very large manual; lest this book become *IrDA Programming Bible* instead of *Palm OS Programming Bible*, this is as much IrDA programming information as you will see in this book.

Take a look at the *Palm OS SDK Reference* and *Palm OS Programmer's Companion*, both part of the Palm OS 3.5 SDK included on the CD-ROM attached to this book, for more information about using the IR Library.

The IrDA Web site, located at `http://www.irda.org`, contains full specifications for all the IrDA protocols, as well as extensive implementation guides and testing guidelines. If you want to hack the IrDA layer on a Palm OS handheld, this Web site should be your first stop.

Summary

This chapter showed you how to use the Exchange Manager to send data and applications between Palm OS handhelds, or from a Palm OS handheld to another type of device. After reading this chapter, you should understand the following:

- To enable transfer of data between the Palm OS and other platforms, you can register Palm OS applications to handle data with either a specific file extension or a particular MIME data type.
- The `ExgSocketType` structure contains all the information about an Exchange Manager data transfer, and you set and read its fields throughout the process of beaming data.
- Beaming data requires four steps: initializing an `ExgSocketType` structure, calling **ExgPut** to begin the transfer, calling **ExgSend** within a loop to send the actual data, and calling **ExgDisconnect** to end the transfer.
- If an application is to receive data, it should handle the `sysAppLaunchCmdAskUser`, `sysAppLaunchCmdReceiveData`, and `sysAppLaunchCmdGoTo` launch codes.

- The **ExgDoDialog** function, available starting with Palm OS version 3.5, makes it easy to prompt the user for a category in which to place a beamed record.
- Receiving beamed data requires four steps: accepting the connection with **ExgAccept**, calling **ExgReceive** within a loop to receive the actual data, calling **ExgDisconnect** to end the transfer, and setting up the exchange socket structure's `goTo` parameters to allow the Exchange Manager to display the newly beamed record.
- The Exchange Manager is built on top of OBEX, which is in turn a layer on top of several IrDA protocols, most of which are available for direct access through the Palm OS IR Library.

✦ ✦ ✦

CHAPTER 15

Using the Serial Port

In This Chapter

Understanding Palm OS serial communications

Using the new serial manager

Using the old serial manager

Opening and closing the serial port

Reading and writing data on the serial port

Retrieving and setting serial port values

Part of the Palm Computing platform's success can be attributed to its ability to connect to and share data with other devices using standard methods of communication. The serial port on a Palm OS device is a perfect example of this kind of easy connectivity. Because a Palm OS handheld uses standard RS-232 serial signals, the only obstacle to direct communication between the handheld and another device is finding a cable with the correct wiring for both ends of the connection. You can hook up a wide variety of electronics to a Palm OS handheld, including modems, Global Positioning System (GPS) receivers, and desktop computers, just to name a few. In fact, HotSync technology, the primary method for synchronizing data between the Palm OS and the desktop computer, relies on a serial connection through a cradle to take care of shoving bits back and forth between the two systems.

This chapter shows you how to use the Palm OS serial manager to send and receive data using the handheld's serial port, including an introduction to the hardware and software layers that make up the Palm OS serial communications system.

Understanding Palm OS Serial Communications

The serial port on current Palm OS devices is a slightly stripped-down version of what you might be used to in a desktop computer's serial port. The UART (Universal Asynchronous Receiver and Transmitter) chip in a Palm OS handheld uses the following five external signals:

- SG (signal ground)
- TD (transmit data)

- ✦ RD (receive data)
- ✦ CTS (clear to send)
- ✦ RTS (request to send)

Some signals used by desktop serial ports, such as RI (ring indicator), are not present in the Palm Computing platform's UART. The hardware UART send and receive buffers also hold only 8 bytes apiece, compared with the 16-byte UART buffers commonly found on desktop computers.

The Palm OS serial port supports serial communications at speeds between 300 and 115,200 bps. Another thing to keep in mind when designing Palm OS programs that communicate with other devices through the serial port is that all data entering or leaving the device is arranged in Motorola's big-endian byte order. In other words, multi-byte data types like UInt16 and UInt32 are arranged with their most significant bytes at the lowest address. This is an important distinction when connecting to an Intel-based machine, all of which use little-endian byte ordering. If your program needs to send multi-byte data to an Intel system, you will need to reverse the byte order, either in the handheld application or, preferably, in whatever program serves as the handheld application's counterpart on the desktop.

On the software side, the Palm OS has several layers that make up its serial communications system. Each layer adds to and relies upon the capabilities of the layer beneath it. The following layers compose the Palm OS serial communications stack:

- ✦ **Serial manager.** At the lowest level, the serial manager provides direct control of RS-232 signals and the hardware serial port. This layer allows for byte-level serial input and output, which makes this layer the most flexible for use in custom applications.
- ✦ **Modem manager.** Built directly on top of the serial manager, the modem manager provides a small API for modem dialing and control, which is capable of handling a modem attached either directly to the handheld's serial port or through a Palm modem cable.
- ✦ **Serial Link Protocol (SLP).** Also built on the serial manager, this protocol provides an efficient send-and-receive system for data packets, including CRC-16 error checking. Both the HotSync desktop program and the Palm Debugger use this protocol for communicating with a Palm OS handheld resting in its cradle. The Palm OS also offers an API for your own applications to use SLP called the Serial Link Manager (SLM), which offers a sockets-like implementation of SLP and provides support for remote debugging and Remote Procedure Calls (RPC).
- ✦ **Packet Assembly/Disassembly Protocol (PADP).** Built on the Serial Link Protocol, PADP provides buffered data transmission capabilities for the Desktop Link Protocol, described below. PADP is entirely internal, and your applications do not have access to this layer.

- **Desktop Link Protocol (DLP).** DLP is built on top of PADP and provides remote access to various Palm OS subsystems, including data storage. HotSync technology uses DLP to perform synchronization and to install and back up databases. Though you cannot directly access DLP's features through a Palm OS application, you indirectly make use of it if you write a HotSync conduit for a desktop computer.
- **Connection Management Protocol (CMP).** CMP is built directly on the serial manager layer, and it is another protocol the system uses for negotiating baud rates and exchanging basic information with outside communication software. Only the operating system has access to CMP. You can, however, alter connection profiles used by the Palm OS to connect applications via IR, serial, or network communications using the connection manager.

Figure 15-1 shows how the different layers of the Palm OS serial communications stack relate to one another.

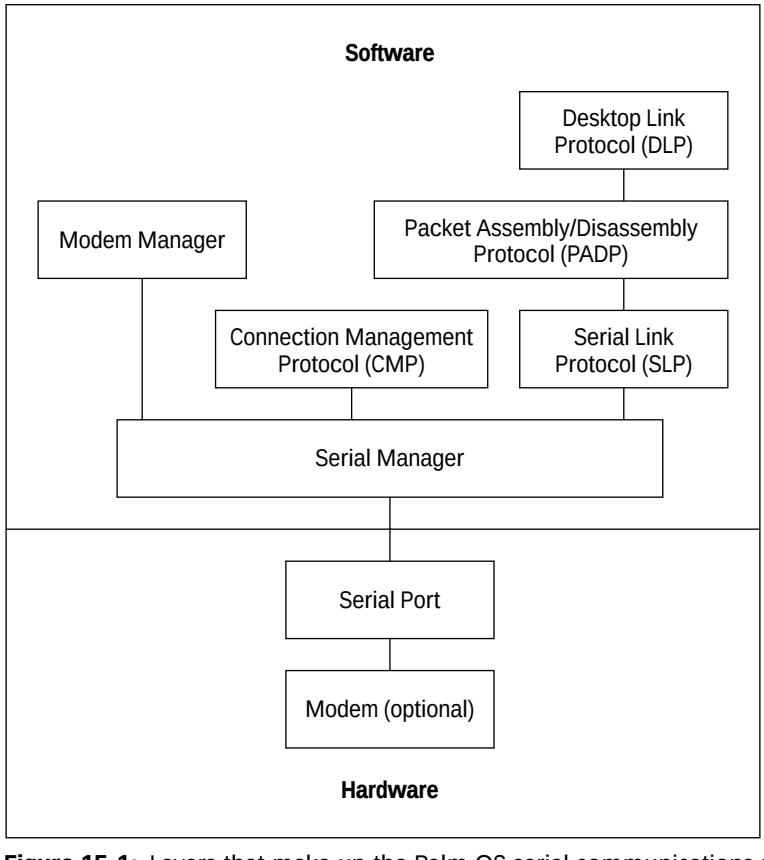


Figure 15-1: Layers that make up the Palm OS serial communications stack

Using the Serial Manager

Since the serial manager is the basic underlying layer for all serial communications on a Palm OS handheld, having the most direct control over the device's hardware, the serial manager is the most flexible way to connect a Palm OS application to another device. Of course, this flexibility comes with a price: you are entirely responsible for implementing your own communications protocols. The higher-level serial protocols available on the Palm OS, such as SLP, provide more advanced communications capabilities than the basic serial manager, but they are rather limited in scope. For example, SLP works well for communicating with a debugger, but it is difficult to adapt to any other purpose. Whatever you connect the handheld to is likely to require its own arbitrary communication format, so handling data transfer at the byte level with the serial manager lets you adapt a Palm OS application to fit the needs of whatever device is attached to the handheld.

The single most important thing to remember about serial communications on the Palm Computing platform is that the serial port is extremely power intensive. An open serial connection drains a handheld's batteries faster than almost any other system on the device.

To prolong battery life, you should keep the serial port open only long enough to perform the necessary data transfer, and then close the port. Leaving the port open for long periods of time will suck the life out of the handheld's batteries, making your application very unpopular with users. You will find various tips on conserving power use during serial communications throughout this chapter.

With the release of Palm OS 3.3, Palm Computing introduced the *new serial manager*. Unlike the original serial manager, the new serial manager can maintain multiple serial connections through different communications devices on a single handheld; the old serial manager can make only a single connection at a time. The new serial manager also allows you to write drivers for both hardware and virtual serial devices, providing an abstraction layer between the serial hardware and the operating system's serial management routines. This new driver-based system is more efficient on devices that have more than one serial device installed, such as those in the Palm III family, whose physical serial and IR ports are both capable of serial communications.

Handhelds that support the new serial manager still understand calls to the old serial manager functions. In such handhelds, the system simply maps the old calls to equivalent routines in the new serial manager. Palm Computing has plans to phase out support for the old serial manager, so this chapter focuses primarily on the new serial manager, presenting a few pointers later on if you need to write code to support an older Palm OS handheld.

If possible, be sure to use the new calls rather than the original serial manager routines, since calling the older functions on a device with the new serial manager installed results in a performance penalty.

Before calling new serial manager functions, you should determine whether the system supports them. Use the following call to **FtrGet** to perform this check:

```
UInt32  value;

Err error = FtrGet(sysFileCSerialMgr, sysFtrNewSerialPresent,
                   &value);
```

If the new serial manager is installed, the `value` parameter will be non-zero, and `error` will have the value `0`, representing no error.

Checking that the operating system version is greater than 3.3 is not a reliable way to ensure that the new serial manager is installed. Not all future Palm OS devices will necessarily include the new serial manager, so checking for the manager directly is a good idea if you want your code to continue working on upcoming versions of the Palm OS.

Using the New Serial Manager

To demonstrate using the new serial manager, this chapter refers to a sample application called Serial Chat, which allows a Palm OS handheld to chat with a connected terminal program on the desktop via the cradle, or even with another handheld connected to the first one with a null modem cable. Figure 15-2 shows Serial Chat in action.

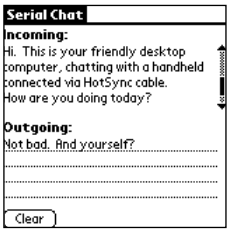


Figure 15-2: The Serial Chat sample application

Serial Chat automatically opens a serial connection when it starts up, and then closes the connection when it exits. Text to send to the desktop may be entered in the Outgoing field at the bottom of the screen; entering a linefeed character sends the contents of the field over the serial connection. Data coming back from the desktop appears in the read-only Incoming field at the top of the screen. The Clear button erases the contents of both fields.

Before you can use the serial manager to send or receive data, you must open an appropriate serial device with either **SrmOpen** or **SrmOpenBackground**, depending on whether you want a foreground or background port connection. A foreground connection, made with the **SrmOpen** function, can read from and write to the port. There can be only one foreground connection at a time. A background connection, made with **SrmOpenBackground**, can receive only incoming data, and if an application makes a foreground connection, the background connection relinquishes the port to the foreground task. Like a foreground connection, only one application or task may have background ownership of the port. Background connections are primarily useful for providing hardware support for an external device, such as a keyboard, which requires only one-way communication from outside the handheld.

To open a connection using either function, you must specify either a *logical port number* or a four-character creator ID that identifies a specific piece of serial hardware on the handheld. Logical port numbers are generic identifiers for different kinds of ports. If you specify a logical port number, the system finds an appropriate port on the handheld and opens it. If you use a creator ID instead, the system attempts to open the specific device identified by the creator ID. The Palm OS header file `SerialMgr.h` defines the following constants for logical port numbers:

```
#define serPortLocalHotSync 0x8000 // Use physical HotSync port
#define serPortCradlePort   0x8000 // Use the RS-232 cradle
                                   // port.
#define serPortIrPort       0x8001 // Use available IR port.
```

You can find useful constants for serial device creator IDs in `SystemResources.h`:

```
#define sysFileCUart328    'u328' // Creator type for '328 UART
                                  //  plug-in
#define sysFileCUart328EZ  'u8EZ' // Creator type for '328EZ
                                  //  UART plug-in
#define sysFileCUart650    'u650' // Creator type for '650 UART
                                      plug-in (IR port on an
                                      upgraded Palm III device)
#define sysFileCVirtIrComm 'ircm' // Creator type for IrComm
                                      virtual port plug-in
```

For compatibility with different versions of the Palm OS, you are better off using a logical port number than the creator ID of a specific hardware port. I ran across this kind of compatibility problem when developing the Serial Chat example program for this chapter, because I had originally used the `u328` code to specify the serial port directly. While this works fine on POSE when it emulates a Palm III running Palm OS 3.3, when I installed the application on my Palm IIIx running Palm OS 3.5, the SrmOpen call at the start of the program returned the error `serErrBadPort`. This is because the IIIx has a DragonBall EZ processor, whose serial UART has a creator ID of `u8EZ` instead of `u328`, which is the code for non-EZ devices like the Palm III. Specifying the logical port ID of `0x8000` ensures that Serial Chat works on devices with either the DragonBall or the DragonBall EZ processor.

Opening the serial port

Specify the logical port number or creator ID as the first argument to **SrmOpen** and **SrmOpenBackground**, both of which take the same parameters. Here is the prototype for **SrmOpen**:

```
Err SrmOpen (UInt32 port, UInt32 baud, UInt16* newPortIdP)
```

After the port identifier, specify an initial baud rate for the connection. The third parameter to **SrmOpen** and **SrmOpenBackground** is a pointer to a variable to receive the *port ID*. The port ID must be passed to other serial manager functions so they can find the open connection and work with it.

A return value of `0` from **SrmOpen** and **SrmOpenBackground** indicates that the port was opened successfully. You should also specifically check for the error value `serErrAlreadyOpen`. If either open function returns this value, the port was successfully opened, but another task is already installed as the foreground or background port owner. In this case, the system increments an open count for the port to keep track of how many tasks are using the port concurrently.

It is possible for two tasks to use the port at the same time, but I do not recommend sharing reads and writes between two applications unless you like vast amounts of pain. Sharing the port is more likely to result in corrupt data and debugging headaches than in anything useful.

If your application receives a `serErrAlreadyOpen` error, it should call **SrmClose** to decrement the open count, and then display an appropriate error message. Failure to call **SrmClose** after a `serErrAlreadyOpen` error will leave the serial port open after your application exits, a surefire way to drain battery power.

Serial Chat calls **SrmOpen** from its **OpenSerial** function to make a foreground connection. The **OpenSerial** function, called from Serial Chat's **StartApplication** routine, looks like this:

```
static Err OpenSerial (void)
{
    Err     error = 0;


    // Open the serial port with an initial baud rate of 9600.
    error = SrmOpen(serPortCradlePort, 9600, &gPortID);
    ErrNonFatalDisplayIf(error == serErrBadPort,
                         "serErrBadPort");
    switch (error) {
        case errNone:
            break;

        case serErrAlreadyOpen:
```

```
                SrmClose(gPortID);
                FrmAlert(SerialBusyAlert);
                return error;
                break;

            default:
                FrmAlert(SerialOpenAlert);
                return error;
                break;
        }

        gConnected = true;

        // Clear the port in case garbage data is hanging around.
        SrmReceiveFlush(gPortID, 100);

        // Code to set connection parameters omitted.

        return error;
    }
```

The **OpenSerial** function calls **SrmOpen** with the `serPortCradlePort` logical port number (`0x8000`), setting an initial baud rate of 9600. Serial Chat uses the global variable `gPortID` to store the port ID obtained by **SrmOpen** for use with other new serial manager functions. After a bit of error checking, including displaying appropriate alerts to the user if there is a problem opening the serial connection, **OpenSerial** sets the global variable `gConnected` to `true`. Serial Chat uses `gConnected` to determine whether or not the application is connected.

Once the connection has been made, **OpenSerial** empties the serial receiving queue with the **SrmReceiveFlush** routine. This step ensures that any stale data remaining in the queue from earlier uses of the serial port are discarded before Serial Chat starts to use the port. The **SrmReceiveFlush** function has two parameters: the port ID returned from **SrmOpen**, and a timeout value:

```
Err SrmReceiveFlush (UInt16 portId, Int32 timeout)
```

When flushing the port, **SrmReceiveFlush** first discards all the waiting data and then waits a number of system ticks equal to `timeout`. If more data arrives at the port before the `timeout` period is up, **SrmReceiveFlush** empties the queue again, resets its timer to the `timeout` value, and waits again. As soon as **SrmReceiveFlush** waits the entire `timeout` period without seeing any more data, the function returns. If you want to just empty the queue once without letting **SrmReceiveFlush** wait for more data, call **SrmReceiveFlush** with a `timeout` value of `0`.

Right after opening the serial port is also a good time to set communications parameters for the connection, such as the number of bits per character and the parity. The code that does this in **OpenSerial** has been omitted in this section; see the "Changing serial port settings" section later in this chapter for more information.

Closing the serial port

Once you are done with the serial port, close it with the **SrmClose** function, which looks like this:

```
Err SrmClose (UInt16 portID)
```

The `portID` parameter of **SrmClose** is the port ID as returned by the **SrmOpen** or **SrmOpenBackground** call that opened the port in the first place. The **SrmClose** function returns `0` if it successfully closes the port, or `serErrBadPort` if the specified port ID is invalid.

Be sure to pair every successful call to SrmOpen or SrmOpenBackground with a call to SrmClose to ensure that the serial port is not left open when your application exits.

Serial Chat wraps a call to **SrmClose** in its **CloseSerial** function, which the application calls in its **StopApplication** routine just before exiting. The **CloseSerial** function looks like this:

```
static void CloseSerial (void)
{
    Err  error;


    error = SrmSendWait(gPortID);
    ErrNonFatalDisplayIf(error == serErrBadPort,
                         "SrmClose: bad port");
    if (error == serErrTimeOut)
        FrmAlert(SerialTimeoutAlert);

    SrmClose(gPortID);

    gConnected = false;
}
```

Before calling **SrmClose**, **CloseSerial** makes a call to **SrmSendWait**. The **SrmSendWait** function waits until all the data in the transmit queue has been sent, and then returns. Calling **SrmSendWait** ensures that all the data sent by your application is transmitted before the application exits; simply closing the port immediately stops transmission and might strand a few bytes in the outgoing queue before they get a chance to leave the port, resulting in lost data.

The **SrmSendWait** function has only one parameter, the port ID as returned by **SrmOpen** or **SrmOpenBackground**. A return value of `0` from **SrmSendWait** means that the function was able to successfully send all the data remaining in the outgoing queue. If **SrmSentWait** times out before it finishes its task, the function returns the `serErrTimeOut` error code, which causes Serial Chat's **CloseSerial** routine to display an alert to inform the user that some data may not have been transmitted.

You can control the timeout value used by SrmSentWait by setting the `ctsTimeout` value with the SrmControl function; see the "Changing serial port settings" section later in this chapter.

Sending data

Sending data over a serial connection is a simple matter of calling **SrmSend**. The **SrmSend** routine has the following prototype:

```
UInt32 SrmSend (UInt16 portId, void *bufP, UInt32 count,
                Err* errP)
```

The first parameter of **SrmSend** is simply the familiar port ID reference obtained with **SrmOpen** or **SrmOpenBackground**. The `bufP` parameter should be a pointer to the data to send, and `count` is the length of that buffer in bytes. If **SrmSend** encounters an error, it returns it through the `errP` parameter.

The **SrmSend** routine returns the number of bytes actually sent. If **SrmSend** successfully transmits all the data in the buffer, the variable pointed to by `errP` is set to `NULL`, and **SrmSend** returns a value equal to the `count` parameter. If `errP` is not `NULL`, check the return value to determine how many bytes **SrmSend** managed to send before it encountered an error.

Serial Chat uses **SrmSend** in its **WriteSerial** function, which is responsible for retrieving the contents of the application's **Outgoing** field and sending it over the serial connection. The **WriteSerial** routine sends a linefeed character after it has sent the field's contents to signal the end of the message. Here is what **WriteSerial** looks like:

```
static void WriteSerial (void)
{
    Err     error;
    FieldType  *field;
    MemHandle  textH;
    Char    *text;
    Char    lineFeed = chrLineFeed;


    // Bail out if not connected.
    if (gConnected == false) return;

    // Retrieve a pointer to the outgoing field's text.
    field = GetObjectPtr(MainOutgoingField);
    textH = FldGetTextHandle(field);
    if (textH) {
        text = MemHandleLock(textH);
```

```
            // Send the contents of the outgoing field.
            SrmSend(gPortID, text, StrLen(text), &error);
            if (error)
                FrmAlert(SerialSendAlert);

            MemHandleUnlock(textH);

            // Send a linefeed character.
            SrmSend(gPortID, &lineFeed, 1, &error);
            if (error)
                FrmAlert(SerialSendAlert);
        }
    }
```

The new serial manager also has some useful utility functions to assist an application with sending data. **SrmSendCheck** checks the transmission queue and returns the number of bytes that have not been sent yet. The prototype for **SrmSendCheck** looks like this:

```
Err SrmSendCheck (UInt16 portId, UInt32* numBytesP)
```

The **SrmSendCheck** function's second parameter is a pointer to a variable that receives the number of bytes left in the queue. If **SrmSendCheck** encounters an error, it returns an appropriate error code. In particular, you should be on the lookout for a `serErrorNotSupported` error, which indicates that checking the status of the outgoing queue is not supported by the serial hardware. Not all serial devices are capable of providing this information.

Another useful function is **SrmSendFlush**, which is the send queue equivalent of **SrmReceiveFlush**, emptying the transmission queue instead of the receiving queue. The **SrmSendFlush** routine takes a single argument, the port ID of the serial connection, and the function returns an error code if it was unable to flush the queue. Unlike **SrmReceiveFlush**, **SrmSendFlush** does not have a timeout feature.

Receiving data

In order to allow an application to receive and process data as it becomes available from the serial port, you need to modify the event loop in your application. Most applications use the constant `evtWaitForever` for the timeout argument to **EvtGetEvent**, which means that the event loop is triggered only when user input places a new event in the queue. If you change the timeout value to an actual number of system ticks, the event loop can be used to service the serial port at regular intervals, while still allowing for user input to be handled. For example, the following call to **EvtGetEvent** sets the timeout period to 100 ticks:

```
EvtGetEvent(&event, 100);
```

Alternatively, if your application needs to receive serial data in a tight loop that does not call EvtGetEvent, you can still process user input with periodic calls to EvtEventAvail to see if any user events are in the queue. These EvtEventAvail calls should probably be no more than a second apart to allow the user ample opportunity to interrupt the application. Sticking the application in a loop that cannot be canceled, particularly when the loop is draining the battery through a serial port connection, is a sure way to annoy users.

With **EvtGetEvent** set up with a timeout value, you can add code to the event loop that checks the serial port for data. As an example, here is the **EventLoop** routine from Serial Chat:

```
static void EventLoop (void)
{
    Err        error;
    EventType  event;
    static UInt32  lastResetTime;


    lastResetTime = TimGetSeconds();
    do {
        // Retrieve an event about once every second.
        EvtGetEvent(&event, 100);

        // Prevent the auto-off timer from putting the handheld
        // into sleep mode by resetting the auto-off timer
        // every 50 seconds.
        if (TimGetSeconds() - lastResetTime > 50) {
            EvtResetAutoOffTimer();
            lastResetTime = TimGetSeconds();
        }

        // Read data from the serial port.
        ReadSerial();

        if (! SysHandleEvent(&event))
            if (! MenuHandleEvent(0, &event, &error))
                if (! ApplicationHandleEvent(&event))
                    FrmDispatchEvent(&event);

    } while (event.eType != appStopEvent);

}
```

Serial Chat's **EventLoop** routine calls the **ReadSerial** function to do the actual receiving of incoming serial data. Before we get to **ReadSerial**, another feature of the **EventLoop** function that we should look at. The **EventLoop** routine calls **EvtResetAutoOffTimer** every 50 seconds to prevent the system auto-off timer

from putting the handheld in sleep mode. Preventing the system from going to sleep means that Serial Chat can continue to receive and display data uninterrupted, even if there has been no user input on the handheld side of the connection. If you plan to use this technique to keep the handheld from sleeping during communication, you need to call **EvtResetAutoOffTimer** at least once a minute, since one minute is the smallest auto-off timer the user can normally set.

The **ReadSerial** function is where Serial Chat does its actual processing of incoming serial data:

```
static void ReadSerial (void)
{
    static Char  buffer[maxFieldLength];
    static UInt16  index = 0;
    Err     error;
    UInt32  bytes;


    if (gConnected == false) return;

    // See if there is anything in the queue.
    error = SrmReceiveCheck(gPortID, &bytes);
    if (error) {
        FrmAlert(SerialCheckAlert);
        return;
    }

    // Make sure the data in the queue won't overflow the
    // buffer. If there is too much data waiting, only
    // retrieve as much data as will fit in the buffer.
    if (bytes + index > sizeof(buffer)) {
        bytes = sizeof(buffer) - index - sizeOf7BitChar('\0');
    }

    // Retrieve data.
    while (bytes) {
        SrmReceive(gPortID, &buffer[index], 1, 0, &error);
        if (error) {
            SrmReceiveFlush(gPortID, 1);
            index = 0;
            return;
        }
        switch (buffer[index]) {
            case chrCarriageReturn:
                // Treat a carriage return as the end of an
                // incoming message, since some terminals may
                // send CR instead of linefeed. Convert the CR
                // to a LF so the message is displayed
                // correctly in the incoming text field.
                buffer[index] = chrLineFeed;
```

```
                // Fall through...

            case chrLineFeed:
                // Treat a linefeed as the end of an incoming
                // message. Leave the linefeed intact in the
                // incoming data to properly format the
                // incoming text field, tack a terminating null
                // onto the string in the buffer, and then
                // display the message in the incoming field.
                buffer[index + 1] = chrNull;
                MainFormDisplayMessage(buffer);
                index = 0;
                break;

            default:
                index++;
                break;
        }
        bytes--;
    }
}
```

The **ReadSerial** function uses a two-step process to retrieve data from the incoming serial queue. First, it checks to see if there is any data in the queue with **SrmReceiveCheck**. Second, if there is data in the queue, **ReadSerial** receives it with **SrmReceive**.

The **SrmReceiveCheck** function retrieves the number of bytes sitting in the serial port's receive queue. This is the prototype for **SrmReceiveCheck**:

```
Err SrmReceiveCheck(UInt16 portId, UInt32* numBytesP)
```

As usual for new serial manager functions, the **SrmReceiveCheck** routine's first parameter is the port ID of the serial connection. The second parameter is a pointer to a `UInt32` variable that receives the number of bytes waiting in the queue. If **SrmReceiveCheck** succeeds, it returns a `0` value, indicating no error.

If **SrmReceiveCheck** reports that there is data to retrieve, **ReadSerial** then calls **SrmReceive** from within a `while` loop to retrieve the data one byte at a time, checking each incoming character to see if it is a linefeed or a carriage return, which Serial Chat uses to indicate the end of a message. If one of these end-of-message characters is encountered, **ReadSerial** calls **MainFormDisplayMessage** to display the message in the **Incoming** field.

The **SrmReceive** function has the following prototype:

```
UInt32 SrmReceive (UInt16 portId, void *rcvBufP, UInt32 count,
                   Int32 timeout, Err* errP)
```

The **SrmReceive** routine's second parameter is a pointer to a buffer to receive data from the serial queue, and the third parameter specifies the number of bytes to retrieve. The `timeout` parameter is the number of system ticks that the new serial manager waits to receive the requested block of data; if the timeout period is reached before the requested number of bytes has been retrieved, **SrmReceive** returns the error code `serErrTimeOut` via the `errP` parameter. Regardless of any errors encountered, **SrmReceive** reports the actual number of bytes received in its return value.

If a line error occurs while receiving data, **SrmReceive** returns the error code `serErrLineErr` in its `errP` parameter. When your application receives a `serErrLineErr` result from a **SrmReceive** call, you should clear the error using the **SrmClearErr** function, which takes the port ID of the connection as its only parameter. Alternatively, you can simply flush the receive queue with **SrmReceiveFlush** to ensure that no garbage data remains in the queue for the next read; the **SrmReceiveFlush** function also calls **SrmClearErr** internally to clear any line errors from the port.

If your application needs to receive records of a specific size, and it cannot handle partial records, you may want to consider using **SrmReceiveWait**. The **SrmReceiveWait** function waits a specified amount of time for a certain amount of data to enter the receive queue, and then returns. The prototype for **SrmReceiveWait** looks like this:

```
Err SrmReceiveWait(UInt16 portId, UInt32 bytes, Int32 timeout)
```

The `bytes` parameter specifies the number of bytes to wait for before returning, and the `timeout` value is the length of time in system ticks that **SrmReceiveWait** will stall while waiting to accumulate the required number of bytes. The **SrmReceiveWait** routine returns the error code `srmErrTimeOut` if it cannot retrieve all the bytes requested before the `timeout` period is up.

The SrmReceiveWait function puts the system into doze mode while it waits for data. Using SrmReceiveWait is a battery-friendly way to await large blocks of incoming data.

Retrieving serial port information

Should you wish to see what line errors were encountered when **SrmReceive**, **SrmReceiveCheck**, or **SrmReceiveWait** returns a `serErrLineErr` error code, you can use **SrmGetStatus** to retrieve them. The **SrmGetStatus** function has the following prototype:

```
Err SrmGetStatus (UInt16 portId, UInt32* statusFieldP,
                  UInt16* lineErrsP)
```

The `lineErrsP` parameter should specify a pointer to a variable that will hold the line error status. The line error value is a bit field that may contain a number of constant values, defined in the Palm OS header file `SerialMgr.h`. Table 15-1 shows the constants, their values, and what error each represents.

Table 15-1
Line Error Constants

Constant	*Value*	*Description*
serLineErrorParity	0x0001	Parity error
serLineErrorHWOverrun	0x0002	Hardware overrun
serLineErrorFraming	0x0004	Framing error
serLineErrorBreak	0x0008	Break signal detected
serLineErrorHShake	0x0010	Line handshake error
serLineErrorSWOverrun	0x0020	Software overrun
serLineErrorCarrierLost	0x0040	Carrier Detect (CD) signal dropped

The variable pointed to by the `statusFieldP` parameter receives a bit field containing hardware status information for the port. The `SerialMgr.h` header also defines constants for use with this status field, as described in Table 15-2.

Table 15-2
Serial Port Status Constants

Constant	*Value*	*Description*
srmStatusCtsOn	0x00000001	CTS line is active
srmStatusRtsOn	0x00000002	RTS line is active
srmStatusDsrOn	0x00000004	DSR line is active
srmStatusBreakSigOn	0x00000008	Break signal is active

Two more functions, **SrmGetDeviceCount** and **SrmGetDeviceInfo**, are useful if you want to take a peek at what serial devices are available on a given handheld and what capabilities those devices have. **SrmGetDeviceCount** requires a pointer to a `UInt16` value that will receive the total number of serial devices, both physical and virtual, present on the handheld:

```
Err SrmGetDeviceCount (UInt16* numOfDevicesP)
```

The **SrmGetDeviceInfo** function provides a description of a given serial port. It has the following prototype:

```
Err SrmGetDeviceInfo (UInt32 deviceID,
                      DeviceInfoType* deviceInfoP)
```

You have a number of options for specifying the `deviceID` parameter, which identifies which serial port you are interested in retrieving information from. The `deviceID` parameter may be a valid port ID returned from **SrmOpen** or **SrmOpenBackground**, the creator ID of a specific device, or a zero-based index. The index number is particularly useful when paired with the **SrmGetDeviceCount** function, as it allows you to enumerate the existing serial devices and retrieve information for all of them. The following code walks through the serial ports on a device and retrieves information about each port:

```
UInt16  index;
UInt32  i;
DeviceInfoType  deviceInfo;

SrmGetDeviceCount(&index);
for (i = 0; i < index; i++) {
    SrmGetDeviceInfo(i, &deviceInfo);
    // Do something with the information here.
}
```

The `DeviceInfoType` that **SrmOpenBackground** returns via its `deviceInfoP` parameter is defined in `SerialMgr.h` as follows:

```
typedef struct DeviceInfoType {
    UInt32 serDevCreator;        // Four Character creator type
                                 //  for serial driver ('sdrv')
    UInt32 serDevFtrInfo;        // Flags defining features of
                                 //  this serial hardware
    UInt32 serDevMaxBaudRate;    // Maximum baud rate for this
                                 //  device
    UInt32 serDevHandshakeBaud;  // HW Handshaking is
                                 //  recommended for baud rates
                                 //  over this value
    Char *serDevPortInfoStr;     // Description of serial HW
                                 //  device or virtual device
    UInt8 reserved[8];           // Reserved
} DeviceInfoType;
```

You can retrieve values from the `serDevFtrInfo` bit field by using the constants described in Table 15-3.

Table 15-3
Serial Capabilities Constants

Constant	*Value*	*Description*
serDevCradlePort	0x00000001	Serial hardware controls RS-232 serial from cradle connector of Palm OS device
serDevRS-232Serial	0x00000002	Serial hardware has RS-232 line drivers
serDevIRDACapable	0x00000004	Serial hardware has IR line drivers and generates IrDA mode serial signals
serDevModemPort	0x00000008	Serial hardware drives modem connection
serDevCncMgrVisible	0x00000010	Serial device port name string should be displayed in the Connection panel

Changing the size of the receive buffer

The default receive buffer provided by the new serial manager is 512 bytes long. If you need a larger buffer, you can provide your own and install it with **SrmSetReceiveBuffer**. The **SrmSetReceiveBuffer** function has the following prototype:

```
Err SrmSetReceiveBuffer (UInt16 portId, void *bufP,
    UInt16 bufSize)
```

Like most other new serial manager functions, the first argument to **SrmSetReceiveBuffer** must be the port ID of the serial connection. The `bufP` parameter is a void pointer to the buffer itself, and `bufSize` is the size of that buffer in bytes.

Call **SrmSetReceiveBuffer** after a successful call to **SrmOpen** to install the new buffer:

```
#define bufferSize 2048

Err     error;
UInt16  portID;
Char    buffer[bufferSize];

error = SrmOpen(serPortCradlePort, 9600, &portID);
if (! error)
    SrmSetReceiveBuffer(portID, buffer, bufferSize);
```

When your application is done using the serial port, it must reinstall the default buffer before closing the port. To accomplish this task, call **SrmSetReceiveBuffer** again, specifying `NULL` for both the `bufP` and `bufSize` parameters:

```
SrmSetReceiveBuffer(portID, NULL, NULL);
```

Changing serial port settings

Depending on what sort of device you are connecting a Palm OS handheld to, you may need to alter various communications settings, such as baud rate and parity, to be able to communicate with the other device. The **SrmControl** routine is the function to use when retrieving or setting communications settings for a serial port, and it has the following prototype:

```
Err SrmControl (UInt16 portId, UInt16 op, void *valueP,
                UInt16 *valueLenP)
```

The usual port ID heads the list of parameters for **SrmControl**. The next parameter, `op`, is a control code that specifies what action you want **SrmControl** to perform. This control code turns what looks like a simple four-parameter function into a tool of Swiss-Army-knife capabilities, both complex and versatile. Values for the `op` parameter should be from the enumerated type `SrmCtlEnum`, which is defined in `SerialMgr.h`. Depending on the value of the control code, the `valueP` parameter may specify either the address of a value to pass to the function, or the address of a variable that will receive data from **SrmControl**. If the requested control code uses the `valueP` parameter, then the `valueLenP` parameter specifies the address of a variable that contains the length of the data in `valueP`. Not all control codes use the `valueP` and `valueLenP` parameters; for those that do not require a value, pass `NULL` for `valueP` and `valueLenP`.

The possible control codes in `SrmCtlEnum`, and how they should be used, are described in Table 15-4.

Table 15-4
SrmControl Control Codes

Control code	*Description*
`srmCtlSetBaudRate`	Sets the baud rate for the connection. `valueP` should point to an `Int32` value that specifies the baud rate, and `valueLenP` should point to `sizeof(Int32)`.
`srmCtlGetBaudRate`	Retrieves the current baud rate of the connection. `valueP` should point to an `Int32` value that will receive the baud rate, and `valueLenP` should point to `sizeof(Int32)`.

Continued

Table 15-4 *(continued)*

Control code	*Description*
`srmCtlSetFlags`	Sets flags for the serial hardware. These flags control things like parity, number of stop bits, and bits per character. `valueP` should point to a `UInt32` bit field containing the flags to set, and `valueLenP` should point to `sizeof(UInt32)`.
`srmCtlGetFlags`	Retrieves flags for the serial hardware. `valueP` should point to a `UInt32` variable to receive the flags, and `valueLenP` should point to `sizeof(UInt32)`.
`srmCtlSetCtsTimeout`	Sets the length of the CTS timeout. `valueP` should point to an `Int32` value containing the timeout value, and `valueLenP` should point to `sizeof(Int32)`.
`srmCtlGetCtsTimeout`	Retrieves the length of the CTS timeout. `valueP` should point to an `Int32` variable to receive the timeout value, and `valueLenP` should point to `sizeof(Int32)`.
`srmCtlStartBreak`	Turns on the RS-232 break signal. Make sure to leave the break signal on long enough to generate a valid break on whatever device the handheld is connected to.
`srmCtlStopBreak`	Turns off the RS-232 break signal.
`srmCtlStartLocalLoopback`	Starts local loopback test.
`srmCtlStopLocalLoopback`	Stops local loopback test.
`srmCtlIrDAEnable`	Enables IrDA connection on the serial port.
`srmCtlIrDADisable`	Disables IrDA connection on the serial port.
`srmCtlRxEnable`	Enables receiver for IrDA communications.
`srmCtlRxDisable`	Disables receiver for IrDA communications.
`srmCtlUserDef`	Passes the `valueP` and `valueLenP` pointers to the SdrvControl function for a serial driver, or the VdrvControl function for a virtual driver. This control code is for use by serial driver developers, who may need to send or receive custom control information that the regular serial manager interface cannot handle.

Control code	*Description*
`srmCtlGetOptimalTransmitSize`	Asks the port for the most efficient buffer size for transmitting data packets. If the serial or virtual driver does not support this control code, SrmControl will return the error code `serErrNotSupported`, in which case no buffering should be done. If the port wants some kind of buffering, but it is not choosy about the buffer size, this control code sets `valueP` to point to `0`. Otherwise, `valueP` will point to a number that specifies the most efficient block size, in bytes, for transmitting data through this port. `valueLenP` points to `sizeof(Int32)`.

The flags for the `srmCtlSetFlags` and `srmCtlGetFlags` control codes may be accessed by means of a number of constant values, defined in `SerialMgr.h`. Table 15-5 describes what each flag represents.

Table 15-5
Serial Settings Constants

Constant	*Value*	*Description*
`srmSettingsFlagStopBitsM`	`0x00000001`	Mask for stop bits field.
`srmSettingsFlagStopBits1`	`0x00000000`	One stop bit.
`srmSettingsFlagStopBits2`	`0x00000001`	Two stop bits.
`srmSettingsFlagParityOnM`	`0x00000002`	Mask for parity on.
`srmSettingsFlagParityEvenM`	`0x00000004`	Mask for parity even.
`srmSettingsFlagXonXoffM`	`0x00000008`	Mask for Xon/Xoff flow control; not implemented as of Palm OS 3.5.
`srmSettingsFlagRTSAutoM`	`0x00000010`	Mask for RTS receive flow control.
`srmSettingsFlagCTSAutoM`	`0x00000020`	Mask for CTS transmit flow control.
`srmSettingsFlagBitsPerCharM`	`0x000000C0`	Mask for bits per character field.
`srmSettingsFlagBitsPerChar5`	`0x00000000`	Five bits per character.
`srmSettingsFlagBitsPerChar6`	`0x00000040`	Six bits per character.

Continued

Table 15-5 *(continued)*

Constant	*Value*	*Description*
srmSettingsFlagBitsPerChar7	0x00000080	Seven bits per character.
srmSettingsFlagBitsPerChar8	0x000000C0	Eight bits per character.
srmSettingsFlagFlowControl	0x00000100	Enables software overrun protection. If this flag and srmSettingsFlagRTSAutoM are both set, the new serial manager asserts the RTS signal when the receive buffer is full to prevent the transmitting device from overrunning the buffer. When the application receives data from the buffer, the system turns off the RTS assertion to allow more incoming data into the buffer. Using this feature prevents software overrun line errors, but it may cause CTS timeouts on the connected device if the RTS line is asserted longer than the current CTS timeout value.

As of this writing, the default settings for the serial port are eight bits per character, one stop bit, and no parity, with RTS flow control engaged and a CTS timeout value of five seconds. The constant `srmDefaultCTSTimeout`, defined in `SerialMgr.h`, specifies the default CTS timeout length; all the other defaults are part of the `srmDefaultSettings` constant.

To demonstrate using **SrmControl**, here are the relevant parts of Serial Chat's **OpenSerial** function that were omitted earlier in this chapter:

```
Err     error;
UInt32  flags = 0;
UInt16  flagsSize = sizeof(flags);

flags = srmSettingsFlagBitsPerChar8 |
        srmSettingsFlagStopBits1 |
        srmSettingsFlagRTSAutoM;
error = SrmControl(gPortID, srmCtlSetFlags, &flags,
                   &flagsSize);
```

Serial Chat sets the port up with eight bits per character, no parity, one stop bit, and RTS hardware flow control. It could just as easily have used seven bits per character, even parity, and one stop bit, in which case the `flags` variable would have been set like this:

```
flags = srmSettingsFlagBitsPerChar7 |
        srmSettingsFlagParityOnM |
        srmSettingsFlagParityEvenM |
        srmSettingsFlagStopBits1 |
        srmSettingsFlagRTSAutoM;
```

If you are experimenting with the Serial Chat sample application by connecting it to a terminal program on the desktop, be sure to set the terminal program to use eight bits per character, no parity, and one stop bit to ensure that the data passed over the serial connection does not become garbled.

If you plan to connect at speeds faster than 19,200 bps, use hardware handshaking to make a successful connection:

```
flags = srmSettingsFlagRTSAutoM |
        srmSettingsFlagCTSAutoM;
```

Using the Old Serial Manager

If you need to write an application that can use the serial port on devices that do not support the new serial manager, you will have to use the old serial manager. Fortunately, the two serial managers are very similar; in fact, the names of functions in the old serial manager are almost identical to their counterparts in the new serial manager, beginning with a **Ser** prefix instead of **Srm**. Most of the information already presented about using the new serial manager is applicable to using the old serial manager, so this section concentrates on the differences between the two managers.

To use any of the old serial manager functions with the Palm OS 3.5 or later SDK headers, be sure to include the file `SerialMgrOld.h` in your project. Even better, it is probably easiest to build with the 3.1 headers to avoid duplication of the enumerated types and constants that are used by both the old and new serial managers.

The first major difference between the old and new serial managers is that the old serial manager does not use the port ID value required by most of the new serial manager functions. Instead, old serial manager functions require a reference to the serial manager library, which you must retrieve with the **SysLibFind** function, like this:

```
UInt16 serialRefNum;

Err error = SysLibFind("Serial Library", &serialRefNum);
```

You must retrieve the serial library reference even before you call **SerOpen** to open the connection, since **SerOpen** also requires the reference as an argument.

Opening and closing the serial port

The **SerOpen** function has the following prototype:

```
Err SerOpen (UInt16 refNum, UInt16 port, UInt32 baud)
```

The first parameter, `refNum`, is the serial library reference retrieved by **SysLibFind**. The old serial manager supports only the standard physical serial port; to indicate this port, pass `0` for the `port` parameter. Specify the baud rate for the connection using the `baud` parameter.

Just as with the new serial manager, if **SerOpen** returns an error code of `serErrAlreadyOpen`, another task is currently using the port, and unless you want to deal with the nightmare of sharing the serial port with another application, you should close the connection with the **SerClose** function.

After opening the port, you may wish to flush the receive buffer with the **SerReceiveFlush** function, which operates just like its cousin **SrmReceiveFlush**.

When you are finished with the serial port, close it with the **SerClose** function. Optionally, you may wish to first drain the transmit queue with **SerSendWait**, which has an additional `timeout` parameter not present in **SrmSendWait**:

```
Err SerSendWait (UInt16 refNum, Int32 timeout)
```

The `timeout` parameter was never really implemented before the new serial manager made its debut, so you should pass the value `-1` for `timeout`.

Sending data

The **SerSend** function operates in the same fashion as **SrmSend**. If you require backward compatibility with Palm OS 1.0, use **SerSend10** instead, which has the following prototype:

```
Err SerSend10 (UInt16 refNum, void *bufP, UInt32 size)
```

The **SerSend10** function works in much the same fashion as **SerSend** and **SrmSend**; it just returns less useful information than later versions of the function.

Receiving data

You can use **SerReceive** in the same way that you would use **SrmReceive**; they have the same parameters. An older version of the function, **SerReceive10**, exists to support Palm OS 1.0:

```
Err SerReceive10 (UInt16 refNum, void *bufP, UInt32 bytes,
                  Int32 timeout)
```

Just like **SerSend10**, **SerReceive10** is somewhat less useful than its newer counterpart, as it returns less information about the transfer.

Getting serial port information

The old serial manager does not have functions equivalent to **SrmGetDeviceCount** and **SrmGetDeviceInfo**. If you want to check on the status of the connection after receiving a `serErrLineErr` error code from **SerReceive**, **SerReceiveCheck**, or **SerReceiveWait**, call **SerGetStatus**. The **SerGetStatus** function looks and operates differently from **SrmGetStatus**:

```
UInt16 SerGetStatus (UInt16 refNum, Boolean *ctsOnP,
                     Boolean *dsrOnP)
```

The values returned to the `ctsOnP` and `dsrOnP` parameters actually have no meaning; they were never implemented before the new serial manager was created. The return value from **SerGetStatus** is equivalent to the bit field returned in the `lineErrsP` parameter of **SrmGetStatus**; see the discussion of **SrmGetStatus** in the "Retrieving serial port information" section earlier in this chapter for some useful constants to use when reading this bit field.

Changing the size of the receive buffer

Just like the new serial manager's **SrmSetReceiveBuffer** function, **SerSetReceive Buffer** allows you to install your own buffer in place of the default 512-byte buffer. The same warnings as for **SrmSetReceiveBuffer** apply; be sure to reinstall the default buffer before closing the port.

Altering serial port settings

In the new serial manager, **SrmControl** is responsible for a great number of tasks. The old serial manager divides these duties among three functions: **SerControl**, **SerGetSettings**, and **SerSetSettings**. The **SerControl** function handles everything except for basic communications properties like baud rate and parity. The prototype for **SerControl** looks like this:

```
Err SerControl (UInt16 refNum, UInt16 op, void *valueP,
                UInt16 *valueLenP)
```

There are fewer control codes available for the `op` parameter of **SerControl** than for the **SrmControl** function, all of which are part of the `SerCtlEnum` enumerated type defined in `SerialMgrOld.h`. Most of the control codes are similar to those required by **SrmControl**; notable omissions are codes for setting and retrieving baud rate, retrieving optimal block size, and passing data to a custom serial driver.

Use **SerGetSettings** to retrieve communications parameters for the serial port and **SerSetSettings** to change the same parameters. These functions have the following prototypes:

```
Err SerGetSettings (UInt16 refNum, SerSettingsPtr settingsP)
Err SerSetSettings (UInt16 refNum, SerSettingsPtr settingsP)
```

The `SerSettingsType` structure is required by both of these functions; this structure is defined as follows in `SerialMgrOld.h`:

```
typedef struct SerSettingsType {
    UInt32  baudRate;    // Baud rate
    UInt32  flags;       // Miscellaneous settings
    Int32   ctsTimeout;  // Max # of ticks to wait for CTS to
                         //  become asserted before
                         //  transmitting; used only when
                         //  configured with
                         //  serSettingsFlagCTSAutoM.
} SerSettingsType;
```

The `flags` parameter is a bit field that uses constants similar to those used by the flags in the `srmCtlGetFlags` and `srmCtlSetFlags` control codes of the **SrmControl** function; just use the same constants, with a `ser` prefix instead of `srm`. Note that there is no equivalent in the old serial manager to the special `srmSettingsFlagFlowControl` mode for preventing software buffer overrun.

Summary

In this chapter, you learned how to use the Palm OS serial manager to communicate with other devices via the handheld's serial port. After reading this chapter, you should understand the following:

- The serial communications stack in the Palm OS is based on the byte-level control of the serial manager; the following protocols are built on top of the serial manager: the modem manager, Serial Link Protocol (SLP), Packet Assembly/Disassembly Protocol (PADP), Desktop Link Protocol (DLP), and Connection Management Protocol (CMP).
- The new serial manager is the preferred interface to basic serial input and output, but there is still backward compatibility with the old serial manager to allow serial communications programming for older devices.
- It is very important to close the serial port when your application does not actually need it to be open, since the serial port drains the batteries rapidly when open.

- New serial manager functions require a port ID, which you retrieve by passing a logical port number or a serial device creator ID to **SrmOpen** or **SrmOpenBackground**; old serial manager functions require a reference to the serial library, which you retrieve with the **SysLibFind** function.
- Unless you want to deal with the nightmare of sharing a connection between two applications, call **SrmClose** and don't use the serial port if **SrmOpen** or **SrmOpenBackground** returns the `serErrAlreadyOpen` error code.
- When receiving incoming data, take care not to completely shut out user input; the simplest way to achieve this is by passing a timeout parameter to the **EvtGetEvent** call in your application's event loop.
- You can change the size of the receive buffer with **SrmSetReceiveBuffer**, but you must make sure to reinstall the default buffer before closing the port.

✦ ✦ ✦

CHAPTER 16

Creating Web Clipping Applications

In This Chapter

Understanding Web clipping

Building Palm Query Applications (PQAs)

Building Web clippings

Testing Web clipping applications

Web clipping on the Palm VII (or the Palm V/Vx with an OmniSky wireless modem) offers a powerful way to connect the personal organizer to timely data on the Web or in the enterprise. Though by no means the first company to offer some kind of Web connectivity for a handheld computer, Palm Computing has approached the problem from a different angle. Before learning the nuts and bolts of Web clipping, it is important to understand Palm's philosophy regarding the Web clipping process.

After presenting the theory behind Web clipping, this chapter delves into the specifics of creating Palm Query Applications (PQAs), the client-side interfaces for Web clipping applications. Later in the chapter, you will find information about how to format Web clippings, the actual HTML pages returned to the handheld from the Web as the user interacts with a PQA.

Building PQAs and Web clippings requires knowledge of how to make Web pages using HTML (HyperText Markup Language). No programming is involved in creating the HTML pages themselves, although you should also be familiar with some form of CGI programming if you wish to dynamically generate Web clippings on the server end.

Understanding Web Clipping

Most Web content is predicated on the assumption that the average Web user is seated in front of a desktop computer, connected to the Web via a modem or even faster connection. Web sites that contain lots of images and tricky page layout

are perfectly acceptable for someone using a machine with a fast processor, plentiful network bandwidth, and a lot of screen real estate. However, trying to display this kind of content on a small personal organizer, connected to the Web via a bandwidth-limited wireless modem, is a ludicrous proposition at best.

As with other Palm OS applications, limited processor power and screen size are major constraints on the design of a PQA. Big, flashy pages simply won't fit on the small 160 × 160-pixel screen, even if the organizer's processor could render them in a reasonable amount of time. In addition, a PQA has another limitation: expensive wireless bandwidth. Unlike modem connections, which tend to be billed at a flat rate, wireless connections are billed by the number of bytes sent across the network. Money matters aside, another problem is that lengthy connections also drain the handheld's battery at an alarming rate.

To deal effectively with the limitations of getting Web content to a handheld device via a wireless network, Palm Computing uses an approach called *Web clipping*. Much like clipping an article from a newspaper, Web clipping enables you to extract only the information you need, and nothing else. Web clipping embodies two important concepts:

- **Query and response.** Unlike Web browsing, where the focus is on hyperlinks between documents, Web clipping focuses on a simple query that generates a response (called the *clipping*). The query is defined by an HTML form, and the clipping is usually generated dynamically with a CGI script. Instead of following links through various documents to retrieve data, you assemble a single query and receive a single answer containing all the data you are interested in. Query and response design results in getting the data you want, and only the data you want, much more quickly than if you had to browse for it. Limiting the amount of data actually sent over the air, in both directions, keeps wireless transmission costs down.
- **Partitioning.** A Web clipping solution is partitioned between the client and the host. The query part of the equation, a *Palm Query Application*, is stored on the client end (the Palm OS handheld itself), separate from the host. Unlike with a normal Web application, in which a form must first be downloaded before the user enters query parameters, filling out a Web clipping query requires no communication over the network because a PQA contains the entire HTML form required for query entry on the client device. Keeping the query form local results in instant response time from the handheld while the user builds the query and fewer bytes of transfer across the expensive wireless connection. Data returned from the query, called a *Web clipping* or just a *clipping*, can also be very small, since it contains only the data specifically requested by the user. Furthermore, any processor-intensive computation required to generate the clipping is restricted to the host server, which is much better suited to complex number-crunching than the handheld.

A key element of Palm Computing's solution to the problem of wireless network bandwidth is the collection of Palm Web clipping proxy servers run as part of the Palm.Net service. PQAs connect to a proxy server, run by Palm, that works as a translator between the wireless network and the Internet. The proxy receives a query request from the handheld via User Datagram Protocol (UDP); use of the simple UDP protocol on the wireless network means that only two small packets are exchanged between the handheld and the proxy, one for the query and one for the clipping. Next, the proxy communicates with HTML servers via standard Internet protocols (TCP, HTTP, and SSL) to retrieve the information specified in the query. Then the proxy compresses the data and sends it back to the handheld over the wireless network.

Understanding Web Clipping Security

Web clipping has been designed to be secure enough for applications such as shopping and online banking. Secure connections between the handheld and the wireless network use Certicom's elliptic curve encryption, a small and efficient but highly secure public key cryptography system. Complete details of how elliptic curve cryptography works are available on Certicom's Web site at `http://www.certicom.com`.

Messages sent to the wireless network are also protected with a message integrity check (MIC), which can detect both tampering and errors in transmission, ensuring that the data you send in a secure message from the handheld remains unmolested on its way to the wireless base station.

Once a Palm proxy server has received a secure query from the wireless network, the connection between the proxy server and the query's destination server on the Internet is further protected by the use of Secure Sockets Layer (SSL) encryption and authentication. SSL is in use throughout the Web, providing strong security for many e-commerce and financial Web sites. To perform client authentication, a Web clipping application should ask the user for a user name and password when a query is submitted. Each Web clipping–enabled handheld device also has a unique identifier attached to its ROM hardware, which may also be sent with the query for added security. See the description of the special `%deviceid` string in the "Constructing Query Forms" section later in this chapter for more details.

Designing PQAs and Web Clippings

Palm Query Applications and Web clippings are much simpler to create than full-fledged Palm OS applications. Instead of being programmed in C or another complex development language, PQAs and clippings begin their lives as HTML documents. A subset of HTML 3.2, with some Palm-specific additions and modifications, serves as the language for defining both the client and server ends of a Web clipping application.

Even though you use HTML to create a Palm Query Application, designing a PQA is very different from designing normal Web-based content. Keep the following points in mind when designing a PQA.

- **Remote hyperlinks are expensive.** Traditional Web pages present a certain subject, and then provide hyperlinks that lead (at least in theory) to more detailed information about the subject. This works fine with enough bandwidth to serve up a new page each time the user follows a link, but over a wireless network, it becomes prohibitively expensive. Where possible, provide static information, such as help documentation for the PQA itself, as part of the client. Use the local query form to allow the user to explore possibilities in the application, and present remote hyperlinks only when there is information that cannot be stored in the client itself. For example, if your PQA displays news headlines, an appropriate use of remote hyperlinks would be to allow the user to retrieve the full text of a news story.
- **The client is static.** Many technologies exist to allow Web pages to be dynamic, changing their contents rapidly in response to user input. In the Web clipping model, the client end does not change. The host side of a PQA may be modified at any time, but the PQA residing on the handheld will remain the same. Design the client carefully to allow for expansion or changes on the host side. It is possible for the user to update a PQA on the handheld, but this requires downloading the updated PQA with the desktop machine and installing it through a HotSync operation.
- **Bandwidth is expensive.** Bandwidth on a desktop machine, hooked up to the Internet via a modem or more direct means, is relatively plentiful, and usually billed at a flat rate. Every exchange of data over a wireless network, on the other hand, requires expenditure of money, time, and battery life. The pricing of the Palm.Net service is based on a typical transaction where the query should send 40 bytes across the network, and the response should contain 360 bytes, figures that reflect compressed data in both the query and the response. Avoid sending unnecessary HTML tags, images, or information to the user. Images in particular require a great deal of bandwidth, and should not be sent across the wireless network. Instead, consider embedding images that you will use repeatedly in the client PQA itself, and then calling them from the clipped pages.
- **Performance is critical.** A handheld user expects more speed of an application than a user sitting at a desktop machine. People using desktop computers are usually not in the middle of a conference call with important clients, dashing through an airport to catch their next flight, or standing on a street corner wondering where to find the nearest cash machine. A desktop user is more likely to be patient with a slow application. Handheld users expect that the device will present them with the data they need instantly and with as little input on their part as possible. A PQA that requires the transfer of too much data is not only expensive to use but will seem sluggish to the typical handheld user. For example, sending an in-depth news story to the handheld would require the user to wait a considerable period of time, but a shorter synopsis of the news story would download much more quickly.

- **Screen size is limited.** Although HTML is designed to format content independent of the device that displays it, most Web designers assume that their pages will be viewed on a desktop monitor. Even the smallest monitor at its lowest resolution can display far more information than the tiny screen on a handheld. The title bar and scroll bar of the Clipper application (the Palm OS browser that displays PQAs) reduce the Palm's already limited screen real estate to a mere 153 pixels wide by 144 pixels high. Clipper's display can scroll vertically, but not horizontally, so avoid lines of text or images that will exceed the 153-pixel width of the screen.

Building Palm Query Applications

Creating the client side of a Palm wireless application is relatively simple and requires no knowledge of programming. You need only two things:

- A standard text editor, or a non-WYSIWYG HTML editor
- The Query Application Builder

Many WYSIWYG (What You See Is What You Get) HTML editors insert nonstandard or unnecessary HTML elements into the pages they produce. Both PQAs and Web clippings use a particular subset of HTML — and weird tags, or tags not supported by Palm OS Web clipping, can cause Clipper to render a page improperly on the handheld. Even if the page renders properly, extra HTML elements beyond the bare minimum required to format a page are a waste of wireless bandwidth; strive to use as little HTML tagging as possible when making a PQA, particularly when designing Web clippings, which must be transmitted in their entirety over the wireless connection. It is safest to use a simple text editor for creating PQAs and Web clippings.

The Query Application Builder is a program for both Windows and Mac OS that converts standard HTML pages into a `.pqa` file on your PC, which you may then install to the handheld like any other Palm OS application or database. Palm Computing provides the Query Application Builder free of charge on its Web site at `http://www.palm.com/devzone/webclipping/`.

The Query Application Builder is also available on the CD-ROM attached to this book.

Think of the Query Application Builder as a compiler that takes your HTML source and compiles it into a form that Clipper, the Palm OS browser application on the handheld, can display. The Query Application Builder also compresses the PQA between 5 and 60 percent, depending on the actual content of the HTML pages and images that make up the PQA.

Organizing HTML Files

A PQA may be composed of multiple pages, each of which is defined by a single HTML file with either an `.htm` or `.html` extension. The best way to organize the pages displayed in a PQA is to center the application around a home page, by convention called `index.html`. The home page should contain the primary query form for the application, or a list of local and online links for requesting more specific content than what the home page displays.

Try to make sure the home page provides quick access to the most-used features in the PQA. In that way, users need to launch only the PQA and immediately start querying for information, instead of having to tap through multiple pages to find the information they are looking for.

All of the pages that make up a PQA must fall under the same root directory, but pages and images may be contained in subdirectories if you wish to better organize development of a complex PQA. Keep in mind that from within a PQA, links to pages and images must be relative paths that include whatever subdirectories contain the element pointed to by the link. For example, the following link points to a page in a subdirectory of the PQA project's root directory:

```
<a href="subdirectory/page2.html">Page Two</a>
```

In particular, pay attention to relative paths when linking to a page in a higher directory level than the current page. For example, the following link is required to return to the `index.html` page in the root directory from the `page2.html` page in the last example:

```
<a href="../index.html">Home</a>
```

However, because of the fact that the Query Application Builder puts all the pages and images into a single Palm OS database, you should make sure that individual pages and images have unique names across the project, even if they are stored in separate directories. For example, if you already have an `index.html` file in the root directory of the PQA project, you cannot also have another file named `index.html` in any subdirectory. In fact, the file extension is not enough to differentiate files, so a page called `index.html` and an image called `index.gif` cannot be in the same PQA. Identically named files cause an error in the Query Application Builder, which also reports the line number of the offending link so you can easily track down any accidental duplicates.

The unique file names also play a part when a Web clipping links to the local PQA. Once the PQA has been compiled, any subdirectories used when you first built the application are "forgotten," so links in Web clippings to local pages on the handheld must treat the PQA as if all its files existed in a single flat directory. For example, the following Web clipping link goes to the same `page2.html` page mentioned earlier in this section:

```
<a href="file:localpqa.pqa/page2.html">Page Two</a>
```

See the "Linking to Other Pages and Applications" section below, and the "Building Web Clippings" section later in this chapter, for more information about adding hyperlinks to query applications and clippings.

Keeping file sizes small

Technically, the total size of a PQA is limited only by the amount of RAM storage on the handheld. An individual page that makes up a PQA can be no larger than 63KB in size, which is the largest size allowed for a database by the system. Of course, 63KB of text and small images is a ridiculous amount for a PQA; something this large would require users to scroll through many pages of information to find what they are looking for.

If at all possible, try to fit all the elements necessary to assemble a query onto a single page; scrolling through a large page to find the submit button at the bottom tends to irritate users.

A good guideline to follow is to make the entire compiled PQA, including all pages and images, no larger than about 15KB. Remember that the Query Application Builder performs a good deal of compression on the text that makes up a PQA, so even an application with many pages should be fairly easy to fit within this limit. When you have compiled a PQA, the Query Application Builder shows you exactly how much compression it was able to achieve.

Although the obvious solution to limiting PQA size is to use fewer images (which do not compress as well as text), you should also try to use as little HTML markup as possible. Every character in an HTML file, including tags, adds to a PQA's overall size.

Defining Header Tags

Just like a standard HTML document, a PQA should start with an `<html>` tag, followed by a `<head>` tag to define the header elements of the page. The following simple example is a complete one-page PQA that simply displays the text "Hello, world" on a mostly blank page. Figure 16-1 shows this page as it appears on the handheld.

```
<html>
<head>
<title>Hello PQA</title>
<meta name="palmcomputingplatform" content="true">
<meta name="palmlauncherrevision" content="1.0">
</head>
<body>
Hello, world.
</body>
</html>
```

The text contained in a page's `<title>` tag appears in the title bar of the Clipper application. This title string must be fairly small; when testing the PQA in POSE or

on an actual device, be sure to check the title to make sure it has not been truncated to fit within the title bar.

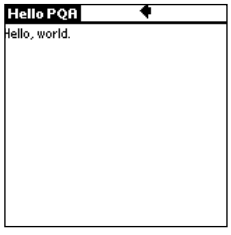


Figure 16-1: A very simple example PQA

Two special `<meta>` tags should be used in PQA pages. The `palmcomputingplatform` tag should appear in the `<head>` section of every page in a PQA. This tag tells the Query Application Builder that the page was written specifically for display on a Palm OS handheld. Without the `palmcomputingplatform` tag, images will not be included in the PQA.

The `palmlauncherrevision` tag allows you to specify a version number for the PQA. Just like the version number resource in a full-fledged Palm OS application, the version number of a PQA appears in the system launcher application's Info dialog box, allowing users to look up what version of the PQA is installed if you have released multiple versions of the same PQA. You should include the `palmlauncherrevision` tag only on the home page of your PQA.

Formatting Text

The Clipper application displays HTML pages using its own special typeface, called Palm TD. At its normal size, Palm TD looks just like the standard Palm OS font (specified by the constant `stdFont` when developing normal Palm OS applications). Unlike the standard Palm OS font, however, Palm TD has bold, italic, bold italic, and monospaced versions.

There are only a few sizes available for the Palm TD font, from 7 to 12, and only certain sizes are available for the `<small>`, `<big>`, or `<font size=n>` HTML tags, depending on what other HTML formatting has been applied to the text in question. For example, text contained in an `<h2>` heading tag is normally displayed in bold Palm TD 11, but with the `<big>` tag applied to it, it becomes bold Palm TD 12. Such interactions between formatting and size tags are not terribly intuitive, since there are so few options available for font size and style in Clipper. Use Table 16-1 to determine how Clipper will display text using specific combinations of formatting and sizing tags. The Smallest and Largest columns of the table display the smallest and largest sizes available when using the `<font size=n>` tag to set text size.

Table 16-1
Fonts Used to Display Specific HTML Elements

HTML Tag	*Normal*	*<small >*	*< big>*	*Smallest*	*Largest*
<address>	TD 9 B It	TD 8 B	TD 10 B	TD 7 B	TD 11 B
<b>	TD 8 B	TD 8 B	TD 10 B	TD 7 B	TD 11 B
<blockquote>	TD 9	TD 8 B	TD 9 B	TD 7 B	TD 11 B
<cite>	TD 9 It	TD 8 B	TD 9 It	TD 7 B	TD 10 B
<dfn>	TD 9 It	TD 8 B	TD 9 It	TD 7 B	TD 10 B
<dir>	TD 9	TD 8 B	TD 9 B	TD 7 B	TD 11 B
<dl>	TD 9	TD 8 B	TD 9 B	TD 7 B	TD 11 B
	TD 9 It	TD 8 B	TD 9 It	TD 7 B	TD 10 B
<h1>	TD 12 B	TD 11 B	TD 12 B	TD 11 B	TD 12 B
<h2>	TD 11 B	TD 10 B	TD 12 B	TD 9 B	TD 12 B
<h3>	TD 10 B	TD 9 B	TD 11 B	TD 8 B	TD 12 B
<h4>	TD 8 B	TD 8 B	TD 10 B	TD 7 B	TD 11 B
<h5>	TD 8 B	TD 8 B	TD 9 B	TD 7 B	TD 10 B
<h6>	TD 7 B	TD 7 B	TD 8 B	TD 7 B	TD 9 B
<i>	TD 9 It	TD 8 B	TD 9 It	TD 7 B	TD 10 B
<kbd>	TD Mono	TD Mono	TD Mono	TD Mono	TD Mono
<listing>	TD Mono	TD Mono	TD Mono	TD Mono	TD Mono
<menu>	TD 9	TD 8 B	TD 9 B	TD 7 B	TD 11 B
	TD 9	TD 8 B	TD 9 B	TD 7 B	TD 11 B
<p>	TD 9	TD 8 B	TD 9 B	TD 7 B	TD 11 B
<plaintext>	TD Mono	TD Mono	TD Mono	TD Mono	TD Mono
<pre>	TD Mono	TD Mono	TD Mono	TD Mono	TD Mono
<sample>	TD Mono	TD Mono	TD Mono	TD Mono	TD Mono
	TD 8 B	TD 8 B	TD 10 B	TD 7 B	TD 11 B
<table>	TD 9	TD 8 B	TD 9 B	TD 7 B	TD 11 B
<tt>	TD Mono	TD Mono	TD Mono	TD Mono	TD Mono
	TD 9	TD 8 B	TD 9 B	TD 7 B	TD 11 B

A complete list of HTML elements supported by Web clipping and explanations of how those elements differ between Clipper and a normal Web browser are available in *Web Clipping Developer's Guide*, included on this book's CD-ROM.

Linking to Other Pages and Applications

The standard HTML `<a>` tag defines links between pages within a local PQA, as well as links to online content. You can also use hyperlinks to launch other PQAs and applications on the handheld.

Marking a location in a document

Just as with standard HTML, you can mark a specific location in a page with the `<a name>` tag. Such a mark is called an *anchor*; the syntax required to define an anchor looks like this:

```
<a name="targetname">Some text string</a>
```

You can then link to that specific location in the page with the following link:

```
<a href="page.html#targetname">Link text</a>
```

This style of linking is mostly useful when a hyperlink needs to point to text on a large page that might otherwise require scrolling to reach the desired part of the page.

Linking to local pages

You can link to another page within the same PQA with the following syntax:

```
<a href="page.html">Link text</a>
```

If the local page you want to link to is in a subdirectory, use this syntax instead:

```
<a href="subdirectory/page.html">Link text</a>
```

Linking to remote pages

Remote links are very similar to local links. Instead of a relative path to a local page, though, you use a complete URL for the desired page:

```
<a href="http://www.companyname.com/page.html">Link text</a>
```

For a secure connection to an Internet server using SSL for security, use the `https` prefix in the URL:

```
<a href="https://www.companyname.com/login.html">Link text</a>
```

When you create a link to a remote page, Clipper automatically appends a distinctive over-the-air icon to the end of the link's text. This icon alerts the user that following the link will make a connection over the wireless network, incurring possible charges for the data transmitted. Similarly, links to secure servers are followed by a secure over-the-air icon, which lets the user know that the link may send even more data across the wireless connection, since the encrypted contents of a secure query require extra header information for message integrity checks and authentication. Figure 16-2 shows both of the over-the-air icons.

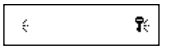


Figure 16-2: The over-the-air icon (left) and the secure over-the-air icon (right)

If you make images that serve as links to online content, incorporate the appropriate over-the-air icon as an element in the image to tell the user that the image is a link over the wireless connection. Clipper does not append the over-the-air icon to images. The two icons are part of the standard Palm OS font, and you can include them in regular applications by using the character constants `chrOta` and `chrOtaSecure`.

Making link buttons

The HTML subset supported by Clipper has a `button` attribute that you can add to standard hyperlinks:

```
<a href="page.html" button>Link text</a>
```

Using `button` surrounds the link text with a standard Palm OS button, as shown in Figure 16-3. Clipper will automatically add an appropriate over-the-air icon to the button's text for remote links.

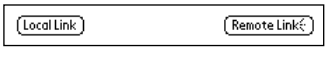


Figure 16-3: Link buttons for local (left) and remote (right) hyperlinks

Linking to another PQA

You can link to another PQA by using the `file:` prefix and the name of the PQA you wish to jump to:

```
<a href="file:other.pqa">Launch other PQA</a>
```

Be sure to include the `.pqa` extension after the name of the other PQA; without the extension, Clipper will return an error.

To link to a specific page within another PQA, use this syntax:

```
<a href="file:other.pqa/page.html">Link text</a>
```

Since you are linking to a compiled PQA, subdirectory names do not exist, so you should treat all the pages in the target PQA as belonging to the same flat directory structure.

Linking to other applications

Hyperlinks within a PQA can also launch other applications on the handheld. There are two ways to launch another application: `palm` and `palmcall`.

The `palm` syntax launches an application using the Palm OS **SysUIAppSwitch** routine, which causes Clipper to quit before launching the program. Here is the syntax for launching an application using the `palm` URL:

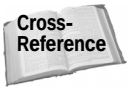
```
<a href="palm:cccc.tttt">Launch app</a>
```

In the above syntax, `cccc` represents the creator ID of the application to launch, and `tttt` is the database type of the application. Usually, you will use `appl` for `tttt` to specify an application database. For example, the following URL launches the built-in Memo Pad application:

```
<a href="palm:memo.appl">Launch Memo Pad</a>
```

The `palmcall` syntax uses **SysAppLaunch** to start another application, which leaves Clipper running in the background and launches the new application as a subroutine of Clipper. An application launched with `palmcall` does not have access to its global variables. This is what the `palmcall` syntax looks like in action:

```
<a href="palmcall:cccc.tttt">Sub-launch app</a>
```



See the "Launching Applications" section of Chapter 10, "Programming System Elements," for more details about SysUIAppSwitch and SysAppLaunch.

Most of the time, you will probably want to use the `palm` URL to give control to another application. The `palmcall` URL is primarily useful for passing data to another application, which the called program can then process or store as appropriate.

When using either the `palm` or `palmcall` URL, Clipper sends the launched application a `sysAppLaunchCmdURLParams` launch code. The parameters to this launch code include the complete URL as it was passed in the link. If you append a question mark (`?`) to the URL, any characters following the question mark may be used

by the called application for whatever purposes the program needs. The following example launches an application and passes it some parameters:

```
<a href="palmcall:cccc.appl?firstName=John&lastName=Doe">
Save name</a>
```

Parsing the URL for data after the question mark is the responsibility of the application handling the `sysAppLaunchCmdURLParams` launch code.

Adding mail links

You can use a standard `mailto` URL to allow the user to send e-mail using the iMessenger application, which is included on a Palm OS device with Clipper installed. The syntax looks like this:

```
<a href="mailto:info@companyname.com">Send mail</a>
```

It is also possible to define the subject and body of the mail using the following syntax:

```
<a href="mailto:info@companyname.com?subject=foo&body=bar">
Send mail</a>
```

When the user taps on a `mailto` link, the iMessenger application opens to display the new mail for editing. The iMessenger application does not automatically send the mail message, even if the subject and body are filled in using parameters, because the user should have ultimate control over when the handheld sends data over the wireless network.

Launching Clipper from Applications

Not only can you launch applications from within a PQA or Web clipping, you can also call PQAs from within another application. Before blindly trying to use wireless features from another application, though, you should check to make sure those features are present on the handheld. To find out if the system supports wireless Internet access, check for the existence of the Clipper and iMessenger applications. The Palm OS provides constants for the creator IDs of Clipper (`sysFileCClipper`) and iMessenger (`sysFileCMessenging`).

To check for the existence of Clipper, use the following code:

```
DmSearchStateType  searchState;
UInt     cardNo;
LocalID  dbID;
Err      error;
```

Continued

Continued

```
error = DmGetNextDatabaseByTypeCreator(true, &searchState,
    sysFileTApplication, sysFileCClipper, true, &cardNo, &dbID);
```

If DmGetNextDatabaseByTypeCreator returns an error, Clipper is not present on the handheld. Substitute `sysFileCMessenging` for `sysFileCClipper` in the code above to check for iMessenger.

Once you have verified that the system supports wireless net access, you can open a PQA in Clipper by calling SysUIAppSwitch to launch Clipper with a `sysAppLaunchCmdOpenDB` launch code. Pass the LocalID and card number of the PQA to display in Clipper as parameters to the launch code. The following function launches a PQA, given the PQA's name:

```
Err LaunchPQA (Char *pqaName)
{
    SysAppLaunchCmdOpenDBType  *params;
    DmSearchStateType  searchState;
    UInt16   cardNo;
    LocalID  dbID;
    Err      error = 0;


    cardNo = 0;
    dbID = DmFindDatabase(0, pqaName);
    if (dbID) {
        params = MemPtrNew(sizeof(SysAppLaunchCmdOpenDBType));
        if (! params)
            return sysErrNoFreeRAM;
        &params->cardNo = cardNo;
        &params->dbID = dbID;
        MemPtrSetOwner(params, 0);
        error = SysUIAppSwitch(cardNo, dbID,
                               sysAppLaunchCmdOpenDB, &params);
    }

    return error;
}
```

You can also pass an arbitrary URL to Clipper using the `sysAppLaunchCmdGoToURL` launch code, and Clipper will fire up a wireless connection and display that URL. Keep in mind that regular Web content that does not have a `palmcomputingplatform <meta>` tag will be truncated and have all its images removed by the Palm.Net proxy server. You should restrict using `sysAppLaunchCmdGoToURL` to calling Palm-friendly content.

The following function passes a URL, given as a string argument to the function, to Clipper:

```
Err LaunchURL (Char *url)
{
    DmSearchStateType  searchState;
    Err      error = 0;
    Char     *tempUrl;
    UInt     cardNo;
    LocalID  dbID;


    tempUrl = MemPtrNew(StrLen(url));
    if (!tempUrl)
        return sysErrNoFreeRAM;
    StrCopy(tempUrl, url);
    MemPtrSetOwner(tempUrl, 0);

    error = SysUIAppSwitch(cardNo, dbID, sysAppLaunchCmdGoToURL,
                           tempUrl);

    return error;
}
```

If you directly call up a link in Clipper using `sysAppLaunchCmdGoToURL`, be sure your application makes it perfectly clear that it is about to make a wireless connection. Because wireless airtime is expensive, the user should never be surprised by an unexpected over-the-air connection. One way to ensure that the user knows a wireless connection might be made is to incorporate the over-the-air icons in your own application. The Palm OS headers define the character constants `chrOta` and `chrOtaSecure` for regular and secure connection icons; both of these characters are part of the standard Palm OS font.

Constructing Query Forms

To collect data for a query from a user, use a standard HTML form, enclosed in the HTML `<form>` tag. All the standard HTML form input types are available, including text fields, text areas, pop-up menus, radio buttons, check boxes, hidden fields, and submit buttons. Also, Clipper understands a couple of Palm OS–specific tags for inserting date and time pickers into a PQA or Web clipping form.

There are two things you should keep in mind when creating a query form:

- **Minimize user input.** Make it easy for the user to enter data into the query form. For example, instead of requiring the user to enter text in a field, consider using a pop-up list instead, which allows the user to pick a value with a couple of stylus taps instead of slowly entering the data using Graffiti or the on-screen keyboard.

- **Minimize query data.** Keep the amount of data that must go over the air as small as possible. One way to do this is to use short, all-lowercase `name` parameters for form controls. Also, try to abbreviate the data itself if it comes from something other than a text field. For example, in a pop-up list of states, if the user selects California from the list, send the string `CA` instead of `California`. Every single byte transmitted counts.

You should also strive to make the interface of a query form as close to the interface of dialog boxes in other Palm OS applications as possible. The easiest way to do this is to use a table to format the input form, which allows you to use a right-justified label for each left-justified form input control. The following example produces the form displayed in Figure 16-4:

```
<form action="http://someURL" method="get">
<table>
  <tr>
    <td align=right><b>First Name:</b></td>
    <td><input type="text" name="fname" size="15"
         maxlength="30"></td>
  </tr>
  <tr>
    <td align=right><b>Last Name:</b></td>
    <td><input type="text" name="lname" size="15"
         maxlength="30"></td>
  </tr>
  <tr>
    <td align=right><b>Color:</b></td>
    <td><select name="color">
        <option selected value="0">-Select a color-
        <option value="r">red
        <option value="g">green
        <option value="b">blue
        </select></td>
  </tr>
</table>
<input type="submit" value="Submit Query">
</form>
```

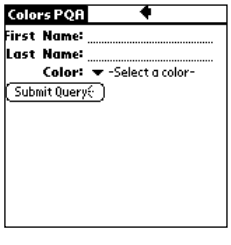


Figure 16-4: A sample query form that uses a table for formatting

HTML form controls become specific Palm OS form elements in Clipper. The following sections describe how the various HTML input tags affect the appearance and properties of user interface elements that appear in Clipper.

Text fields and text areas

For a single line of text input, use a text field input type:

```
<input type="text" name="name" size="15" maxlength="20">
```

The `size` attribute controls the width of the text field on the screen. You will have to experiment with the `size` attribute to find a width that works. A `size` of 25 will give you a line of text that fills the width of the Clipper browser.

Use the `maxlength` attribute to control the maximum number of characters that may be entered in the field.

If you want to allow entry of larger amounts of text in a query form, use the `<textarea>` tag:

```
<textarea rows="3" name="name" maxlength="100">This text
    appears in the text area</textarea>
```

The `<textarea>` tag creates a multiline text field, complete with scroll bar if the user enters more text than can be displayed at once in the field. Use the `rows` attribute to control how many visible rows the text field can display, and `maxlength` to set the maximum number of characters that the field will accept.

Pop-up menus

The `<select>` and `<option>` tags work together to create a pop-up list. For example, the following code creates a simple four-item pop-up list:

```
<select name="popup">
<option selected value="0">-Select a color-
<option value="r">red
<option value="g">green
<option value="b">blue
</select>
```

Whichever `<option>` contains the `selected` attribute is initially displayed in the pop-up trigger. Remember to keep the `value` attributes as small as possible, since they will be transmitted over the wireless connection. Also, try to limit the size of a pop-up list; scrolling through a massive list is time-consuming and tends to annoy users.

Check boxes and radio buttons

The `checkbox` type of `<input>` tag creates a check box:

```
<input type="checkbox" value="1" name="name">
```

Note that there is no label associated with the check box. If you want to label the check box (a good idea if you want the user to know what the check box is for), simply include the label as text before or after the `<input>` tag:

```
<input type="checkbox" value="1" name="name">Check here
```

Radio buttons have a similar format. All radio buttons that share a common `name` form an exclusive group; only one radio button in a group may be selected at a time, and that button's `value` attribute is sent when the user submits the form:

```
<input type="radio" name="color" value="r">Red
<input type="radio" name="color" value="g">Green
<input type="radio" name="color" value="b">Blue
```

Clipper implements each radio button as a Palm OS push button. Whatever text immediately follows the `<input>` tag appears within that particular push button. To make sure that the push buttons line up properly, keep the text within the radio button tags short, and be sure not to include any `<br>` or other formatting tags between buttons that make up a group.

Buttons

The `<input type="button">` variety of form button is not supported by Web clipping, since there is no scripting possible in a PQA or clipping page, and the primary function of a `button` input is to launch script. Clipper does support the `submit` and `reset` styles of button, however.

A `submit` button launches the action defined in the query's `<form>` tag:

```
<input type="submit" value="Submit Query">
```

A `reset` button resets the form's values to the state they were in when the page was first displayed:

```
<input type="reset" value="Reset">
```

The `value` attribute for `submit` and `reset` buttons defines the text that appears within the button. Clipper automatically appends the over-the-air icon to the text in a `submit` button.

Date and time pickers

Clipper understands two special form input types that take advantage of the built-in Palm OS date and time picker dialog boxes: `datepicker` and `timepicker`. Tapping the selector trigger defined by these input types launches the appropriate picker dialog box for selection of a date or a time.

To insert a selector trigger that runs the date picker, use the following syntax:

```
<input type="datepicker" name="name" value="YYYY-MM-DD">
```

The date picker displays the date using whatever system date preferences are currently in use on the handheld, even though the format of the date returned by the date picker and the format used to set the `value` attribute is `YYYY-MM-DD`. If you omit the `value` attribute, the date picker initially displays the current date according to the handheld's internal clock.

To insert a selector trigger that runs the time picker, use the following syntax:

```
<input type="timepicker" name="name" value="HH:MM">
```

Like the date picker, the time picker displays the time in whatever user-defined time format is in use on the handheld. The format used for setting and returning the time picker's value is the 24-hour `HH:MM` clock format. If you omit the `value` attribute, the time picker initially displays the current time according to the handheld's internal clock.

Special Palm OS variables

Two special variables may be included in any data that you submit from a PQA query: `%zipcode` and `%deviceid`. These variables are usually included as `hidden` inputs in the form, though they may also be tacked onto the end of the form's `action` URL.

The Palm.Net proxy servers fill in the `%zipcode` variable with the ZIP code of the nearest wireless base station, usually within 5 to 10 miles of the handheld. Use the following syntax to send the ZIP code in a hidden form input:

```
<input type="hidden" value="%zipcode" name="zip">
```

The proxy servers also determine if a query comes from a valid Palm VII handheld; if so, they can fill in the `%deviceid` variable with the unique identifier contained in a particular handheld's ROM. Include `%deviceid` in a hidden form input like this:

```
<input type="hidden" value="%deviceid" name="id">
```

Possible return values for `%deviceid` comprise the following:

- `1` if the proxy server recognizes the device ID as valid
- `0` if the query may or may not have come from a Palm VII handheld
- `-1` if the query did not originate from a valid Palm VII handheld
- A string unique to the user and handheld

You may also include the `%zipcode` and `%deviceid` variables as part of the URL submitted for the query:

```
<form action="http://www.companyname.com/
cgi?zip=%zipcode&id=%deviceid" method="get">
```

Adding Images

You can add images to a PQA using the standard HTML `<img>` tag. Clipper understands images in both GIF and JPEG formats, with color depths of either 1 bit (black and white) or 2 bits (four-color grayscale). Images may be no wider than 153 pixels, the maximum display width available in the Clipper window. A suggested maximum height for images is 144 pixels, which is as much vertical space as Clipper can display at once. Vertically, larger images may be scrolled, so 144 pixels is not a hard and fast limit on the height of an image.

Graphics tend to be space hogs in a PQA. Try to minimize the number of images included in an application, and make those images that you do include as small and as simple as possible.

The following example shows the syntax for inserting an image into a PQA:

```
<img src="image.gif">
```

Just as with linking to other HTML pages in a PQA, you may include images in subdirectories and reference them using a relative path:

```
<img src="subdirectory/image.gif">
```

Adding PQA icons

Every PQA should have both large and small icons to display in the system application launcher. Just like standard Palm OS application icons, large icons should be 32×22 pixels, and small icons should be 15×9 pixels. The Query Application Builder can make icons from BMP, GIF, or JPEG format images. If you do not explicitly add your own icons to a PQA, the Query Application Builder provides default icons.

The default icons used by the Query Application Builder are shipped with the builder as separate BMP files. You can use these files as a starting point for your own PQA icons.

Using Palm Image Checker

To get a feel for what a particular image will look like once it is on the handheld, you can use the Palm Image Checker. In the Windows version of the Query Application Builder, the image checker is a separate application, called `pic.exe`. The Mac OS version of the Query Application Builder includes the image checker as a built-in function. Figure 16-5 shows the Palm Image Checker running on a Windows system.

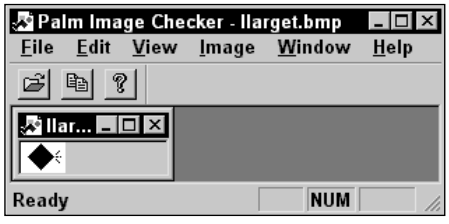


Figure 16-5: The Palm Image Checker

To open an image, click the Open button in the toolbar (signified by an open folder icon), and select the File➪Open menu option, or press Ctrl+O. Select an image to display from the file dialog box, and image checker then displays the image as it will appear in Clipper.

You can also use the image checker to resize images. Once you have an image open, select the Image➪Resize Image menu option, or press Ctrl+S. The New Image Dimensions dialog box, pictured in Figure 16-6, appears.

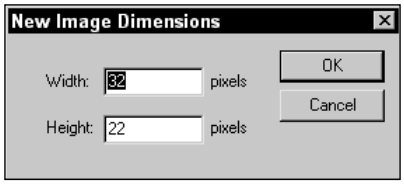


Figure 16-6: The New Image Dimensions dialog box

Enter the new image dimensions, and then click the OK button. Palm Image Checker opens a new window containing the resized image. From here, you may copy the image to the clipboard (using the Copy button or the Edit➪Copy menu option, or pressing Ctrl+C) and paste the resized image into a graphics-editing program of your choice.

Palm Image Checker does not actually modify any image files. If you want to use the image checker to resize images, or convert them into black and white or grayscale, you need to have a separate graphics program into which you can paste modified images.

Using the Query Application Builder

Once you have assembled all the HTML pages and images that compose a PQA, you need to build it into an installable `.pqa` file with the Query Application Builder. The interface for the Windows version of the Query Application Builder is shown in Figure 16-7; the Mac OS interface is nearly identical.

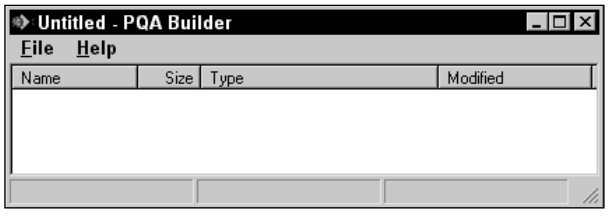


Figure 16-7: The Query Application Builder's main screen

Opening an index file

The first step toward building a PQA is to open the project's home page, which you can do by selecting File ➪ Open Index or by pressing Ctrl+O. A standard file dialog box appears, prompting you for the location of the PQA's index file. Once you select the index file and click the Open button, the Query Application Builder looks through the home page for links to other pages and images that make up the PQA. Assuming that there are no errors in your PQA, the Query Application Builder then displays all the files that make up the PQA's interface, as shown in Figure 16-8.

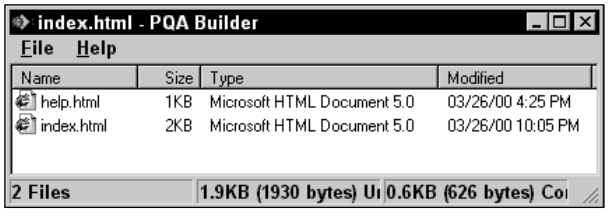


Figure 16-8: The Query Application Builder, after loading an index file

The status bar at the bottom of the Query Application Builder gives you a comparison of the uncompressed and compressed sizes of the PQA.

Building a PQA

Once you have a PQA loaded into the Query Application Builder, select File➪ Build PQA, or press Ctrl+B, to build the PQA. The Build PQA dialog box, pictured in Figure 16-9, appears.

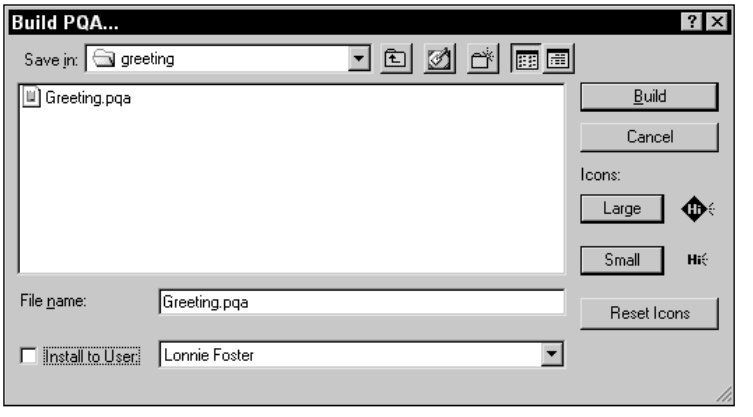


Figure 16-9: The Build PQA dialog box

In this dialog box, you need to do three things:

- Set the location and file name for the finished `.pqa` file in the file dialog box in the upper-left area of the Build PQA screen. The Mac OS version of the Query Application Builder gives you a separate space to define the PQA's display name in the Palm OS application launcher, which might be different from its file name on the desktop computer. The Windows version does not have this option, instead using the beginning of the file name before the `.pqa` extension for the display name. You can, however, change the display name to something else in the Windows Query Application Builder by calling it from the command line; see below for details.
- Check the Install to User check box and select an appropriate user if you want to immediately install the PQA to a connected handheld at the next HotSync operation.
- Set the large and small icons for the PQA with the Large and Small buttons.

Pressing the icon buttons opens a standard file dialog box, prompting you for the BMP, GIF, or JPEG images that you wish to use for the PQA's small and large launcher icons. The Reset Icons button resets the icons to the default blank PQA icons that ship with the Query Application Builder.

Building a PQA from the command line

You may also run the Windows version of the Query Application Builder from the command line, which allows you to build a PQA as part of a batch process. Table 16-2 shows the parameters you may pass to the `qab.exe` application and how those parameters affect the building of a PQA.

Table 16-2
QAB Command-Line Parameters

Parameter	*Description*
`/h`	Displays a help dialog box, which explains how to use the command-line options for `qab`
`/pqa "index.html"`	Builds a PQA, using the file `index.html` as the PQA's home page
`/n "pqaname"`	Specifies the display name for the PQA that appears in the Palm OS application launcher
`/o "output.pqa"`	Specifies the output file name for the compiled PQA
`/l "largeicon.bmp"`	Specifies the file name of the image to use for the PQA's large icon
`/s "smallicon.bmp"`	Specifies the file name of the image to use for the PQA's small icon
`/u "username"`	Installs the compiled PQA during the next HotSync operation, using `username` to specify the user to synchronize with
`/v`	Verbose mode; displays more detailed error strings if the PQA could not be compiled

As an example, the following command line compiles the PQA described in the next section:

```
qab /pqa "index.html" /n "Greeting" /o "Greeting.pqa"
    /l "largeicon.bmp" /s "smallicon.bmp"
```

Looking at a Sample PQA

The Formal Greeting Generator is a complete, if simple, PQA. Its purpose is to generate a personalized greeting, taking into account the user's preferred title and the time of day. Figure 16-10 shows the Formal Greeting Generator in action.

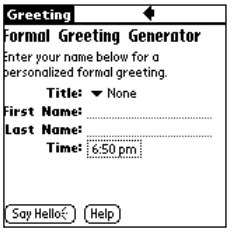


Figure 16-10: The Formal Greeting Generator sample PQA

The home page for the Formal Greeting Generator, defined in the file `index.html`, appears in Listing 16-1.

Listing 16-1: **The Formal Greeting Generator's home page, index.html**

```
<html>
<head>
<title>Greeting</title>
<meta name="palmcomputingplatform" content="true">
<meta name="palmlauncherrevision" content="1.0">
</head>
<body>
<h2>Formal Greeting Generator</h2>
<p>Enter your name below for a personalized formal
greeting.</p>

<form action="http://www.palmosbible.com/greeting.cgi"
 method="get">
<table>
  <tr>
    <td align=right><b>Title:</b></td>
    <td><select name="title">
        <option selected value="0">None
        <option value="1">Mr.
        <option value="2">Mrs.
        <option value="3">Ms.
        <option value="4">Dr.
        <option value="5">General
        <option value="6">King
        <option value="7">Queen
        </select></td>
  </tr>
  <tr>
    <td align=right><b>First Name:</b></td>
    <td><input type="text" name="fname" size="15"
         maxlength="30"></td>
  </tr>
```

Continued

Listing 16-1 *(continued)*

```
    <tr>
      <td align=right><b>Last Name:</b></td>
      <td><input type="text" name="lname" size="15"
           maxlength="30"></td>
    </tr>
    <tr>
      <td align=right><b>Time:</b></td>
      <td><input type="timepicker" name="time"></td>
    </tr>
  </table>
  <pre>  </pre>
  <pre>  </pre>
  <input type="submit" value="Say Hello">
  <a href="help.html" button>Help</a>
  </form>
  </body>
  </html>
```

The `action` URL used by the Formal Greeting Generator to create its greeting calls the `greeting.cgi` script on a remote server. The `greeting.cgi` script is described in the next section of this chapter, "Building Web Clippings."

Formal Greeting Generator presents a pop-up list, from which the user may select a title. Notice that the `value` attributes in this pop-up list are not the actual titles themselves, which would take up unnecessary bytes if sent as part of the query. Instead, `index.html` just uses the zero-based index of the selected list item, which `greeting.cgi` expands into the appropriate string on the server end when generating a Web clipping response. This technique is a little harder to maintain and debug, since any changes to the pop-up list in `index.html` must be duplicated on the server end, but it is worth the extra trouble to make the query smaller and thereby save wireless bandwidth.

The time picker automatically displays the current time according to the handheld's clock, so if the user leaves this picker alone, the greeting returned as a result of Formal Greeting Generator's query will reflect the current time of day.

A Help button at the bottom of the page is a link to the local file `help.html`, shown in Listing 16-2. This sort of help is information that does not change, so it is not appropriate to store it on the server and require the user to query for it; instead, it is a second page built into the PQA itself.

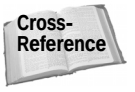
Listing 16-2: **The Formal Greeting Generator's help page, help.html**

```
<html>
<head>
<title>Greeting</title>
<meta name="palmcomputingplatform" content="true">
</head>
<body>
<h2>Formal Greeting Help</h2>
<p>The fields on the home page have the following meanings:</p>
<ul>
  <li><b>Title</b>. Sets the title by which you wish to be
                    addressed in the greeting.
  <li><b>First Name</b> and <b>Last Name</b>. Enter your first
                    and last names in these text fields.
  <li><b>Time</b>. Enter the time of day for which you wish to
                   be greeted, or leave the time picker alone
                   to use the current time of day according to
                   your handheld's clock.
</ul>
<pre>  </pre>
<a href="index.html" button>Return to Home Page</a>
</body>
</html>
```

Building Web Clippings

The biggest difference between creating a PQA and creating a Web clipping is that Web clippings tend to be generated on-the-fly from some sort of CGI program at the server end. You can link to static HTML pages from a PQA, but there is little point in doing so, since any information that does not change very often should probably be built into the PQA itself. Therefore, when building the server end of a Web clipping solution, you will probably write a program on the server end that dynamically generates HTML pages to return to the PQA as Web clippings.

There are far more languages and tools available for generating dynamic Web content than this book could possibly cover. Whatever tools you use to generate Web clippings, you need only make sure that the HTML pages created are properly formatted for use by Clipper. The Web clipping end of the sample Formal Greeting Generator in this book uses a Perl CGI script to generate Web clippings in response to queries from the Formal Greeting Generator PQA. You do not need to understand Perl CGI programming to create Web clippings, though; you should be able to adapt the techniques in this section to any language or system to generate dynamic content.



If you are unfamiliar with Perl CGI programming, you can find an excellent primer on the Web at `http://www.cgi101.com`.

Most of the details of building Web clippings are identical to the details of building a PQA. The primary difference to keep in mind when assembling Web clippings is that the entire clipping is returned via the wireless network. Even more than with PQAs, you must strive to keep Web clippings short and to the point to prevent wasting precious airtime.

Defining Header Tags

A Web clipping should have the same `palmcomputingplatform` `<meta>` tag as a PQA page. Without the `palmcomputingplatform` tag, the Palm.Net proxy server truncates the page by 1KB and removes all images, so it is important to include this tag so the proxy knows that the page is formatted properly for use as a Web clipping.

In addition, you should include a `<meta>` tag containing a `historylisttext` attribute. This tag has the following syntax:

```
<meta name="historylisttext" content="string &date &time">
```

Clipper uses the contents of the `historylisttext` tag to keep track of cached clippings. Whatever text you supply for `string` in the `content` attribute appears in the History pop-up list in the upper right of Clipper's title bar. The `&date` and `&time` variables are replaced with the date and time when the user made the query. If you omit the `historylisttext` tag entirely, Clipper will use the name of your application in the History pop-up list.

You do not need to use `palmlauncherrevision` in a Web clipping page.

Creating Clipping Pages for Desktop Browsers

If you want to make a CGI program that does double duty, serving pages that work for display in both Clipper and in more traditional desktop browsers, you can use the `<smallscreenignore>` tag to hide content from the handheld. For example, if you have a big full-color image that you would like to display in the Web page for the benefit of regular Web browsers, you can place it within a `<smallscreenignore>` tag to keep it from being displayed in Clipper:

```
<smallscreenignore>
<img src="hugeimage.jpg" alt="Huge image">
</smallscreenignore>
```

Linking Outside the Web Clipping

Hyperlinks in a Web clipping may point to the same things that links in a PQA may point to. Any links from a Web clipping to pages in a local PQA require a `file:` URL, like this:

```
<a href="file:my.pqa/page.html">Link text</a>
```

Remember that links from a Web clipping to a PQA must treat the pages in the PQA as if they were all part of the same flat directory; no subdirectories are available in a PQA to anything outside that PQA.

The `palm` and `palmcall` URLs have the same effect from a Web clipping that they do from within a PQA, as do `mailto` URLs.

Adding Images

In general, it is a bad idea to download images through a Web clipping, though you may include images from the server with the standard `<img>` tag:

```
<img src="image.gif">
```

A better approach is to store in the PQA itself images that the Web clipping might need to display. If there is no reference to an image elsewhere in a PQA with the `<img>` tag, you can make sure it is included in the PQA by using the `localicon` `<meta>` tag in the header of one of the PQA's HTML pages:

```
<meta name="localicon" content="image.gif">
```

Once you have included an image in a PQA with `localicon`, you can access it from a Web clipping with the following syntax:

```
<img src="file:my.pqa/image.gif">
```

In fact, you can also use `localicon` to store entire pages in a PQA that are only referred to from Web clippings:

```
<meta name="localicon" content="page.html">
```

You may then refer to these pages just as you would any other local pages from a Web clipping.

Looking at a Sample Web Clipping

The Formal Greeting Generator PQA described earlier in this chapter has a companion Perl CGI script, called `greeting.cgi`, which generates Web clippings based on input from the PQA. The complete `greeting.cgi` script is listed at the end of this

section in Listing 16-3, but it is easiest to see how `greeting.cgi` and Formal Greeting Generator work together by looking at the actual clipping generated by some specific input to the PQA.

In this example, Figure 16-11 shows the Formal Greeting Generator as it appears after the user enters some sample query information, but just before the user taps the Say Hello button.

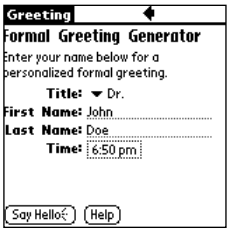


Figure 16-11: The Formal Greeting Generator PQA, ready to go with some sample data

When the user sends this query, the following parameters are passed to the `greeting.cgi` script:

```
title=4&fname=John&lname=Doe&time=18:50
```

Notice that the time is sent as a 24-hour value in the form `HH:MM`; `greetin.cgi` must be able to parse this time properly to come up with the time of day for the greeting it generates.

The `greeting.cgi` script parses the parameters for the title, first name, last name, and time values, and then uses those values to assemble a greeting appropriate for the title, name, and time of day. Figure 16-12 shows the clipping returned by `greeting.cgi` for these particular input values.

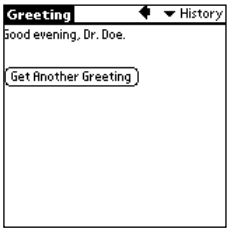


Figure 16-12: A Web clipping returned by greeting.cgi

The HTML for the clipping returned by `greeting.cgi` looks like this:

```
<html>
<head>
<title>Greeting</title>
```

```
<meta name="historylisttext"
    content="Greeting - &date &time">
<meta name="palmcomputingplatform" content="true">
</head>
<body>
<p>Good evening, Dr. Doe.</p>
<pre>  </pre>
<a button href="file:Greeting.pqa/index.html">
    Get Another Greeting</a>
</body>
</html>
```

The Web clipping returned contains a handy button link to the Formal Greeting Generator's home page in the local PQA, using the `file:` style of URL.

The Perl 5 CGI module, which `greeting.cgi` uses to create the actual HTML code itself, does not format HTML nearly as nicely as the listing above. The CGI module tends to omit linefeeds, resulting in HTML source that is hard to read for humans, but perfectly acceptable to browser applications. An added bonus of omitting linefeeds, however, is that it saves a few bytes in the returned Web clipping.

Listing 16-3 shows the `greeting.cgi` script, which contains all the Perl code necessary to respond to queries from the Formal Greeting Generator PQA:

Listing 16-3: **The greeting.cgi script**

```
#!/usr/bin/perl
# hello.cgi - Sample Web clipping application for the
#             Palm OS Programming Bible
use strict;

use CGI qw(:standard);


# Set up the titles array.
my @titles = ("None",
              "Mr.",
              "Mrs.",
              "Ms.",
              "Dr.",
              "General",
              "King",
              "Queen");

# Retrieve parameters
my $titleIndex = param("title");
my $firstName  = param("fname");
```

Continued

Listing 16-3 *(continued)*

```
my $lastName   = param("lname");
my $time       = param("time");

# Assemble an appropriate formal name if there is enough name
# to provide a proper greeting.
my $titleName = "";
if ($titleIndex == 0) {
    if ($firstName && $lastName) {
        $titleName = $firstName . " " . $lastName;
    }
} elsif ($titleIndex >= 1 && $titleIndex <= 5) {
    if ($lastName) {
        $titleName = $titles[$titleIndex] . " " . $lastName;
    }
} elsif ($titleIndex >= 6 && $titleIndex <= 7) {
    if ($firstName) {
        $titleName = $titles[$titleIndex] . " " . $firstName;
    }
}

# Make an appropriate string for the time of day.
my $greetTime;
$time =~ s/:.+//;
if ($time < 4 || $time >= 22) {
    $greetTime = "night";
} elsif ($time >= 4 && $time < 12) {
    $greetTime = "morning";
} elsif ($time >= 12 && $time < 17) {
    $greetTime = "afternoon";
} else {
    $greetTime = "evening";
}

# Output the Web clipping.
print header();
print start_html(-title => 'Greeting',
                 -meta  => {'palmcomputingplatform' => 'true',
                            'historylisttext' =>
                                'Greeting - &date &time'});

# Check to see if there is a title name. If title name has not
# been assembled, there is not enough data to form a proper
# greeting. Present an appropriate message to the user.
if (! $titleName) {
    my $errTitle;
    my $errName;
    if ($titleIndex == 0) {
        $errTitle = ", but without a title,";
```

```
        $errName  = "first and last names";
    } elsif ($titleIndex == 1) {
        $errTitle = " sir, but";
        $errName  = "last name";
    } elsif ($titleIndex == 2) {
        $errTitle = " ma'am, but";
        $errName  = "last name";
    } elsif ($titleIndex == 3) {
        $errTitle = " miss, but";
        $errName  = "last name";
    } elsif ($titleIndex == 4) {
        $errTitle = " doctor, but";
        $errName  = "last name";
    } elsif ($titleIndex == 5) {
        $errTitle = " General, but";
        $errName  = "last name";
    } elsif ($titleIndex == 6) {
        $errTitle = " your Majesty, but";
        $errName  = "first name";
    } elsif ($titleIndex == 7) {
        $errTitle = " your Majesty, but";
        $errName  = "first name";
    }
    print p("I'm sorry$errTitle I need your $errName to give",
            " you a proper greeting.");
    print p("Tap the back arrow above and try again.");
} else {
    print p("Good $greetTime, $titleName.");
}

print pre("  ");
print a({href => 'file:Greeting.pqa/index.html',
         button => undef},
        "Get Another Greeting");
print end_html();
```

Testing Web Clipping Applications

Testing the local side of a PQA is easy — all you need to do is install the PQA on the Palm OS Emulator (POSE), or an actual Palm OS device that supports Web clipping. In particular, you should be on the lookout for page formatting problems, missing images, and links that do not call up the correct page.

To test the actual online links in a PQA, you can install the PQA on an actual handheld that supports Web clipping. However, this can become very expensive if you have a lot of testing to do, since each query must go through a standard Palm.Net account.

Fortunately, POSE can redirect over-the-air queries to its host computer's net connection, using the desktop machine as a replacement for the wireless network when the Emulator hooks your PQA up to the Palm.Net proxy servers. Using this proxy system is free of charge, and considerably more convenient for heavy-duty debugging.

To set up POSE for PQA testing, start up the Palm OS Preferences applet by tapping the Prefs icon in the system launcher application. In the Preferences applet, select Wireless from the pop-up list in the upper right-hand corner. Figure 16-13 shows the resulting screen in the Preferences application.

Figure 16-13: The Wireless screen in the Palm OS Preferences applet

The **Proxy** selector trigger in the center of the screen should display the IP address `207.240.80.136`, which is a specific proxy server that Palm Computing has set up for testing purposes. If the selector displays a different address, tap it, and enter the correct address in the dialog box that appears.

Palm Computing may move the proxy server in the future. If the IP address listed above does not work, take a look at `http://www.palm.com/devzone/webclipping` for the current address of the proxy server.

After verifying the proxy address, select Settings ➪ Properties from POSE's menu. The dialog box pictured in Figure 16-14 will appear.

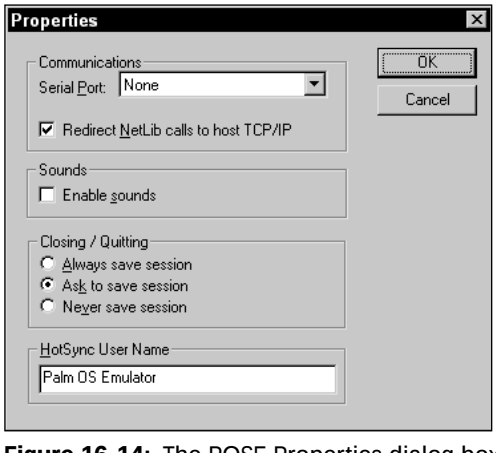


Figure 16-14: The POSE Properties dialog box

Make sure that Redirect NetLib calls to host TCP/IP is checked. If it is, POSE redirects all network library calls to your desktop machine's network connection.

With the proxy address and NetLib redirection set up, PQAs in POSE should connect to remote Internet servers as if POSE were a real Palm OS device. The only imperfections to this masquerade involve the `%zipcode` and `%deviceid` variables mentioned earlier in this chapter. Since the proxy connection through POSE does not actually hook up to a wireless base station, there is no ZIP code information available, and POSE does not contain the unique identifier present in an actual handheld's ROM. To test these two features, you will have to use a real Palm OS device.

Summary

In this chapter, you learned about how to take advantage of the wireless features of some Palm OS devices. After reading this chapter, you should know the following:

- Palm Computing's wireless Internet access model, called Web clipping, is built around a query and response architecture to save on expensive wireless bandwidth.
- A query starts at a Palm Query Application on the handheld, passes via radio connection to the Palm.Net proxy servers, and connects to an Internet site; the response from the Internet passes back through the proxy for processing, and then travels across the wireless network again to arrive at the Palm OS handheld.
- Web clipping solutions may be easily designed using standard HTML tools, but you do need to keep in mind the limitations of expensive wireless bandwidth, small screen real estate, and a slower processor when designing Web content for Clipper, the Palm OS Web-browsing application.
- The Query Application Builder is the tool you use to convert HTML pages and images into a PQA, which you can then install on a handheld that supports Web clipping.
- You can use any dynamic Web content generation system you wish to create Web clippings, as long as it is able to work within the guidelines required for designing Clipper Web content.
- You can test PQAs with an actual Web clipping-enabled handheld, or through POSE, using your desktop computer instead of an actual wireless network connection.

✦ ✦ ✦

CHAPTER 17

Introducing Conduit Mechanics

In This Chapter

Understanding conduits

Designing conduits

Installing conduits

Logging actions in the HotSync log

Conduits are code modules that perform synchronization between Palm OS handheld applications and data on a desktop computer. The HotSync Manager calls conduits during a HotSync operation to keep records in sync between the desktop and the handheld, back up data from the handheld to the desktop, or download data from the desktop to the handheld.

The Conduit Development Kit (CDK), available as a free download from Palm Computing, contains all the templates, object classes, and documentation necessary to create conduits for the Mac OS and Windows operating systems. To develop conduits for the Mac OS, you also need to have Metrowerks CodeWarrior version 3.0 to 3.3 (the full version of CodeWarrior, not just Metrowerks CodeWarrior for Palm Computing platform). To create conduits for Windows, you need to either use Microsoft Visual C++ 6.0 and the CDK for Windows, or Symantec Visual Cafe Pro for Java and the CDK Java Edition for Windows. The CDK Java Edition has also been tested with Microsoft Visual J++ 1.1, but Visual Cafe Pro is the development platform supported by Palm Computing.

The CD-ROM included with this book contains the 4.0 versions of the CDK for Mac OS, the CDK for Windows, and the CDK Java Edition for Windows.

Part VI, "Synchronizing Data with the Desktop," focuses on conduit development for Windows, using Visual C++ and the CDK for Windows. However, much of the information in this chapter is conceptual and applies just as well to Java and Mac OS conduit development as it does to Visual C++ conduit development. The next chapter, "Building Conduits," delves into the actual details of conduit programming using Visual C++, so it will not be as useful as this chapter for Mac OS or Java conduit developers.

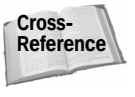
Because conduit development involves creating software to run on a desktop machine instead of a Palm OS handheld, it requires a different set of skills and tools from those needed for handheld application development. You should be familiar with object-oriented programming in C++ or Java, depending on the CDK you wish to use, and it also helps to have some experience with creating dynamic link libraries (if using the CDK for Windows) or Code Fragment Manager plug-ins (if using the CDK for Mac OS). Knowing your way around Microsoft Foundation Classes (MFC) is a must if you plan to build a conduit based on the Palm MFC Base Classes.

Understanding Conduits

A standard Windows conduit is a DLL module with entry points called by the HotSync Manager (Mac OS conduits are Code Fragment Manager plug-in modules, and Java conduits are Java classes that use the `jsync.dll` module to communicate with the HotSync Manager). A conduit is only one piece of software that must cooperate with a number of other programs to transfer data between a handheld and a desktop computer. Some or all of the following components may be involved in a given HotSync operation:

- **HotSync Manager.** This program controls the entire HotSync process. The HotSync Manager runs in the background on the desktop computer, watching appropriate communications ports for a HotSync request from a Palm OS handheld. The HotSync Manager handles basic communication between the desktop and a handheld, manages multiple users synchronizing with the same desktop machine, provides an interface with which users can customize the behavior of individual conduits, installs new applications and databases to a handheld, and restores data on the handheld in the event of a hard reset or other catastrophic data loss.
- **Conduits.** Conduits are plug-in modules that handle the actual transfer of data between a handheld application and a desktop data source. During a HotSync operation, the HotSync manager calls each registered conduit in turn to synchronize a handheld application with its desktop data. A conduit does not require any user interaction to perform its duties, instead relying on internal logic to correctly modify data on the desktop, the handheld, or both. This lack of interaction is important to remember when designing a conduit; if a user synchronizes a handheld remotely, there is no way for the conduit to prompt the user for input.

- **Notifier DLLs.** If both a conduit and a desktop application can modify the same data, it may be necessary to tell the desktop application to leave the data alone during the course of a HotSync operation, thereby preventing data loss, duplicate records, or just plain mangled data. The HotSync Manager uses a process called *notification* to prevent this sort of mess. Before the HotSync Manager launches a conduit to perform data transfer, the manager calls a *notifier DLL* for that particular conduit. The notifier DLL in turn passes information to the appropriate desktop application in a format that the application understands. The best example of this process is the Palm Desktop application, which does not allow the user to change any data during a HotSync operation; the Palm Desktop knows that a HotSync operation is in progress because it was notified by its notifier DLL, `pdn20.dll`.
- **Handheld applications.** A handheld application may serve as a quick data collection tool for a desktop application or as a portable viewer for information imported from the desktop. Also, an application on the handheld may simply share data with a desktop application, just as the four main ROM applications share data with the Palm Desktop program. If you follow standard Palm OS programming guidelines, there is nothing you need to add to a handheld application to allow it to work with the HotSync process.



See Chapter 13, "Manipulating Records," for more details about handling data in a Palm OS application.

- **Desktop applications.** Because of the flexibility inherent in conduit design, virtually any desktop application may share data with a Palm OS handheld. A desktop application can create data to send to the handheld, process data retrieved from the handheld, or share data with the handheld.
- **Sync Manager API.** This application programming interface allows conduits to communicate with the handheld, regardless of how the handheld is connected to the desktop computer. The Sync Manager API can directly read and write data on the handheld, and it forms the most basic layer in conduit programming.

Figure 17-1 shows the relationships between the various components that may be present in a Palm OS synchronization scheme. The arrows in the figure show how data flows between different pieces of software. In most cases, components perform two-way communication, sending data in both directions between the components; notification, however, usually flows in one direction only, from the HotSync Manager to a notifier DLL to a desktop application.

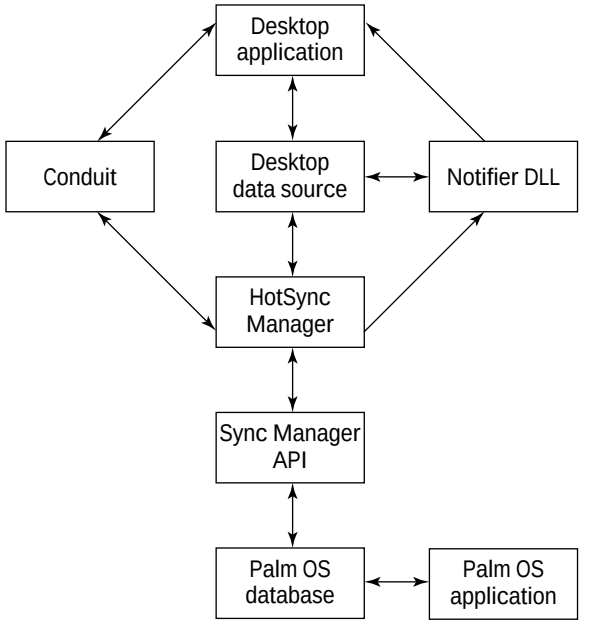


Figure 17-1: Data flow between the HotSync Manager and other software components

Stepping Through the HotSync Process

When the user initiates a HotSync operation, either by pressing the HotSync button on the cradle or by tapping the HotSync button in the HotSync application on the handheld, the HotSync Manager springs into action, following a particular series of steps to synchronize the desktop and the handheld.

A HotSync operation may be initiated only from the handheld. Because of limitations in current cradle hardware, there is no way to start the synchronization process from the desktop computer.

The following steps outline the actions taken by the HotSync Manager during a HotSync operation:

1. **User validation and location.** Each Palm OS handheld has a unique *user ID* associated with it. When the user synchronizes the handheld for the first time, the HotSync Manager assigns a pseudo–random number to that particular handheld, which allows a single desktop computer to synchronize with multiple handhelds and still keep their data separate. At the beginning of a HotSync operation, the HotSync Manager makes sure the user ID on the handheld is valid, and then locates the path to that particular user's data on the desktop computer. For example, my own user path is `c:\Palm\FosterL`. Most conduits save their information in subdirectories of the user path.

2. **Determination of synchronization type.** A conduit may perform two kinds of synchronization: *SlowSync* and *FastSync*. In a SlowSync, the conduit compares each record in the handheld database with its corresponding record in the desktop data source. In a FastSync, the conduit compares only records whose modification flag is set.

   The HotSync Manager determines which type of synchronization to perform by looking at the *PC ID* stored on the handheld from its last HotSync operation. Like a user ID, the PC ID is a pseudo–random number generated by the HotSync Manager to uniquely identify a desktop computer. Whenever a HotSync operation takes place, the HotSync manager stores the PC ID of the last desktop machine the handheld synchronized with on the handheld itself.

   If the handheld was last synchronized with the same machine as the current HotSync operation, the HotSync Manager tells conduits to perform a FastSync. If the last machine the handheld synchronized with was a different machine, conduits cannot rely on the modification flag of each record being valid, so in this case the HotSync Manager tells installed conduits to perform a SlowSync.

3. **Desktop application notification.** The HotSync Manager calls the appropriate notifier DLLs to let desktop applications know that the HotSync Manager is about to modify data shared between desktop and handheld applications.

4. **Conduit setup.** Once notification is out of the way, the HotSync Manager retrieves the creator ID of each application on the handheld (databases with type `appl`). If a conduit is installed for a particular creator ID, the HotSync Manager adds that conduit to a list of modules that should be run.

   The HotSync Manager also looks through the other databases on the handheld that are not of type `DATA`. If such a database has its backup bit set, the HotSync Manager adds that database to a list that will be handled by the built-in Backup conduit.

5. **Installation.** Now the HotSync Manager uses its built-in Install conduit to install any databases that are queued up on the desktop computer. Typically, these databases were queued by the Palm Install Tool (`instapp.exe`). On Windows, the HotSync Manager knows there are databases to install when a particular Registry key is present, namely `\HKEY_CURRENT_USER\Software\U.S. Robotics\Pilot Desktop\HotSync Manager\InstallNNNNN`, where `NNNNN` is the pseudo–random user ID assigned to the handheld. The Palm Install Tool creates this Registry key when it queues databases for installation and then copies those databases to the `Install` subdirectory of the appropriate user data folder. For example, files awaiting installation on my machine go into the `c:\Palm\FosterL\Install` directory. The Install conduit looks in this particular directory for databases to install.

6. **Conduit execution.** The HotSync Manger cycles through the list of conduits it assembled in Step 4, calling each in turn to synchronize data between the handheld and desktop applications.

7. **Second Installation.** Version 3.0.1 or later of the HotSync Manager calls the Install conduit a second time, which gives the HotSync operation a chance to pick up any databases queued for install by any of the conduits. This step allows a conduit to generate a database and "push" it out to the handheld, which can be very useful for conduits that retrieve information from the Web or other network sources. For example, AvantGo uses this mechanism to install newly downloaded Web pages to the handheld. Note that prior to HotSync Manager version 3.0.1, this second installation phase does not happen.
8. **Database backup.** The HotSync Manager calls the Backup conduit to copy databases queued for backup in Step 4. These databases are stored in a `Backup` subdirectory of the appropriate user data directory. For example, my own backup directory is `c:\Palm\FosterL\Backup`.
9. **Synchronization information update.** Now that the HotSync Manager has completed most of its tasks, it updates the sync time, PC ID, and user ID, if necessary, in the HotSync application on the handheld. At this point, the HotSync Manager also transfers a shortened version of the HotSync log to the handheld, which the user may view to determine the nature of any errors or warnings generated by the HotSync operation.
10. **Second desktop application notification.** The HotSync Manager calls the appropriate notifier DLLs a second time, to alert applications that the HotSync operation is complete and it is now safe for desktop applications to modify shared data sources again.
11. **Handheld application notification.** The Palm OS itself gives notification of a finished HotSync operation to newly installed handheld applications, and those whose data was modified during the HotSync process, by sending a `sysAppLaunchCmdSyncNotify` launch code to each of these applications. Any application that needs to perform some operation immediately after installation or having its data modified by a HotSync operation, such as resetting alarms or registering to receive beamed data, may do so by handling the `sysAppLaunchCmdSyncNotify` launch code.

Designing Conduits

Each conduit can synchronize with a single application on the handheld. You can build a conduit to synchronize a custom Palm OS application with a custom data source on the desktop, or to synchronize one of the built-in handheld applications with a custom desktop data source. You could also hijack the Palm Desktop data and synchronize it with your own handheld application, or even replace one of the built-in conduits so it synchronizes a built-in handheld application with the default Palm Desktop in a different way, although these two scenarios are less likely.

Given the different databases you can synchronize with each other, there are also four different types of synchronization you can perform. From most complex to least complex, here are the different styles of synchronization:

- **Transaction-based.** In a transaction-based scenario, the desktop computer must perform some sort of processing between each record synchronization. For example, a conduit that updates information on the handheld from a live Internet source would require a transaction-based approach. This style of synchronization takes a lot longer than other types and should be used only when absolutely necessary, since it slows down the entire HotSync process.
- **Mirror image.** In this scenario, modifications to records may be made on both the handheld and the desktop, and the conduit makes the databases identical on both platforms. The conduit must also resolve conflicts when the user modifies the same record on both the desktop and the handheld. Palm's four basic built-in applications use mirror image synchronization with the Palm Desktop.
- **One-directional.** Only one side of the connection — either the desktop or the handheld — may modify data in a one-directional scenario. This type of synchronization is ideal for desktop applications that update some sort of data, such as stock quotes, and then dump that data to the handheld for remote viewing. One-directional synchronization also works well the other way, for applications that use the handheld as a data collection device, and then save that raw data to the desktop computer for further processing.
- **Backup.** If an application does not have a desktop component, it can rely on the default Backup conduit provided with the HotSync Manager. This could also work as a "poor man's" one-directional synchronization if a desktop application can parse the `.pdb` file used by the Palm OS to store the handheld application's records on the desktop. An actual one-directional conduit is easier to use, though.

Consider speed when picking a style of synchronization. Try not to over-engineer your conduit; use the simplest type of synchronization that will get the job done.

Rapid synchronization is a vital part of the Palm Computing philosophy, and it has been a key factor in the popularity of the Palm Computing platform. Also, because most HotSync operations take place through the handheld's serial port, which is a major drain on the batteries, it is imperative to keep the total synchronization time as short as possible. Try to design conduits to execute quickly and efficiently.

Choosing a Development Path

The CDK for Windows gives you three basic starting points for building a conduit, each with its own strengths and weaknesses:

- Start with the *Palm MFC Base Classes* and customize them for your application.
- Start with the *Palm Generic Conduit Base Classes* and customize them for your application.
- Start from scratch, calling *Sync Manager API* functions directly from your conduit.

The Palm MFC Base Classes are a set of C++ object classes based on the Microsoft Foundation Classes (MFC), and they provide a high-level interface to the HotSync process. You must customize the base classes to interact properly with the handheld application's data format. On the desktop side, an MFC conduit reads and writes data using the MFC serialization format. To access this data from your own desktop application, that application must be an MFC program that also makes use of the Palm MFC Base Classes. The Palm MFC Base Classes provide all the logic necessary to perform mirror image synchronization; all you need to do is fill in the details about the data format from the handheld side of the HotSync operation.

The Palm Generic Conduit Base Classes are a different set of C++ object classes, which also provide a high-level interface to the HotSync process. Unlike the Palm MFC Base Classes, a generic conduit may be customized for different data formats on both the handheld and on the desktop, so the Palm Generic Conduit Base Classes are ideal for connecting a handheld application with a standard desktop database format, such as ODBC or plain old comma-delimited value format. The generic base classes are also designed with portability in mind; the same source code works on both the Mac OS and Windows, making generic conduits ideal for cross-platform conduit development. Like the Palm MFC Base Classes, the generic conduit classes also provide their own synchronization logic, allowing you to concentrate on filling in code to convert between handheld and desktop data formats.

Generic conduits are the wave of the future as far as Palm Computing is concerned. The Palm MFC Base Classes were originally the only high-level conduit development classes available, but starting with the CDK version 3.0, Palm Computing released the newer Palm Generic Conduit Base Classes as an unsupported feature. Because generic conduits are more flexible and can be used for cross-platform development, Palm Computing suggests that developers who are new to conduit development start with generic conduits instead of MFC conduits.

The Sync Manager API is a set of low-level functions that directly control interaction between the desktop and the handheld. Both MFC and generic conduits use the Sync Manager API to perform the basic tasks of sending and receiving data during a HotSync operation. The base classes take care of all the synchronization logic required to keep a desktop data source in mirror image synchronization with a handheld application. If your application does not require mirror image synchronization, or if you want to implement your own sync logic, directly controlling the handheld through the Sync Manager API requires less overhead than using the base classes and can result in a much quicker and more efficient conduit.

Installing Conduits

A conduit cannot run without first being properly registered. Registration tells the HotSync Manager that it should call your conduit when syncing databases with a specific creator ID. For a conduit to be a good HotSync citizen and play nicely with other conduits, its installation program must be able to register and unregister the conduit without any user intervention and, more important, without damaging

the HotSync process by overwriting or removing conduit registration information that belongs to other conduits.

Prior to version 3.0 of the CDK, conduit registration involved adding entries directly to the Windows Registry. This situation was tenuous at best, since there was no mechanism in place to prevent one developer's registration or unregistration code from destroying the registration information of other conduits. Worse yet, uninstalling conduits often involved renaming existing Registry entries and, if not done carefully, could prevent the entire HotSync process from working properly.

With the introduction of the CDK 3.0, Palm Computing added the Conduit Manager, a set of functions for conduit registration and unregistration. The Conduit Manager functions reside in a DLL called `CondMgr.dll`, which ships with version 3.0 and later of the HotSync Manager. As of this writing, the HotSync Manager still uses the Windows Registry to store conduit registration information, but the Conduit Manager API hides the details of registration storage from developers, providing a much cleaner and safer interface for registering conduits. Not only is it easier to use the Conduit Manager than it is to try tiptoeing your way through the minefield of HotSync Registry entries, it separates the act of registering and unregistering conduits from the method used to store registration data, so it is possible that future versions of the HotSync Manager may not use the Windows Registry at all.

Installing Conduits Manually

When developing and testing a conduit, you can manually register and unregister it using the Conduit Configuration tool supplied with the CDK. This tool, pictured in Figure 17-2, provides a graphical interface for registering and unregistering conduits.

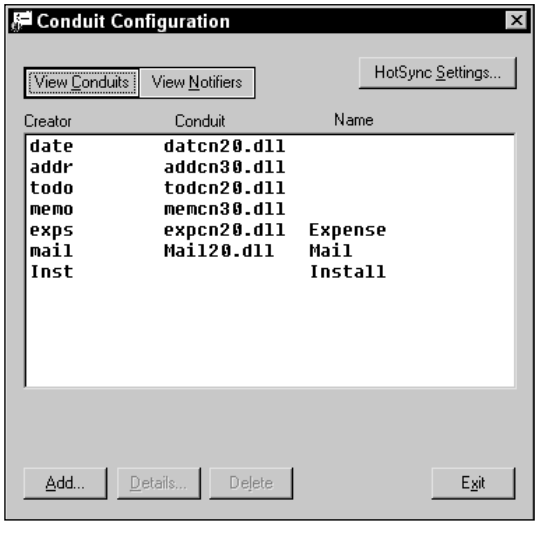


Figure 17-2: The Conduit Configuration tool

The Conduit Configuration tool itself is an executable called `CondCfg.exe`, which resides in the `bin\HSM\Release` and `bin\HSM\Debug` directories, underneath the directory where you installed the CDK. Use the copy in the `Release` directory with the release version of the HotSync Manager, and the copy in the `Debug` directory when using the debug version of the HotSync Manager.

If you use the same desktop machine for both conduit development and for syncing your own Palm OS handheld, be very careful when using the Conduit Configuration tool. Improper use of the tool can cause unpredictable HotSync behavior, preventing you from backing up your handheld's data. Use the Conduit Switch tool, described later in this section, to back up your HotSync registration settings before modifying them.

The View Conduits and View Notifiers buttons in the configuration tool display the currently registered conduits or notifier DLLs, respectively. Clicking the HotSync Settings button brings up a dialog box, pictured in Figure 17-3, from which you may change basic settings of the HotSync Manager itself.

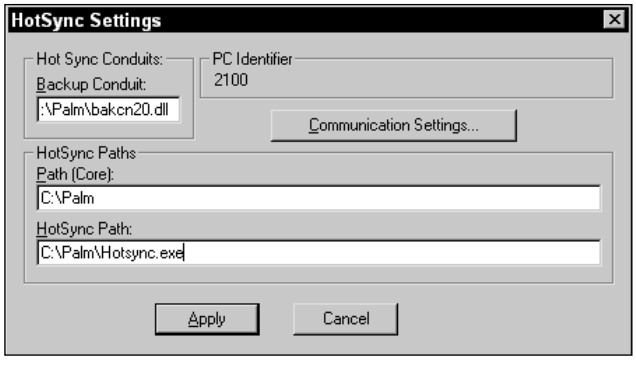


Figure 17-3: The Conduit Configuration tool's HotSync Settings dialog box

On the main Conduit Configuration tool screen, if the list is currently displaying conduits, clicking the Add button brings up the Conduit Information dialog box, shown in Figure 17-4. From this dialog box, you can enter the settings for a new conduit that you want to register.

The Conduit Type section at the top of the dialog box specifies whether the conduit is independent (Application) or integrated as part of the Palm Desktop application (Component). Generally, most third-party conduits are of the Application variety.

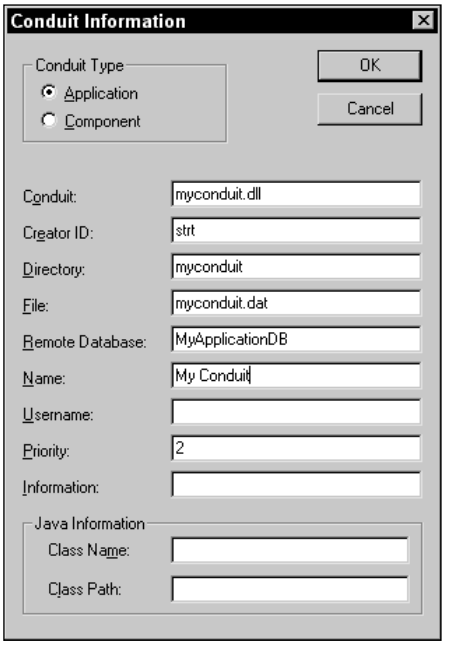


Figure 17-4: The Conduit Configuration tool's Conduit Information dialog box

The text boxes in the rest of the dialog box are pieces of registration information for the conduit. The first three are required to properly register a conduit, while the rest are optional. The entries are described below:

- **Conduit.** This is the file name of the conduit DLL. Without a complete path name, the file needs to be located either in the same directory as the HotSync Manager, or somewhere on the system path. Most conduit DLLs are placed in the HotSync directory. If your conduit is written in Java, the DLL listed here should be `JSync.dll`, which is a C++ DLL that communicates between the HotSync Manager and the Java-based conduit. This entry is required.
- **Creator ID.** Enter the creator ID of the database or databases on the handheld that this conduit should sync with into this text box. This entry is required and case-sensitive.
- **Directory.** This should be the name of a directory where your conduit's data files will be stored. The HotSync Manager creates this directory within each individual user's directory, and it serves to separate your conduit's files from those used by other conduits. This entry is required.

- **File.** The file name of the desktop data file with which your conduit synchronizes goes here. Without a complete path, this file is assumed to be in the conduit's Directory, entered above. This entry is optional, because your conduit might synchronize with more than one file on the desktop; the File entry is provided as a convenience for conduits based on the Palm MFC or generic base classes, which use this file name by default for the desktop data file. This field must contain at least one character, however, even if it is unused.
- **Remote Database.** This should be the name of the database on the handheld, which the conduit may use to create the database if it does not already exist. The name is case-sensitive, and like the File entry, must contain at least one character.
- **Name.** This entry is the display name of the conduit, which the HotSync Manager shows in its Custom dialog box and elsewhere.
- **Username.** This entry is not currently used, but it is intended to store the name of the user for whom this conduit is installed.
- **Priority.** Every conduit has a priority value that determines the order in which the HotSync Manager executes conduits. This value should be from 0 to 4, with 2 being the normal value for most conduits. The HotSync Manager runs conduits with lower values in the Priority field before conduits with higher values in this field. For example, a priority 1 conduit runs before one with priority 3.
- **Information.** This entry is used to display extra information to the user if there is a conflict between your conduit and another that wishes to access the same database on the handheld. A conduit installer might display this string and ask the user which of two conflicting conduits should be permitted to handle a particular database.

The two entries under Java Information are for Java-based conduits. See the CDK for Windows, Java Edition, for more information.

Back on the main Conduit Configuration tool screen again, clicking the Details button also brings up the Conduit Information dialog box, but with certain fields disabled. The conduit currently selected in the main screen's list is displayed. You can edit many of the registration entries for an installed conduit in this way.

To uninstall a conduit, select it from the list in the main screen, and then click Delete. In the confirmation dialog box that appears, click OK to delete the conduit's registration entries. Note that this does not delete the actual DLL file, only its registration with the HotSync Manager.

Any changes you make using the Conduit Configuration tool will not take effect until you restart the HotSync Manager.

Backing up and restoring HotSync configurations

Because changes made using the Conduit Configuration tool can potentially cause strange things to happen during a HotSync operation, it is a good idea to back up a working HotSync configuration before making changes so that you can restore it

later if necessary. Being able to save different configurations is also useful during development if you want to quickly switch back and forth between saved states to try out different scenarios.

The CDK comes with a command-line tool for easily making backups of the HotSync Manager's registration settings and restoring them again later. The Conduit Switch tool, an executable called `CondSwitch.exe`, is located in the `bin\HSM\Release` directory under the directory where you installed the CDK. To save the current HotSync configuration to a file, call the Conduit Switch tool with the following parameters:

```
CondSwitch -b backup_file.txt
```

Later, when you want to restore a configuration you saved earlier, use this syntax:

```
CondSwitch -d -i backup_file.txt
```

Just as in the Conduit Configuration tool, any changes you make to the HotSync configuration with the Conduit Switch tool will not take effect until you restart the HotSync Manager. To save time, you can tack the `-r` switch onto the `CondSwitch` command line to restart the HotSync Manager.

Creating Automatic Conduit Installations

Because your conduit's users do not have access to the Conduit Configuration tool (nor should they!), whatever program you use to install your conduit must take care of registering and unregistering your conduit at installation and de-installation time. The CDK for Windows contains a sample install script for version 5.5 of the popular InstallShield Professional program, which you can modify for your own conduit.

The InstallShield sample included with the CDK does not work with the free version of InstallShield that comes with Microsoft Visual C++. The free edition cannot call functions in a DLL, an action that is required for proper conduit registration.

If you want to create your own installation program, you will need to use the Conduit Manager API, contained in `CondMgr.dll`, to register or unregister your conduit. The first hurdle you must clear during installation is finding the `CondMgr.dll` file.

Finding CondMgr.dll

One of the benefits of using a DLL to store the Conduit Manager functions is that if Palm Computing decides to change the way it stores conduit registration information, it can simply replace `CondMgr.dll` with a new version, and your installation program will be none the wiser. Unfortunately, Palm Computing does not install the DLL in the Windows system folder, where it would be easily accessible. Instead, `CondMgr.dll` is in the same directory as the HotSync Manager executable.

Finding the location of the HotSync Manager and all the core Palm Computing programs is actually very simple: Use the **CmGetCorePath** function from the Conduit Manager. However, this function, as you may have already guessed, is located in `CondMgr.dll` with the rest of the Conduit Manager API. This makes **CmGetCorePath** somewhat difficult to use, because the point of this whole exercise is to find the location of the `CondMgr.dll` file itself.

The way to solve this problem is to include a copy of `CondMgr.dll` with your installation program. That way, the installer can call **CmGetCorePath** from its local copy of `CondMgr.dll` to find out where the "real" `CondMgr.dll` is located, and then use the up-to-date copy to register the conduit. The sequence of events your installer should follow looks like this:

1. Look for `CondMgr.dll` on the system path and use that version if you find it, because it is probably from a more recent version of the Palm Desktop software than the version you are shipping with your conduit.
2. If `CondMgr.dll` is not on the path, use the **CmGetCorePath** function from the version bundled with your installer to find where the Palm Desktop is installed, and look in that directory for `CondMgr.dll`. Use this version if it exists, because it might be newer than the version included with your installer.
3. If you still cannot find `CondMgr.dll` in the Palm Desktop's directory, use the copy included with your installer. You may have to do this if the Palm Desktop software installed on the user's computer is older than version 3.0, because `CondMgr.dll` did not ship with earlier versions of the Palm Desktop.

Registering with the Conduit Manager

Once you have located `CondMgr.dll`, you can call its functions to register your conduit. There are two basic ways to accomplish this task:

- ✦ Build an installation structure and call a single function to register the conduit.
- ✦ Call many individual functions to set configuration entries for the conduit.

The first technique uses a `CmConduitType` structure with the **CmInstallConduit** function. The `CmConduitType` structure looks like this:

```
typedef struct {
    int iStructureVersion;
    int iStructureSize;
    int iType;
    char szCreatorID[CREATOR_ID_SIZE];
    DWORD dwPriority;
    int iConduitNameOffset;
    int iDirectoryOffset;
    int iFileOffset;
    int iRemoteDBOffset;
```

```
    int iUsernameOffset;
    int iTitleOffset;
    int iInfoOffset;
} CmConduitType;
```

All of the various `Offset` fields in this structure specify the offset in bytes from the beginning of the structure to the first character of a string value. This method of storing strings saves some space, because it does not need to store empty string values for those entries that you do not wish to include in your call to **CmInstall Conduit**. When you allocate space for a `CmConduitType` structure, make sure to include memory for the string values themselves.

The **CmInstallConduit** function takes a handle to a `CmConduitType` structure:

```
int CmInstallConduit (HANDLE hStruct)
```

The alternative, and much simpler, method of installing a conduit is to call individual functions to set the various registration entries. First, you need to call **CmInstall Creator** to set the creator ID that your conduit is registered to handle:

```
int CmInstallCreator (const char *pCreator, int iType)
```

For `iType`, specify the constant value `CONDUIT_APPLICATION`; `pCreator` should point to a string containing the creator ID to register.

After calling **CmInstallCreator**, you can call other functions, in any order you like, to set other registration entries:

- **CmSetCreatorName.** This function sets the file name of the conduit DLL, and it corresponds to the Conduit field in the Conduit Configuration tool.
- **CmSetCreatorDirectory.** This function sets the conduit's data directory.
- **CmSetCreatorFile.** This function sets the file name the conduit uses to store desktop data.
- **CmSetCreatorRemote.** This function sets the name of the database on the handheld that should be created if it does not already exist; it corresponds to the Remote Database field in the Conduit Configuration tool.
- **CmSetCreatorTitle.** This function sets the display name for the conduit; it corresponds to the Name field in the Conduit Configuration tool.
- **CmSetCreatorUser.** This function sets the user name for which this conduit was installed; it corresponds to the Username field in the Conduit Configuration tool.
- **CmSetCreatorPriority.** This function sets the conduit's priority.
- **CmSetCreatorInfo.** This function sets the conflict-resolution information string for the conduit; it corresponds to the Info field in the Conduit Configuration tool.

All of these functions take, as their first argument, a pointer to a string containing the creator ID that the conduit is registered to handle. For example, here is the prototype for the **CmSetCreatorName** function:

```
int CmSetCreatorName(const char *pCreatorID,
    const TCHAR *pConduitName)
```

Unregistering with the Conduit Manager

In the uninstall portion of your conduit's installation program, cleanly uninstall the conduit by calling the **CmRemoveConduitByCreatorID** function:

```
int CmRemoveConduitByCreatorID(const char *pCreatorID)
```

The **CmRemoveConduitByCreatorID** function removes all the conduits registered under the creator ID you supply and returns the number of conduits removed.

In the current implementation of CmRemoveConduitByCreatorID, the return value is always 1 if there is no error, because the HotSync Manager will allow only one conduit per creator ID.

Logging Actions in the HotSync Log

The HotSync Manager keeps a log of its actions, along with certain errors it might encounter, in a text file on the desktop computer, as well as a smaller version of the log accessible from the HotSync application on the handheld. Although the HotSync Manager does not actually write out the log file until after it has run all its conduits, the Sync Manager API provides a number of functions to allow conduits to add their own messages to the log. The HotSync Manager appends its own messages, and those added by conduits, to the end of the log file. To keep the HotSync log file from becoming too large, the HotSync Manager also trims the log file so that it only contains information about the ten most recent HotSync operations.

By default, the log file generated by the HotSync Manager is called HotSync.log, and it resides in the current user's directory on the desktop.

The simplest function for adding a message to the log is **LogAddEntry**, which has the following prototype:

```
long LogAddEntry (LPCTSTR pszEntry, Activity act,
    BOOL bTimeStamp)
```

The pszEntry parameter is a pointer to a null-terminated string containing the text that should be entered into the log. If bTimeStamp is TRUE, **LogAddEntry** appends a timestamp to the log entry. The act parameter specifies the kind of activity that you want to log, which must be a member of the Activity enumerated type. Table 17-1 contains descriptions of most of the activity constants.

Table 17-1
Activity Logging Constants

Constant	*Description*
`slArchiveFailed`	An archive operation failed.
`slCategoryDeleted`	A category was deleted.
`slChangeCatFailed`	Changing the category of a record failed.
`slDateChanged`	The date was changed.
`slDoubleModify`	A record was modified on both the desktop and the handheld.
`slDoubleModifyArchive`	A record that was modified on both desktop and the handheld was archived.
`slDoubleModifySubsc`	A file link record was modified on the desktop.
`slFileLinkCompleted`	File link processing has finished.
`slLocalAddFailed`	Adding a record to the desktop failed.
`slLocalSaveFailed`	Saving the desktop data file failed.
`slRecCountMismatch`	The number of records on the desktop and the number of records on the handheld do not match.
`slRemoteAddFailed`	Adding a record to the handheld failed.
`slRemoteChangeFailed`	Changing a record on the handheld failed.
`slRemoteDeleteFailed`	Deleting a record from the handheld failed.
`slRemotePurgeFailed`	Purging a record from the handheld failed.
`slRemoteReadFailed`	Reading a record on the handheld failed.
`slResetFlagsFailed`	Resetting synchronization flags failed.
`slSyncAborted`	Synchronization was aborted.
`slSyncFinished`	The synchronization operation completed successfully.
`slSyncStarted`	The synchronization operation started.
`slText`	Indicates a simple text entry.
`slTooManyCategories`	The maximum number of categories has already been reached.
`slWarning`	Logs a warning.
`slXMapFailed`	The position cross-map function failed.

Most of the time, you will need to use only five of the activity logging constants:

- `slSyncStarted`. Call **LogAddEntry** with this constant and an empty string when your conduit first starts up:

  ```
  LogAddEntry("", slSyncStarted, FALSE);
  ```

- `slSyncAborted`. Use this constant if your conduit encounters an error that forces it to quit synchronization:

  ```
  LogAddEntry("MyConduit", slSyncAborted, FALSE);
  ```

- `slSyncFinished`. This constant signals to the log that your application successfully completed its synchronization:

  ```
  LogAddEntry("MyConduit", slSyncFinished, FALSE);
  ```

- `slWarning`. When this activity constant is specified, the HotSync Manager displays a dialog box at the end of the HotSync operation to alert the user that there are messages of interest in the log. This constant should be used to warn the user about errors that did not cause the conduit to abort.

- `slText`. Use this constant to add a simple message to the log without alerting the user at the end of the HotSync process. This constant is a good choice to use if you want to quietly add diagnostic information to the HotSync log that you could later use in resolving possible synchronization bugs in your conduit. These messages do not require immediate attention from the user, but if the user has problems with your conduit, you can look through the user's HotSync log and find out exactly what your conduit was doing when the problem occurred.

Besides **LogAddEntry**, there is a **LogAddFormattedEntry** function; it allows you to use standard C `sprintf` format specifiers to format a log entry. The **LogAdd FormattedEntry** function has the following prototype:

```
long LogAddFormattedEntry (Activity act, BOOL bTimeStamp,
    const char* dataString, ...);
```

Use the `dataString` parameter the same way you would use the format specifier for a `sprintf` call:

```
LogAddFormattedEntry(slWarning, false,
    "%d terrible things happened during synchronization.",
    nTerrible);
```

You can also find out how many errors were logged using **LogTestCounters**, which has the following prototype:

```
WORD LogTestCounters()
```

This function simply returns the number of error entries in the log, or `0` if the log contains no errors. The following activity constants do not count as errors for the purposes of the **LogTestCounters** function:

- `slSyncAborted`
- `slSyncFinished`
- `slSyncStarted`
- `slText`

The **LogTestCounters** function treats every other type of log entry as an error and adds it to its return value.

Internally, this function is how the HotSync Manager determines whether or not to display a dialog box to the user mentioning the presence of messages in the HotSync log. Although explicit use of the `slWarning` constant will trigger the dialog box, any of the other entry types that LogTestCounters counts will also cause the dialog box to be displayed.

Summary

In this chapter, you learned some of the basics of how conduits work, as well as the mechanical aspects of installing conduits and logging their actions to the HotSync log. After reading this chapter, you should understand the following:

- A conduit is a code module that the HotSync Manager calls to synchronize data between a handheld application and a data source on the desktop.
- The four types of synchronization a conduit can perform, from most complex to least, are transaction-based, mirror image, one-directional, and backup.
- There are three basic paths to developing a conduit: starting with the Palm MFC Base Classes, starting with the Palm Generic Conduit Base Classes, or calling Sync Manager API functions directly.
- During development, you can manually install conduits using the Conduit Configuration tool, and you can use the Conduit Switch tool to save backups of your configurations and restore them later on.
- To provide for smooth installation of a conduit, your conduit's install program needs to register the conduit using the Conduit Manager API.
- You can add entries to the HotSync log with the **LogAddEntry** and **LogAdd FormattedEntry** functions.

✦ ✦ ✦

CHAPTER 18

Building Conduits

In This Chapter

Using the Conduit Wizard

Implementing conduit entry points

Using the Palm MFC base classes

Using the Palm generic conduit base classes

Using the Sync Manager API

No matter which set of base classes you decide to use for your conduit, or even if you decide to forgo using the base classes at all and use the Sync Manager API directly, the Conduit Development Kit for Windows provides one easy way to start any conduit project: the Conduit Wizard. The Conduit Wizard is installed in Visual C++ when you install the rest of the CDK for Windows.

You do not have to use the Conduit Wizard to build a conduit in Visual C++, but it certainly saves you a lot of time and frustration to let the wizard generate boilerplate code for you to fill in. If you really need to start from scratch (or if pain is something you enjoy), you can build a conduit DLL without the wizard, but be sure to build it as a regular DLL, not as an extension DLL.

Using the Conduit Wizard

To create a new conduit project in Visual C++ using the Conduit Wizard, select File➪New. The New dialog box, pictured in Figure 18-1, appears.

Select the Projects tab in the dialog box, then select "Palm Conduit Wizard (dll)" from the list of available project types. Enter a name for your project in the Project name text box, then enter an appropriate path for the project in the Location text box. Click the OK button when you have set everything to your satisfaction.

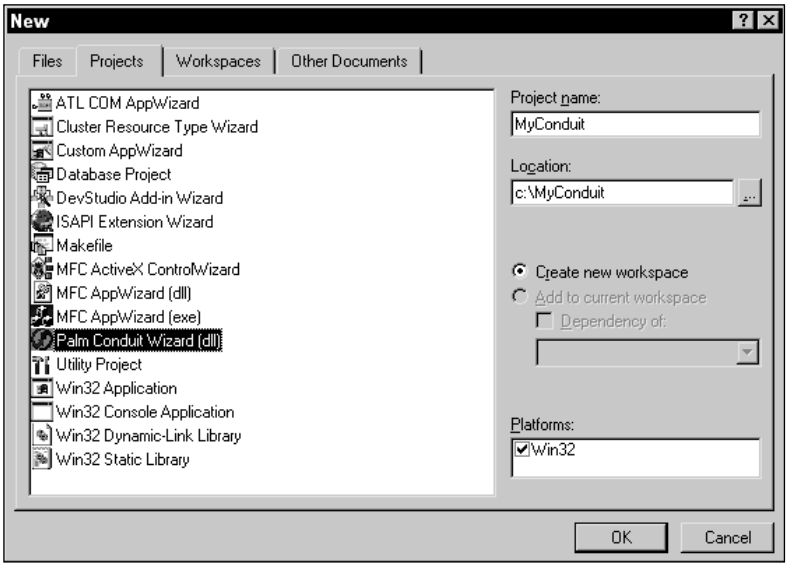


Figure 18-1: The Visual C++ New dialog box, from which you can create a new conduit project using the Conduit Wizard

After you click OK, the Conduit Wizard launches and begins to prompt you for parameters that control what kind of conduit project you want to create. The Conduit Wizard has five steps:

1. Select the type of conduit.
2. Choose a handheld application with which to synchronize.
3. Select a type of data transfer.
4. Select conduit features.
5. Confirm the classes to be created by the Conduit Wizard.

If you choose to create only conduit entry points in Step 1 instead of using the base classes, the Conduit Wizard skips Steps 2 and 3 and proceeds straight to feature selection in Step 4. However, because of a glitch in the Conduit Wizard, it still displays “Step 2 of 5” and “Step 3 of 5” in its title bar for Steps 4 and 5, respectively. This little interface eccentricity will not have any affect on the template code generated by the wizard, but it is a little confusing.

Selecting a Conduit Type

Step 1 in the Conduit Wizard prompts you for the set of base classes you want to use for building the conduit. Figure 18-2 shows the Conduit Wizard at this stage of the process.

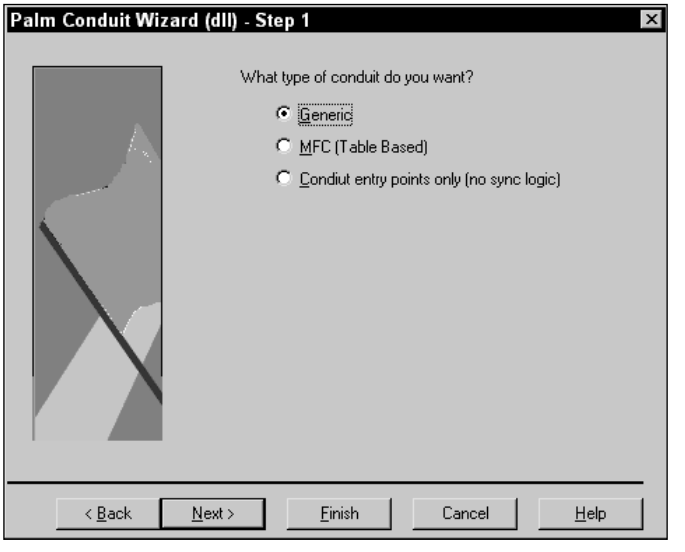


Figure 18-2: Conduit Wizard Step 1: conduit type selection

The Conduit Wizard presents you with three choices:

- **Generic.** Choose Generic if you want to build a conduit using the Palm Generic Conduit Base Classes.
- **MFC (Table Based).** Select this option if you want to build a conduit using the Palm MFC Base Classes.
- **Conduit entry points only (no sync logic).** Pick this option if you do not want to use either set of base classes but instead merely wish to use the Conduit Manager to generate a framework for a Sync Manager API-based conduit.

Click Next to proceed to the next step or Back to return to the New dialog box.

Choosing a Handheld Application

From Step 2, pictured in Figure 18-3, you can choose with which handheld application the conduit interfaces.

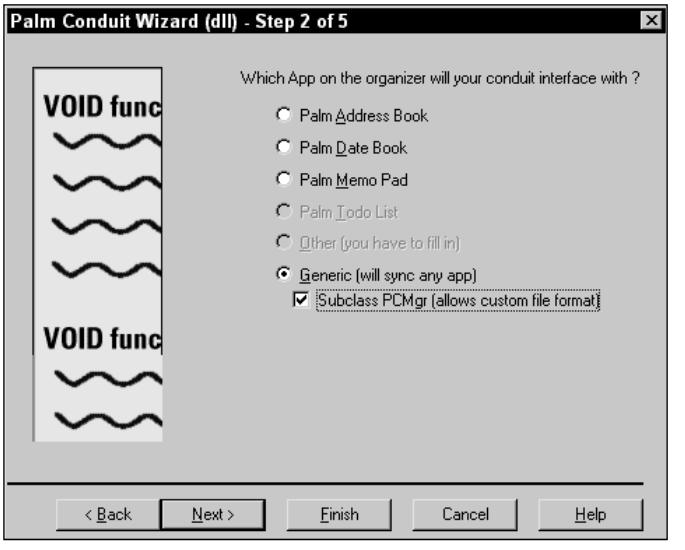


Figure 18-3: Conduit Wizard Step 2: handheld application selection

The Conduit Wizard can generate code for connecting to any of the four main Palm OS ROM applications, or it can assemble a framework for you to fill in for synchronization with your own custom handheld applications. The options presented by the handheld application selection screen are as follows:

- **Palm Address Book.** Synchronize with the Address Book application.
- **Palm Date Book.** Synchronize with the Date Book application.
- **Palm Memo Pad.** Synchronize with the Memo Pad application.
- **Palm Todo List.** Synchronize with the To Do List application. This option is not available if you selected Generic in Step 1.
- **Other (you have to fill in).** Synchronize with a custom application. This option is for MFC conduits only and is not available if you selected Generic in Step 1.
- **Generic (will sync any app).** Synchronize with a custom application. This option is for generic conduits only and is not available if you selected MFC (Table Based) in Step 1. In addition, when you select this option you may also select the Subclass PCMgr (allows custom file format) check box, which allows you to implement your own storage format on the desktop. Without this box checked, the code generated by the Conduit Wizard saves to the same MFC serialized format as an MFC conduit, so checking the box is important if you want to synchronize with a different data source on the desktop.

Click Next to proceed to the next step or Back to return to Step 1.

Selecting a Data Transfer Type

Step 3 in the Conduit Wizard, shown in Figure 18-4, allows you to choose what type of data transfer you want in your conduit.

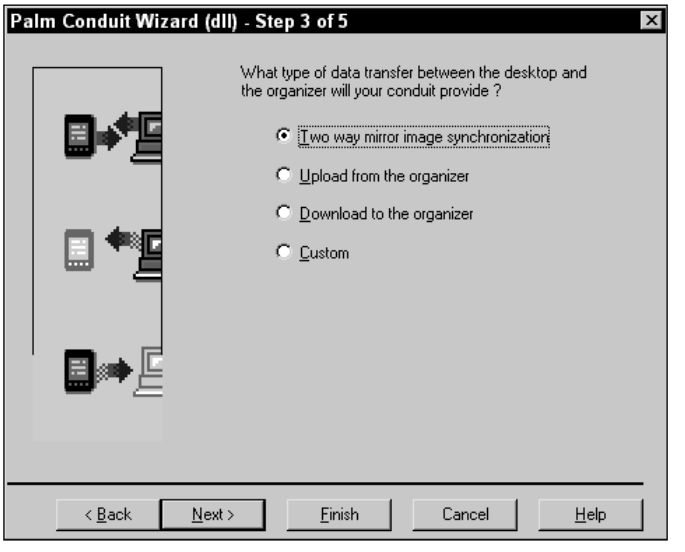


Figure 18-4: Conduit Wizard Step 3: data transfer selection

The choices available in this step are described below:

- **Two-way mirror image synchronization.** If you select this option, the Conduit Wizard creates code to implement mirror image synchronization between the desktop and handheld, as described in Chapter 17, "Introducing Conduit Mechanics." This is by far the easiest way to add mirror image synchronization to a conduit, because the base classes can take care of all the ugly details of resolving record conflicts without your having to write a single line of code.
- **Upload from the organizer.** With this option selected, the code produced by the Conduit Wizard is geared toward retrieving information from the handheld and storing it on the desktop, probably performing some kind of operation on the data in the process.

If you are interested only in backing up data from the handheld application, rely on the standard Backup conduit instead of building a conduit with the Upload from the organizer data transfer option. You need to use this option only if you want to convert the data to another format before storing it on the desktop. For example, if you want your handheld application's data to be stored in a Microsoft Access database on the desktop computer, you should use the Upload from the organizer option.

- **Download to the organizer.** This option creates application code that moves data from the desktop computer to the handheld. If your data may be modified only on the desktop computer and then transferred to the handheld, this is a good option to select.
- **Custom.** Select this option if the other options listed above do not suit your application.

Click Next to proceed to the next step or Back to return to Step 2.

Selecting Conduit Features

Step 4 in the Conduit Wizard, pictured in Figure 18-5, allows you to add optional features to your conduit.

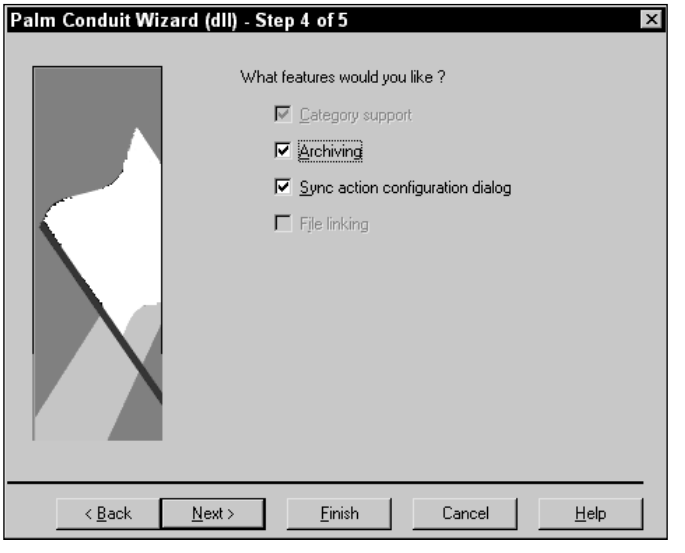


Figure 18-5: Conduit Wizard Step 4: feature selection

Depending on your selections in Steps 1 and 2, some of the features listed in Step 4 may not be available, in which case they will be grayed out in the dialog box. Others may be required by the conduit type and application, and they will be grayed out but checked, to indicate that you cannot tell the Conduit Wizard to skip making code for that feature. The features are:

- **Category support.** If this feature is selected, the Conduit Wizard generates code to synchronize the standard system of record categories used by many Palm OS applications.

- **Archiving.** Selecting Archiving causes the Conduit Wizard to add support for archiving deleted and modified records to a separate file from the normal desktop data source.
- **Sync action configuration dialog box.** Select this option to have the wizard add code implementing a standard conduit configuration dialog box, pictured in Figure 18-6. This dialog box appears when the user chooses the Custom option in the HotSync Manager's menu, and it allows the user to change how the conduit synchronizes its data.

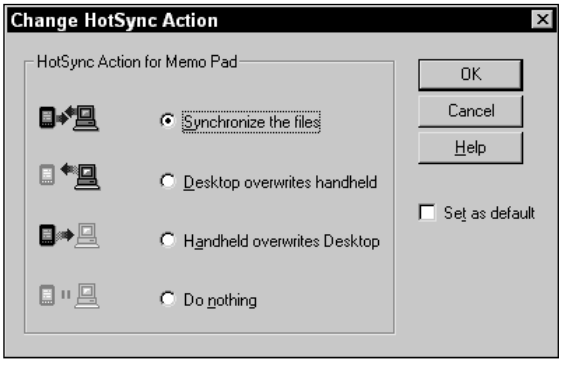


Figure 18-6: The standard Change HotSync Action dialog box, which allows the user to modify a conduit's actions during a HotSync operation

- **File linking.** If this option is selected, the Conduit Wizard adds functions to support *file linking* into your conduit project. File linking is a way to automatically update handheld data from a file on the desktop. For example, you could link a handheld application to an address database kept on the desktop; changes made to the desktop database would then be made automatically to the handheld application's data during every HotSync operation. A file link updates data in only one category of the handheld database.

Click Next to proceed to the next step or Back to return to Step 3.

Confirming Class and File Names

The final step in the Conduit Wizard, shown in Figure 18-7, allows you to customize the names of classes and files generated by the wizard.

Figure 18-7: Conduit Wizard Step 5: class and file name confirmation

By default, the Conduit Wizard creates class and file names based on the project name you supplied when you first started the wizard. Sometimes, these names can be somewhat unwieldy to use when actually programming the conduit, so this step gives you an opportunity to change the names to something more aesthetically pleasing or mnemonic.

To change a class, header file, or implementation file name, select a class name from the list at the top of the dialog box. Modify the names in the Class name, Header file, and Implementation file text boxes. When you have altered the names to your taste, click Finish to proceed to the New Project Information dialog box, shown in Figure 18-8. You may also click Back to return to Step 4.

The New Project Information dialog box summarizes the classes and files that the Conduit Wizard will create, and it also lists the options you select for the project. Verify that everything is set up the way you want it, then click OK to actually create the new conduit project. If you find that you would like to change something, click Cancel, and Visual Studio will return you to the Conduit Wizard so you can make changes to the project. You will not lose the changes you have already made if you click Cancel.

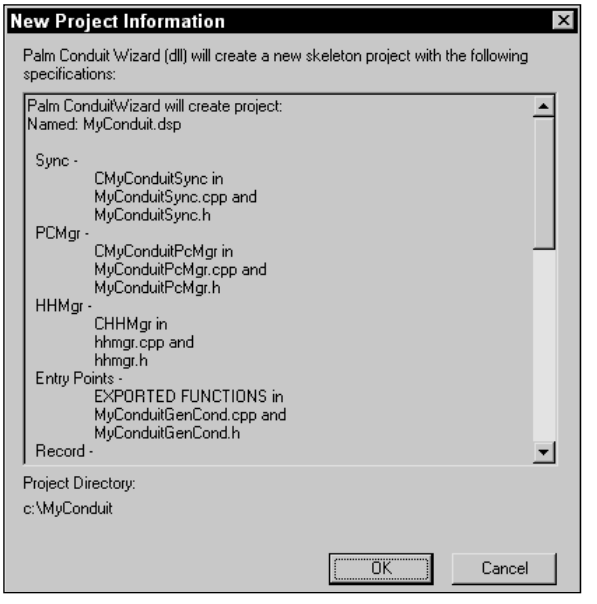


Figure 18-8: The New Project Information dialog box

Implementing Conduit Entry Points

Now that you have created an application framework with the Conduit Wizard, you can get to the actual work of coding your conduit. Whether you use the base classes or not, every conduit has certain entry points that you must implement, and several that are optional if you want to add other features to your conduit. The following four functions are required entry points to any conduit:

- **GetConduitInfo.** This function returns various bits of information about the conduit, including its name, whether MFC is used to build the conduit, and the conduit's default action.
- **GetConduitName.** This function returns the name of the conduit.
- **GetConduitVersion.** This function returns the version number of the conduit.
- **OpenConduit.** This function is the main entry point into the conduit. When the HotSync Manager needs to call a conduit to perform synchronization, it calls the conduit's **OpenConduit** function.

In addition to the required entry points, you will need to implement additional functions to provide certain features in your conduit:

- **ConfigureConduit** and **CfgConduit.** These two functions serve an identical purpose, which is to display a configuration dialog box to allow the user to customize how the conduit synchronizes its data. The **CfgConduit** function is a newer version of the **ConfigureConduit** function that provides more data to your conduit; **CfgConduit** is available in version 3.0 and later of the HotSync Manager.
- **ConfigureSubscription.** This function allows the HotSync Manager to retrieve file-linking details from your conduit.
- **ImportData.** This function loads data from a linked file and displays a dialog box allowing the user to choose how fields should be mapped between the linked file and the data source.
- **SubscriptionSupported.** The HotSync Manager calls this function to determine if a conduit supports file linking.
- **UpdateTables.** This function updates a desktop data source with information from a file-linking operation, such as changes to category names.

External entry points to a conduit DLL, just like any DLL entry point built with Visual C++, must have a return type of `__declspec( dllexport )`. One of the header files created by the Conduit Wizard provides a bit of syntactic sugar for this unwieldy expression:

```
#define ExportFunc __declspec( dllexport )
```

All of the entry point functions generated by the Conduit Wizard are declared with an `ExportFunc` return type.

Implementing GetConduitInfo

The Conduit Wizard generates all of the code you will normally need for the **GetConduitInfo**, **GetConduitName**, and **GetConduitVersion** functions. For the most part, you should not have to customize these functions. Instead, you should customize some of the strings in the string table created by the Conduit Wizard. Click the ResourceView tab in the Visual C++ workspace toolbar, then find the String Table resource and double-click it to view the strings in the conduit project. Figure 18-9 shows the string table open for editing.

The two strings you should modify are `IDS_CONDUIT_NAME`, at the top of the list, and `IDS_SYNC_ACTION_TEXT`, at the bottom of the list. The `IDS_CONDUIT_NAME` string is simply the name of your conduit, as it will appear in the HotSync Manager's Custom dialog box (pictured in Figure 18-10) and in the HotSync log. Typically, this will be the same as the name of the handheld application with which the conduit synchronizes.

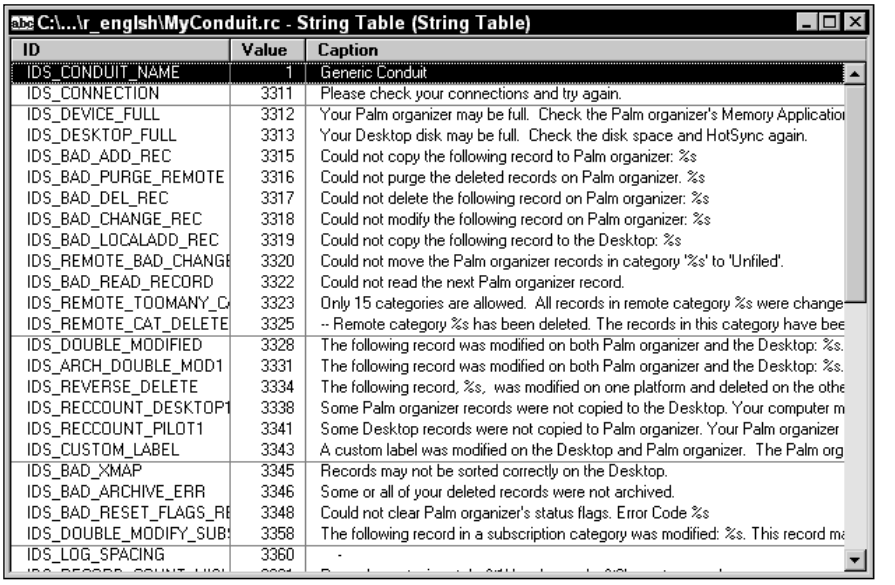


Figure 18-9: The string table in a typical Conduit Wizard–generated project

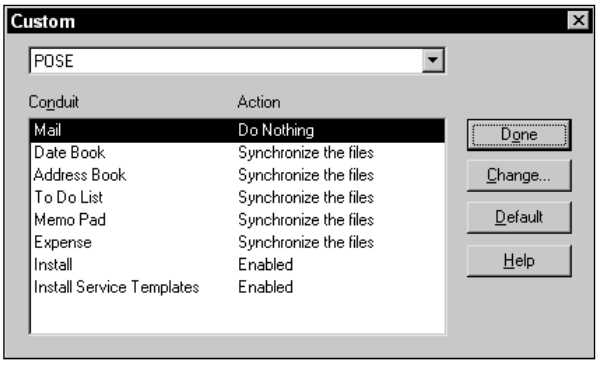


Figure 18-10: The HotSync Manager's Custom dialog box

The `IDS_SYNC_ACTION_TEXT` string appears in the default configuration dialog box (see Figure 18-6) that the Conduit Wizard provides if you select the **Sync action configuration** dialog box feature during Step 4 of the Conduit Wizard. You should set this string to something like `HotSync Action for` *`MyApp`*, where *`MyApp`* is the name of your application.

Once you have the string table set up properly, you need only to alter **GetConduitInfo**, **GetConduitName**, and **GetConduitVersion** if you want to make them return information other than just information from the string table. Here is the wizard-generated **GetConduitInfo** function:

```
ExportFunc long GetConduitInfo (ConduitInfoEnum infoType,
                                void *pInArgs, void *pOut,
                                DWORD *pdwOutSize)
{
    if (!pOut)
        return CONDERR_INVALID_PTR;
    if (!pdwOutSize)
        return CONDERR_INVALID_OUTSIZE_PTR;

    switch (infoType) {
        case eConduitName:
            if (!pInArgs)
                return CONDERR_INVALID_INARGS_PTR;
            ConduitRequestInfoType *pInfo;
            pInfo = (ConduitRequestInfoType *)pInArgs;
            if ((pInfo->dwVersion !=
                 CONDUITREQUESTINFO_VERSION_1) ||
                (pInfo->dwSize != SZ_CONDUITREQUESTINFO))
                return CONDERR_INVALID_INARGS_STRUCT;

            if (!::LoadString((HINSTANCE)hLangInstance,
                              IDS_CONDUIT_NAME,
                              (TCHAR*)pOut, *pdwOutSize))
                return CONDERR_CONDUIT_RESOURCE_FAILURE;
            break;

        case eDefaultAction:
            if (*pdwOutSize != sizeof(eSyncTypes))
                return CONDERR_INVALID_BUFFER_SIZE;
            (*(eSyncTypes*)pOut) = eFast;
            break;

        case eMfcVersion:
            if (*pdwOutSize != sizeof(DWORD))
                return CONDERR_INVALID_BUFFER_SIZE;
            (*(DWORD*)pOut) = MFC_NOT_USED;
            break;

        default:
            return CONDERR_UNSUPPORTED_CONDUITINFO_ENUM;
    }
    return 0;
}
```

The **GetConduitInfo** function must deal with three possible requests for information:

- `eConduitName` — a request for the name of the conduit

- ✦ `eDefaultAction` — a request for the conduit's default action, expressed as a member of the `eSyncTypes` enumerated type
- ✦ `eMfcVersion` — a request for the version of MFC used to build the conduit

Boilerplate code generated by the Conduit Wizard for **GetConduitInfo** returns the `IDS_CONDUIT_NAME` string when **GetConduitInfo** receives an `eConduitName` request, and it returns `eFast` as the default action for the conduit. The `eSyncTypes` enumerated type, defined in the CDK header file `syncmgr.h`, looks like this:

```
enum eSyncTypes {eFast,
                 eSlow,
                 eHHtoPC,
                 ePCtoHH,
                 eInstall,
                 eBackup,
                 eDoNothing,
                 eProfileInstall,
                 eSyncTypeDoNotUse=0xffffffff
};
```

If your application should perform a default action other than a FastSync, simply replace the `eFast` constant in **GetConduitInfo** with the appropriate `eSyncTypes` constant. For example, if your conduit only backs up a handheld application's data to a database on the desktop computer, you should choose `eBackup`.

If you used the Conduit Wizard to generate an MFC conduit, the **GetConduitInfo** function will return different values in response to an `eMfcVersion` request than if you used the Conduit Wizard to create a conduit based on the Palm Generic Conduit Classes. These values are defined as follows in the CDK header file `Condapi.h`:

```
#define MFC_VERSION_41  0x00000410
#define MFC_VERSION_50  0x00000500
#define MFC_VERSION_60  0x00000600
#define MFC_NOT_USED    0x10000000
```

The versions in these constants correspond to the version of Visual C++ that MFC shipped with, not the version number included in the MFC libraries themselves.

Implementing GetConduitName

The default **GetConduitName** function provided by the Conduit Wizard merely returns the `IDS_CONDUIT_NAME` string:

```
ExportFunc long GetConduitName (char* pszName, WORD nLen)
{
    long retval = -1;
```

```
    if (::LoadString((HINSTANCE)hLangInstance,
                     IDS_CONDUIT_NAME, pszName, nLen))
        retval = 0;

    return retval;
}
```

Implementing GetConduitVersion

The **GetConduitVersion** function is even simpler than **GetConduitName**, returning a constant from one of the conduit project's header files; the following code from a wizard-generated generic conduit implements **GetConduitVersion**:

```
ExportFunc DWORD GetConduitVersion()
{
    return GENERIC_CONDUIT_VERSION;
}
```

Within the `DWORD` value you return from **GetConduitVersion**, you must pack the major version number in the high byte of the return value's low word, and the minor version number goes into the low byte of the low word. For example, a standard generic conduit created by the Conduit Wizard defines the following constant, equivalent to a version number of `1.2`:

```
#define GENERIC_CONDUIT_VERSION 0x00000102
```

Never use a conduit version of `0x00000100` (`1.0`). Although it seems intuitive for a new conduit to have a version of `1.0`, there is a bug in version 2.*x* of the Palm Desktop software that prevents any conduit from running with a version of `1.0`. This affects all Macintosh Palm Desktop users. The default version constant provided by the Conduit Wizard is `0x00000101`, which you should leave alone until you need to increment your conduit's version number.

Implementing OpenConduit

The **OpenConduit** function is responsible for actually performing synchronization between a specific handheld application and its desktop data source. If you used the Conduit Wizard to create a conduit framework based on the generic or MFC base classes, **OpenConduit** will already contain some code that calls base class methods to perform the synchronization. If you used the Conduit Wizard only to create entry points, the **OpenConduit** function is rather sparse, providing you with only a placeholder to fill in with your own code:

```
ExportFunc long OpenConduit (PROGRESSFN pFn,
                             CSyncProperties& rProps)
{
    long retval = -1;
    if (pFn)
```

```
    {
        // TODO - create your own custom sync class, and run it
    }
    return(retval);
}
```

The `pFn` argument to **OpenConduit** is a pointer to a callback function provided by the HotSync Manager, which allows you to update the HotSync Manager with status messages regarding the progress of your conduit.

None of the conduits for the four built-in applications actually makes use of the callback, and documentation regarding it is scant at best. The MFC boilerplate generated by the Conduit Wizard assigns the function pointer to the `m_pfnProgress` member of the conduit's subclass of CBaseConduitMonitor, but it never actually calls the function. If you feel like experimenting, the following comment from the `basemon.cpp` source file, which implements the MFC CBaseConduitMonitor class, might provide a starting place if you want to explore further:

```
// The Progress Callback is not currently used, but here
// is an example of how you could use it:
//     char probString[64];
//     memset(probString,0,sizeof(probString));
//     strcpy(probString, "Joe");
//     (*m_pfnProgress)(probString);
```

The HotSync Manager fills the `rProps` argument to **OpenConduit** with a pointer to a **CSyncProperties** object, which contains a wealth of information about the environment in which the conduit is running. The CDK header file `Syncmgr.h` defines **CSyncProperties** as follows:

```
class CSyncProperties
{
public:
    eSyncTypes m_SyncType;
    char       m_PathName[BIG_PATH];
    char       m_LocalName[BIG_PATH];
    char       m_UserName[BIG_PATH];
    char*      m_RemoteName[SYNC_DB_NAMELEN];
    CDbListPtr *m_RemoteDbList;
    int        m_nRemoteCount;
    DWORD      m_Creator;
    WORD       m_CardNo;
    DWORD      m_DbType;
    DWORD      m_AppInfoSize;
    DWORD      m_SortInfoSize;
    eFirstSync m_FirstDevice;
    eConnType  m_Connection;
    char       m_Registry[BIG_PATH];
    HKEY       m_hKey;
    DWORD      m_dwReserved;
};
```

Table 18-1 describes the data members of the **CSyncProperties** class.

Table 18-1
CSyncProperties Data Members

Member	*Description*
`m_SyncType`	Current synchronization type.
`m_PathName`	Pathname that precedes files on the desktop computer.
`m_LocalName`	File name of the data source on the desktop.
`m_UserName`	User name used for this HotSync operation.
`m_RemoteName`	An array of names of databases on the handheld to synchronize with.
`m_RemoteDbList`	Pointer to a CDbList class that contains further information about the remote databases, including various status flags and modification dates.
`m_nRemoteCount`	Number of databases on the handheld that should be synchronized
`m_Creator`	Creator ID of the handheld databases.
`m_CardNo`	Card number containing the handheld databases.
`m_DbType`	Database type for databases on the handheld.
`m_AppInfoSize`	Size of the handheld database's application info block, stored here for convenience.
`m_SortInfoSize`	Size of the handheld database's sort info block, stored here for convenience.
`m_FirstDevice`	Specifies if this is the first time this particular desktop or handheld has been synchronized, using the `eFirstSync` enumerated type, which has the following possible values: `eNeither`, `ePC`, and `eHH`.
`m_Connection`	Connection medium used for this HotSync operation. This value comes from the `eConnType` enumerated type, which has the following possible values: `eCable` and `eModemConnType`.
`m_Registry`	Full Windows Registry path for the conduit.
`m_hKey`	Primary Windows Registry key for the conduit.
`m_dwReserved`	Reserved for future use. Set this field to `NULL` before using a CSyncProperties object.

Within the **OpenConduit** function, you should create a new instance of whatever class you are using to implement synchronization logic, call an appropriate method of that class to begin the actual synchronization process, then delete the object. If you are working from a bare bones conduit with only the entry points created by the Conduit Wizard, you have your work cut out for you. If you are using either the generic or MFC base classes, the Conduit Wizard will have already filled in the appropriate code for the **OpenConduit** function. In a generic conduit, **OpenConduit** looks like this:

```
ExportFunc long OpenConduit (PROGRESSFN pFn,
                             CSyncProperties& rProps)
{
    long retval = -1;
    if (pFn)
    {
        CLibCondSync* pGeneric;
        pGeneric = new CLibCondSync(rProps,
            GENERIC_FLAG_CATEGORY_SUPPORTED |
            GENERIC_FLAG_APPINFO_SUPPORTED );
        if (pGeneric){
            retval = pGeneric->Perform();

            delete pGeneric;
        }
    }
    return(retval);
}
```

An MFC-based conduit has the following **OpenConduit** implementation:

```
ExportFunc long OpenConduit (PROGRESSFN pFn,
                             CSyncProperties& rProps)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    long retval = -1;
    if (pFn)
    {
        CLibCondMonitor* pMonitor;

        pMonitor = new CLibCondMonitor(pFn, rProps, myInst);
        if (pMonitor)
        {
            retval = pMonitor->Engage();

            delete pMonitor;
        }
    }
    return(retval);
}
```

Calling the AFX_MANAGE_STATE macro at the start of each exported function in an MFC DLL is required for the DLL to function properly. The Conduit Wizard puts the macro in the proper places when it generates a framework for your conduit, and you should not delete any occurrence of AFX_MANAGE_STATE.

The "Using Palm MFC base classes" and "Using Generic Conduit base classes" sections later in this chapter provide more details about customizing the base classes so that they will perform appropriate actions for your own conduits.

Implementing Configuration Entry Points

The **ConfigureConduit** and **CfgConduit** entry points allow the user to change how the conduit synchronizes its data. You can think of the dialog box displayed by these functions as a control panel from which the user can not only select synchronization types (such as choosing between "Synchronize the files" and "Do nothing") but also change any other settings that might be specific to your own conduit (like setting paths to other files that your conduit might need on the desktop).

Although doing so is not strictly required, you should implement ConfigureConduit or CfgConduit in every conduit, because without one of these functions, nothing happens when the user tries to configure the conduit from the HotSync Manager's Custom dialog box (see Figure 18-10). At the very least, you should display a dialog box explaining that there is nothing to configure in your conduit, or perhaps just display an About box.

The **CfgConduit** function is a newer version of **ConfigureConduit**, which Palm Computing added with the introduction of version 3.0 of the HotSync Manager. When the user chooses to change a conduit's HotSync action, 3.0 and later versions of the HotSync Manager attempt to call **CfgConduit** first; if that function is not implemented, the HotSync Manager tries to call **ConfigureConduit**. Earlier versions of the HotSync Manager call only **ConfigureConduit**.

If you used the Conduit Wizard to generate a generic conduit, or only conduit entry points, the boilerplate code already contains a default dialog box resource (`IDD_CONDUIT_ACTION`) and the code to implement it in **CfgConduit** and **ConfigureConduit**. Figure 18-11 shows `IDD_CONDUIT_ACTION` as it appears in the Visual C++ resource editor.

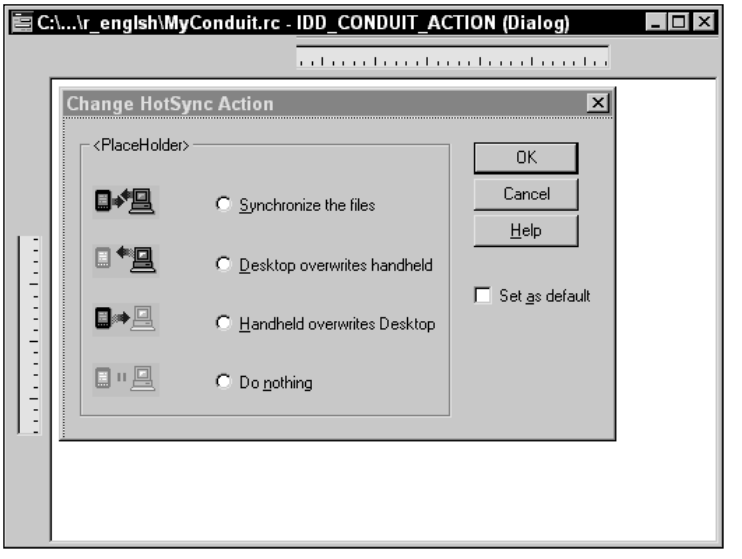


Figure 18-11: The IDD_CONDUIT_ACTION resource generated by the Conduit Wizard

The `IDD_CONDUIT_ACTION` dialog box is a good starting point if you want to offer the user more, or possibly fewer, configuration options for your conduit. All you need to do is modify the resource, then add code to **CfgConduit** and **ConfigureConduit** to implement user interaction with the new dialog box. The base code for the wizard-generated **ConfigureConduit** function in generic and bare bones conduits looks like this:

```
ExportFunc long ConfigureConduit (CSyncPreference& pref)
{

    long nRtn = -1;
    CfgConduitInfoType cfg;
    cfg.dwVersion = CFGCONDUITINFO_VERSION_1;
    cfg.dwSize  = sizeof(CfgConduitInfoType);
    cfg.dwCreatorId = 0;
    cfg.dwUserId = 0;
    memset(cfg.szUser , 0, sizeof(cfg.szUser));
    memset(cfg.m_PathName, 0, sizeof(cfg.m_PathName));
    cfg.syncPermanent = pref.m_SyncType;
    cfg.syncTemporary = pref.m_SyncType;
    cfg.syncNew = pref.m_SyncType;
    cfg.syncPref = eTemporaryPreference;
```

```
    int iResult;
    iResult = DialogBoxParam((HINSTANCE)hLangInstance,
              MAKEINTRESOURCE(IDD_CONDUIT_ACTION),
              GetForegroundWindow(),
              (DLGPROC)ConfigureDlgProc,
              (LPARAM)&cfg);
    if (iResult == 0) {
        pref.m_SyncType = cfg.syncNew;
        pref.m_SyncPref = cfg.syncPref;
        nRtn = 0;
    }
    return nRtn;
}
```

A wizard-generated MFC conduit project also contains fully implemented **CfgConduit** and **ConfigureConduit** functions. Instead of a resource that is included as part of the project, an MFC conduit created by the Conduit Wizard uses a **CHotSyncActionDlg** object, which is a descendant of the MFC **CDialog** class. The resulting dialog is identical to that produced by the generic conduit code, but changing the dialog requires that you subclass **CHotSyncActionDlg** and make modifications there instead of altering a dialog resource. The **ConfigureConduit** function in an MFC conduit project looks like this:

```
ExportFunc long ConfigureConduit (CSyncPreference& pref)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    long nRtn = -1;
    char szName[81];
    CHotSyncActionDlg actDlg(CWnd::GetActiveWindow());

    pref.m_SyncPref = eNoPreference;

    GetConduitName(szName,80);
    actDlg.m_csGroupText = szName;

    switch (pref.m_SyncType) {
        case eFast:
        case eSlow:
            actDlg.m_nActionIndex = 0;
            break;
        case ePCtoHH:
            actDlg.m_nActionIndex = 1;
            break;
        case eHHtoPC:
            actDlg.m_nActionIndex = 2;
```

```
            break;
        case eDoNothing:
        default:
            actDlg.m_nActionIndex = 3;
    }

    if (actDlg.DoModal() == IDOK) {
        switch (actDlg.m_nActionIndex) {
            case 0:
                pref.m_SyncType = eFast;
                break;
            case 1:
                pref.m_SyncType = ePCtoHH;
                break;
            case 2:
                pref.m_SyncType = eHHtoPC;
                break;
            case 3:
            default:
                pref.m_SyncType = eDoNothing;
                break;
        }

        pref.m_SyncPref = (actDlg.m_bMakeDefault) ?
            ePermanentPreference : eTemporaryPreference;

        nRtn = 0;
    }
    return nRtn;
}
```

The `pref` argument that the HotSync Manager passes to **ConfigureConduit** is a **CSyncPreference** object containing information about the current HotSync settings for the conduit. Defined in the CDK header file `Syncmgr.h`, this is what the **CSyncPreference** class looks like:

```
class CSyncPreference
{
public:
    char       m_PathName[BIG_PATH];
    char       m_Registry[BIG_PATH];
    HKEY       m_hKey;
    eSyncPref  m_SyncPref;
    eSyncTypes m_SyncType;
    DWORD      m_dwReserved;
};
```

Table 18-2 describes what the data members of the **CSyncPreference** class mean.

Table 18-2
CSyncPreference Data Members

Member	*Description*
`m_PathName`	Indicates the pathname that precedes files on the desktop computer.
`m_Registry`	Specifies the full Windows Registry path for the conduit.
`m_hKey`	Indicates the primary Windows Registry key for the conduit.
`m_SyncPref`	Specifies whether the user's selected synchronization preferences should be applied temporarily (on the next HotSync action only) or permanently. This field uses the `eSyncPref` enumerated type, which can have the following values: `eNoPreference`, `ePermanentPreference`, or `eTemporaryPreference`.
`m_SyncType`	Specifies the synchronization type using a constant from the `eSyncTypes` enumerated type.
`m_dwReserved`	Reserved for future use. Set this field to `NULL` before using a CSyncPreference object.

Not only does the **CSyncPreference** object serve to provide the **ConfigureConduit** function with information it needs, **CSyncPreference** also returns values to the HotSync Manager in response to changes the user makes to the conduit's settings. In particular, the `m_SyncPref` and `m_SyncType` members of **CSyncPreference** should be set to appropriate values before returning from **ConfigureConduit**.

The **CfgConduit** function allows later versions of the HotSync Manager to pass more data to the conduit than can be done through the **ConfigureConduit** function. Instead of passing data through a **CSyncPreference** object, **CfgConduit** accepts a `CfgConduitInfoType` structure. Here is the prototype for **CfgConduit**:

```
long CfgConduit (ConduitCfgEnum cfgType, void *pArgs,
                 DWORD *pdwArgsSize);
```

The first argument to **CfgConduit** indicates what version of the **CfgConduit** function the HotSync Manager is attempting to call, as defined by the `ConduitCfgEnum` enumerated type. As of this writing, there is only one version of the **CfgConduit** function, which you should specify using the `eConfig1` constant.

The `pArgs` argument contains a pointer to the incoming `CfgConduitInfoType` structure, and `pdwArgsSize` is a pointer to a variable containing the size of that structure, in bytes. Defined in the CDK header file `Condapi.h`, the `CfgConduitInfoType` structure looks like this:

```
typedef struct CfgConduitInfoType {
    DWORD dwVersion;
    DWORD dwSize;
```

```
    DWORD dwCreatorId;
    DWORD dwUserId;
    TCHAR szUser[64];
    char  m_PathName[BIG_PATH];
    eSyncTypes syncPermanent;
    eSyncTypes syncTemporary;
    eSyncTypes syncNew;
    eSyncPref  syncPref;
} CFGCONDUITINFO;
```

Table 18-3 describes the fields in the `CfgConduitInfoType` structure.

Table 18-3
CfgConduitInfoType Fields

Field	*Description*
`dwVersion`	Version number of the `CfgConduitInfoType` structure
`dwSize`	Size of the `CfgConduitInfoType` structure
`dwCreatorId`	Creator ID of the handheld application this conduit is registered to handle
`dwUserId`	User ID number for whom the conduit should be configured
`szUser`	Name of the user for whom the conduit should be configured
`m_PathName`	Pathname that precedes files on the desktop computer
`syncPermanent`	Type of synchronization to perform on a permanent basis
`syncTemporary`	Type of synchronization to perform temporarily (on the next HotSync operation)
`syncNew`	Type of synchronization to perform for a handheld that has never been synchronized before
`syncPref`	Specifies whether the user's selected synchronization preferences should be applied temporarily (on the next HotSync action only) or permanently

Like the **CSyncPreference** class, the `CfgConduitInfoType` structure carries return values back to the HotSync Manager. In particular, you should make sure the `syncPermanent`, `syncTemporary`, and `syncPref` values are set appropriately before the end of the **CfgConduit** function.

As an example of how to use values from the `CfgConduitInfoType` structure, here is the default **CfgConduit** function from a generic conduit project:

```
ExportFunc long CfgConduit (ConduitCfgEnum cfgType,
                            void *pArgs, DWORD *pdwArgsSize)
```

```
{
    long nRtn = -1;
    TCHAR szName[256];
    DWORD dwNamesize;
    ConduitRequestInfoType infoStruct;
    CfgConduitInfoType *pCfgInfo;


    dwNamesize = sizeof(szName);


    if (!pArgs)
        return CONDERR_INVALID_INARGS_PTR;
    if (!pdwArgsSize)
        return CONDERR_INVALID_ARGSSIZE_PTR;
    if (*pdwArgsSize != SZ_CFGCONDUITINFO)
        return CONDERR_INVALID_ARGSSIZE;

    if (cfgType != eConfig1)
            return CONDERR_UNSUPPORTED_CFGCONDUIT_ENUM;

    pCfgInfo = (CfgConduitInfoType *)pArgs;
    if (pCfgInfo->dwVersion != CFGCONDUITINFO_VERSION_1)
        return CONDERR_UNSUPPORTED_STRUCT_VERSION;

    infoStruct.dwVersion = CONDUITREQUESTINFO_VERSION_1;
    infoStruct.dwSize = SZ_CONDUITREQUESTINFO;
    infoStruct.dwCreatorId = pCfgInfo->dwCreatorId;
    infoStruct.dwUserId = pCfgInfo->dwUserId;
    strcpy(infoStruct.szUser, pCfgInfo->szUser);
    nRtn = GetConduitInfo(eConduitName, (void *)&infoStruct,
                          (void *)szName, &dwNamesize);
    if (nRtn)
        return nRtn;

    int iResult;
    iResult = DialogBoxParam((HINSTANCE)hLangInstance,
              MAKEINTRESOURCE(IDD_CONDUIT_CFG_DETAILED),
              GetForegroundWindow(),
              (DLGPROC)ConfigureDlgProc,
              (LPARAM)pCfgInfo);

    return 0;
}
```

Understanding Native Synchronization Logic

Synchronizing two sets of records that may have been modified on both the desktop computer and the handheld requires a large amount of complex logic to resolve conflicts. If the user alters the same record in both places, or deletes the record on one platform and changes it on another, the conduit must be sure to do the "right thing" with the data, ensuring that the result of the sync makes sense to the user and that no data is ever lost that the user did not explicitly delete. Writing code to handle all the possible combinations of modification on two different devices can be sheer hell, particularly when it comes time to debug it.

Fortunately, the developers at Palm Computing have already gone through the agony of implementing native synchronization logic in both sets of base classes. If you use the base classes as a foundation for your conduit, you can concentrate on the details of customizing the conduit for your handheld and desktop application data formats, instead of spending time trying to implement your own two-way mirror synchronization code.

Because it may be helpful for you to know what is going on under the hood, the following table describes what the native sync logic does when it encounters each possible combination of changes that can be made to handheld and desktop records.

Record Status on Handheld	***Record Status on Desktop***	***Conduit Action***
Added	No record	Add the handheld record to the desktop database.
Archived	Deleted	Archive the handheld record, then delete the record from both the handheld and desktop databases.
Archived	No changes	Archive the handheld record, then delete the record from both the handheld and desktop databases.
Archived	No record	Archive the handheld record.
Archived and changed	Changed	Archive both the handheld record and the desktop record if the changes are identical. If the changes are not identical, do not archive the handheld record; instead, add the desktop record to the handheld database, and add the handheld record to the desktop database.
Archived, no changes	Changed	Do not archive the handheld record; instead, replace it with the desktop record.

Continued

Continued

Record Status on Handheld	***Record Status on Desktop***	***Conduit Action***
Changed	Archived and changed	Archive the handheld record if the changes are identical, and then delete the record from both the handheld and desktop databases. If the changes are not identical, do not archive the desktop record; instead, add the desktop record to the handheld database and add the handheld record to the desktop database.
Changed	Archived, no changes	Do not archive the desktop record; instead, replace the record in the desktop database with the handheld record.
Changed	Changed	Take no action if the changes are identical. If the changes are not identical, add the handheld record to the desktop database and add the desktop record to the handheld database.
Changed	Deleted	Do not delete the desktop record; instead, replace the desktop record with the handheld record.
Changed	No changes	Replace the desktop record with the handheld record.
Deleted	Changed	Do not delete the handheld record; instead, replace the handheld record with the desktop record.
Deleted	No changes	Delete the record from the desktop and handheld databases.
No changes	Archived	Archive the desktop record, then delete the record from both the desktop and handheld databases.
No changes	Changed	Replace the handheld record with the desktop record.
No changes	Deleted	Delete the record from both the desktop and handheld databases.

Keeping records in sync is not the only task a conduit implementing mirror image synchronization needs to worry about. It is equally important to maintain consistency between ategories of records on the desktop and the handheld. Like record synchronization logic, this is rather difficult and time-consuming to code, so category syncing is also part of the native synchronization logic.

Maintaining harmony between categories on two devices requires that the sync logic keep an eye on each category's name, category ID, and index location in relation to other categories. The following table shows how the native sync logic deals with different category situations between desktop and handheld.

Name	*Category ID*	*Index*	*Conduit Action*
Desktop name matches handheld name	Not applicable	Desktop index matches handheld index	Take no action.
Desktop name matches handheld name	Not applicable	Desktop index does not match handheld index	Change all desktop records in the category to match the handheld's category ID.
Desktop name does not exist on handheld and desktop name has been modified	Desktop ID matches handheld ID	Not applicable	Change the category name on the handheld to match the desktop.
Desktop name does not exist on the handheld	Desktop ID does not exist on the handheld	Desktop index is not in use on the handheld	Add desktop category to the handheld.
Desktop name does not exist on the handheld	Desktop ID does not exist on the handheld	Desktop index is already in use on the handheld	Add desktop category at the next free index on the handheld, and update desktop records in the category to use the new index.

Using the Palm MFC Base Classes

To create a conduit using the Palm MFC base classes, you must make your own subclasses of the various base classes, then override virtual methods of those classes to customize the conduit's behavior for your own purposes. In order to do all that, you need to know what the base classes are and how they interact with one another. The following list introduces and briefly describes the base classes:

- **CBaseMonitor.** The base monitor class controls the entire synchronization operation, creating, using, and destroying instances of the other base classes as necessary to perform its job. In fact, the **CBaseMonitor** class contains, as data members, a **CBaseDTLinkConverter** object, four **CBaseTable** objects, and two **CCategoryMgr** objects. The monitor class also contains most of the methods responsible for running the synchronization process, including the **Engage** method, which **OpenConduit** calls to set the entire sync procedure in motion.

- **CBaseDTLinkConverter.** A link converter object is responsible for converting records from the handheld into their equivalent format on the desktop computer. It can convert data both ways, from handheld to desktop and vice versa.
- **CCategoryMgr.** Category manager objects manage the categories to which records belong. There are two **CCategoryMgr** objects in the **CBaseMonitor** class, one to keep track of desktop categories and one for handheld categories. Category managers are also present in the **CBaseTable** class.
- **CBaseTable.** A table object contains records in a linear format. The **CbaseMonitor** class uses a number of table objects for storing records from the desktop database, an archive copy of the desktop database, and the handheld.
- **CBaseIterator.** Every table object has an iterator object associated with it that takes care of sorting and searching for records within the table.
- **CBaseSchema.** Each table object also has an associated schema object, which serves as a template for identifying individual records within the linear format the table object uses for storage.
- **CBaseRecord.** All records within a table object are accessed by an object derived from the **CBaseRecord** class. In order for desktop data to mesh properly with the database format used on the handheld, you must heavily modify whatever subclass of **CBaseRecord** you create for your conduit.

Figure 18-12 shows how the various Palm MFC base classes relate to one another. The base classes are all closely related to one another. Some classes are nested within others (for example, there are four **CBaseTable** objects within **CBaseMonitor**), whereas others allow one another access to their internals using the C++ `friend` mechanism (for example, all the functions in **CBaseTable** are friends of **CBaseRecord**).

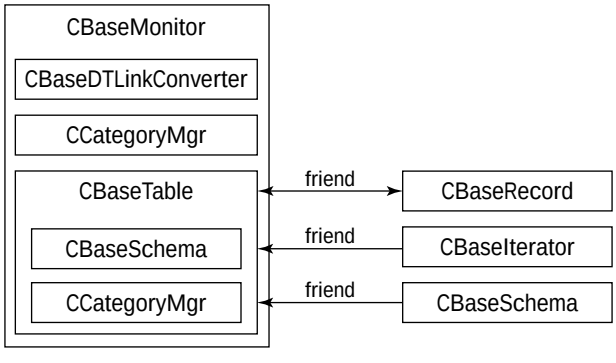


Figure 18-12: How the Palm MFC base classes relate to one another

In order to allow customization of the conduit's actions, the Conduit Wizard generates subclasses of the base classes. You must add the custom behavior by overriding virtual functions in the subclasses. The default subclass names provided by the Conduit Wizard look just like the base class names described previously, substituting your project name for the **Base** part of the name. For example, if your project is named **MyConduit**, the wizard-generated monitor subclass is **CMyConduitMonitor**.

Following MFC Conduit Flow of Control

The flow of control through an MFC-based conduit starts in the **OpenConduit** function, which creates an instance of your conduit's **CBaseMonitor** subclass, then calls the monitor's **Engage** method to start the sync process. After the monitor object has done its job, **OpenConduit** destroys the monitor and returns. Here is a sample **OpenConduit** function, as generated by the Conduit Wizard:

```
ExportFunc long OpenConduit (PROGRESSFN pFn,
                             CSyncProperties& rProps)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    long retval = -1;
    if (pFn)
    {
        CMyConduitMonitor* pMonitor;

        pMonitor = new CMyConduitMonitor(pFn, rProps, myInst);
        if (pMonitor)
        {
            retval = pMonitor->Engage();

            delete pMonitor;
        }
    }
    return(retval);
}
```

Once the monitor object is in control, it creates a pair of table objects (**CBaseTable** subclasses), one to represent the records on the desktop and one to contain records from the handheld database. Using an iterator object (**CBaseIterator** subclass), the monitor steps through the records in the handheld table object, retrieving only modified records from the handheld if the HotSync operation is a FastSync or retrieving all the handheld records in the case of a SlowSync.

As it retrieves handheld records, the monitor converts each record into its desktop format by calling methods from a link converter object (**CBaseDTLinkConverter** subclass). Once the handheld and desktop records are in the same format, the monitor can compare them using the conduit's native sync logic and deal with each record appropriately. See the sidebar "Understanding Native Synchronization Logic" in this chapter for more details about how the native sync logic works.

Implementing a Monitor Class

To make your conduit's subclass of **CBaseMonitor** work, you need to override five of its virtual methods:

- **ConstructRecord.** This method creates a new record object within a table, derived from the **CBaseRecord** class.
- **CreateTable.** This method creates a new table, derived from **CBaseTable**.
- **LogApplicationName.** This method returns the name of your conduit, for purposes of writing the conduit's name to the HotSync log.
- **LogRecordData.** If the monitor object has to report errors with synchronizing a specific record, it needs to write a string describing the problem record to the HotSync log. The **LogRecordData** function assembles this string, based on the unique record format of the handheld application to which the conduit is attached.
- **SetArchiveFileExt**. This method sets up the file extension your conduit appends to its desktop archive file.

The Conduit Wizard makes light work of most of the function overrides; a wizard-generated conduit already has minimal implementations for all but the **LogRecordData** function. Because every handheld application's record format is different, the wizard cannot automatically generate **LogRecordData**. Some handheld applications may store their data as simple text strings, such as the storage method used by the built-in Memo Pad application, but other applications, like the Address Book, use a more complicated structure containing multiple fields of data for each record. The string you return from **LogRecordData** should uniquely identify a single record in the handheld database. For example, if the conduit connects to the Address Book, **LogRecordData** should generate a string consisting of the first name, last name, and company fields in an address book record.

As an example, the following **LogRecordData** function is from the Address Book conduit. Notice how the function casts the **CBaseRecord** `rRec` parameter as the address conduit's own **CAddressRecord** class, then uses methods from **CAddressRecord** to retrieve the appropriate field values needed for the log string.

```
void CAddressConduitMonitor::LogRecordData (CBaseRecord& rRec,
                                            char * errBuff)
{
    CAddressRecord &rLocRec = (CAddressRecord&)rRec;
    CString  csStr;
    int      len = 0;

    rLocRec.GetName(csStr);
    len = csStr.GetLength();
    if (len > 20)
        len = 20;
```

```
    strcpy(errBuff, "       ");
    strncat(errBuff, csStr, len);
    strcat(errBuff, ", ");

    rLocRec.GetFirst(csStr);
    len = csStr.GetLength();
    if (len > 20)
        len = 20;

    strncat(errBuff, csStr, len);
    strcat(errBuff, ", ");

    rLocRec.GetCompany(csStr);
    len = csStr.GetLength();
    if (len > 30)
        len = 30;

    strncat(errBuff, csStr, len);
}
```

Implementing a Table Class

A **CBaseTable** object stores records in a linear format, relying on the position and size of each record to identify individual records within the table. All of the size and position data for a table lies within its associated **CBaseSchema** object, which is described in the next section of this chapter. Most of the really tricky work that goes on in a table is handled by the schema and record objects, but you do need to perform a little bit of customization before the table subclass provided by the Conduit Wizard will work properly.

Conceptually, a conduit table object looks just like a table in a database or spreadsheet. The table is composed of rows and columns; each row represents a single record from the database, and each column is a particular field within that record. Usually, the logical storage structure to use for this sort of construct would be a two-dimensional array, but **CBaseTable** actually keeps the entire mess in a big single-dimensional array, packing records in one after another. A table object relies on its associated schema object to properly position a **CBaseRecord** object at the right place in the array for changing an individual record. Each table object is able to read itself from and write itself out to disk in the standard MFC data serialization format.

To implement a table class, you need to override its **AppendDuplicateRecord** method, which adds a new row to the table. The conduit wizard generates a skeleton for the **AppendDuplicateRecord** method, which looks something like this:

```
long CMyConduitTable::AppendDuplicateRecord(CBaseRecord& rFrom,
                                            CBaseRecord& rTo,
                                            BOOL bAllFlds)
```

```
{
    long  retval = -1;

    CMyConduitRecord& rFromRec = (CMyConduitRecord&)rFrom;
    CMyConduitRecord& rToRec = (CMyConduitRecord&)rTo;

    //
    // Source record must be positioned at valid data.
    //
    if (rFromRec.m_Positioned)
    {
        if (!CBaseTable::AppendBlankRecord(rToRec))
        {
            // TODO - successfully added blank record, now copy
            // the data into it
            if (bAllFlds)
            {
                // copy record ID and status as well
            }
            // copy all remaining field data
        }
    }
    return(retval);
}
```

The monitor object passes **AppendDuplicateRecord** the addresses of a pair of record objects derived from **CBaseRecord**. Of these two record objects, `rFrom` is the address of a record to copy data from, and `rTo` is the address of a newly created blank record where the data should be copied. Note that the first action **AppendDuplicateRecord** takes is to cast the incoming classes as your own conduit's subclass of **CBaseRecord**; in the example above, this happens to be **CMyConduitRecord**. This step is necessary, because **AppendDuplicateRecord** relies on the methods of the derived record class to retrieve and set record fields. Implementing these **Get** and **Set** methods is covered in more detail under "Implementing a Record Class."

The `bAllFlds` argument to **AppendDuplicateRecord** will be `true` if the monitor wants to make an exact copy of a record, including its record ID and status bits. If `bAllFlds` is `false`, the monitor needs only **AppendDuplicateRecord** to copy the record's data, and not the information describing the record itself.

What follows is an example of what **AppendDuplicateRecord** might look like for a hypothetical database containing only one field:

```
long CMyConduitTable::AppendDuplicateRecord(CBaseRecord& rFrom,
                                            CBaseRecord& rTo,
                                            BOOL bAllFlds)
{
    long     retval = -1;
    int      nTemp;
    CString  csTemp;
```

```
    CMyConduitRecord& rFromRec = (CMyConduitRecord&)rFrom;
    CMyConduitRecord& rToRec = (CMyConduitRecord&)rTo;

    //
    // Source record must be positioned at valid data.
    //
    if (rFromRec.m_Positioned)
    {
        if (!CBaseTable::AppendBlankRecord(rToRec))
        {
            if (bAllFlds)
            {
                // If bAllFlds is true, copy all fields,
                // including the record ID.
                retval = rFromRec.GetRecordId(nTemp) ||
                    rToRec.SetRecordId(nTemp);
                if (retval != 0)
                    return retval;

                retval = rFromRec.GetStatus(nTemp) ||
                    rToRec.SetStatus(nTemp);
                if (retval != 0)
                    return retval;

                retval = rToRec.SetArchiveBit(
                    rFromRec.IsArchived() );
                if (retval != 0)
                    return retval;
            }

            // Copy remaining field data.
            retval = rToRec.SetPrivate(rFromRec.IsPrivate() );
            if (retval != 0)
                return retval;

            retval = rFromRec.GetData(csTemp) ||
                rToRec.SetData(csTemp);
            if (retval != 0)
                return retval;

            return 0;
        }
    }
    return(retval);
}
```

There is one other method in **CBaseTable** that you may need to override if your application's data requires special initialization or sorting: **AppendBlankRecord**. This function has a single parameter, which is the address of a **CBaseRecord** object. The declaration of **AppendBlankRecord** looks like this:

```
virtual long CBaseTable::AppendBlankRecord (CBaseRecord& rec);
```

Implementing a Schema Class

Because the schema object is responsible for positioning records within a table object, it must be custom-built to work with the record format used in the table. The subclass of **CBaseSchema** created by the conduit wizard also has a **DiscoverSchema** method, which you need to override, because it sets up data members of the schema object according to the record format.

Before setting up the **DiscoverSchema** function, you should `#define` some constants to identify the fields in each record. These constant definitions should go in the header file that contains the declaration of your conduit's **CBaseSchema** subclass. For example, here are some sample definitions for a very simple record format:

```
#define myFLDRecordID      0
#define myFLDStatus        1

#define myFLDData          2

#define myFLDPrivate       3
#define myFLDCategoryID    4
#define myFLDPosition      5

#define myFLDLast myFLDPosition
```

In this example, the actual data stored by the record format is in the `myFLDData` field; everything else describes various properties of the record itself.

The **DiscoverSchema** function tells the schema object what type of data is present in each field. Another task that **DiscoverSchema** must perform is to let the schema object know where certain key fields are located in the record, such as the record ID and status. Here is an implementation of **DiscoverSchema** that sets up a schema object for the fields described above:

```
long CMyConduitSchema::DiscoverSchema (void)
{
    // Define the number of fields per record.
    m_FieldsPerRow = myFLDLast + 1;
    m_FieldTypes.SetSize(m_FieldsPerRow);

    // Set field positions.
    m_FieldTypes.SetAt(myFLDRecordID,   (WORD)eInteger);
    m_FieldTypes.SetAt(myFLDStatus,     (WORD)eInteger);

    m_FieldTypes.SetAt(myFLDData,       (WORD)eString);

    m_FieldTypes.SetAt(myFLDPrivate,    (WORD)eBool);
    m_FieldTypes.SetAt(myFLDCategoryID, (WORD)eInteger);
    m_FieldTypes.SetAt(myFLDPosition,   (WORD)eInteger);

    // Set the location of the four common fields.
    m_RecordIdPos     = myFLDRecordID;
```

```
    m_RecordStatusPos = myFLDStatus;
    m_CategoryIdPos   = myFLDCategoryID;
    m_PlacementPos    = myFLDPosition;

    return 0;
}
```

Implementing a Record Class

Instances of a record class are not used to actually store record data; that data resides in a table object. Instead, an object derived from **CBaseRecord** provides other parts of the conduit with read and write methods for accessing and modifying the data stored in a table. Your conduit's subclass of **CBaseRecord** will probably require more customization than the other classes in the conduit, simply because you must provide **Set** and **Get** functions for each and every data field in a record. The base class provides methods for reading and writing the record ID, status, category ID, and position of a record, but you must provide your own methods for everything else.

Make sure the fields you define in your record subclass match the fields you defined in your schema subclass. Any disagreement between them will be very messy.

To continue with the simple record format shown in the last example, here is the declaration of a **CMyConduitRecord** class:

```
class CMyConduitRecord : public CBaseRecord
{
protected:
    friend  CLibCondTable;

public:
    CMyConduitRecord  (CBaseTable &rTable,
                       WORD wModAction = MODFILTER_STUPID) :
                       CBaseRecord (rTable, wModAction){};

    CMyConduitRecord  (CMyConduitTable &rTable,
                       WORD wModAction = MODFILTER_STUPID) :
                       CBaseRecord (rTable, wModAction){};

    long GetData      (CString &csData);
    BOOL IsPrivate    (void);

    long SetData      (CString csData);
    long SetPrivate   (BOOL bPrivate);

    // Required function overrides
    long Assign(const CBaseRecord&);
    BOOL operator==(const CBaseRecord&);

};
```

Constructors for a record subclass do not actually need to do anything special, so they simply call the **CBaseRecord** constructor. The `MODFILTER_STUPID` constant used for the `wModAction` parameter's value indicates that the conduit will mark a record dirty if any of its **Set** methods are called to alter its data. If your conduit changes the contents of records during synchronization instead of just keeping them in sync between the desktop and handheld ("smart" filtering, as opposed to the default "stupid" filtering provided by `MODFILTER_STUPID`), you should pass a value of `0` for `wModAction`.

The **CMyConduitRecord** class described previously defines two methods for each data field in the record, one for retrieving its value and one for setting its value. The **GetData** and **IsPrivate** functions look like this:

```
long CMyConduitRecord::GetData (CString &csData)
{
    CStringField* pFld;

    if (m_Positioned &&
        (pFld = (CStringField*) m_Fields.GetAt(myFLDData) ) &&
        pFld->GetValue(csData) == 0)
        return 0;
    else
        return DERR_RECORD_NOT_POSITIONED;
}

BOOL CMyConduitRecord::IsPrivate (void)
{

    CBoolField* pFld;

    if (m_Positioned &&
        (pFld = (CBoolField*) m_Fields.GetAt(myFLDPrivate))) {
        if (pFld->IsTrue() )
            return TRUE;
        else
            return FALSE;
    }
    else
        return DERR_RECORD_NOT_POSITIONED;
}
```

Aside from the **CStringField** and **CBoolField** classes used above for retrieving string and Boolean field values, the Palm MFC base classes also provide **CIntegerField** for integer values, **CDateField** for date values, and **CAlphaField** for fixed-length byte fields. All of these classes are defined in the CDK header file `bfields.h` as subclasses of **CBaseField**, also defined in `bfields.h`.

Before retrieving data, the **Get** functions check the value of the `m_Positioned` member of the record object. This data member keeps track of whether or not the table is positioned on the record in question. If the table is not positioned on the correct record, the **Get** methods return the error constant `DERR_RECORD_NOT_POSITIONED`.

Following is **SetData**, the partner function to **GetData**. **SetPrivate** is almost identical, so it is not listed here. Notice that **SetData** leaves the modification status of the record alone if the new value of the field is equal to the old value; if they are identical, no changes have really been made, so no update is required.

```
long CMyConduitRecord::SetData (CString csData)
{
    BOOL      bFlipStatus = FALSE;
    int       nStatus     = 0;
    long      retval      = DERR_RECORD_NOT_POSITIONED;
    CString*  pFld        = NULL;


    if (m_Positioned &&
        (pFld = (CString*) m_Fields.GetAt(myFLDData))) {
        if (m_wModAction == MODFILTER_STUPID) {
            GetStatus(nStatus);
            if (nStatus != fldStatusADD) {
                CString csTemp(csData);
                if (pFld->Compare(&csTemp))
                    bFlipStatus = TRUE;
            }
        }
        if (!pFld->SetValue(csData)) {
            if (bFlipStatus)
                SetStatus(fldStatusUPDATE);
            retval = 0;
        }
    }

    return retval;
}
```

The `fldStatusADD` and `fldStatusUPDATE` constants used above are part of a family of status constants defined in the CDK header file `tables.h`. These constants describe the status of a desktop record. Table 18-4 describes each of the constants and what they signify.

Table 18-4
Desktop Field Status Constants

Constant	*Value*	*Description*
`fldStatusNONE`	`0`	The record has not changed since the last sync.
`fldStatusADD`	`0x01`	The record has been added since the last sync.
`fldStatusUPDATE`	`0x02`	The record has been modified since the last sync.
`fldStatusDELETE`	`0x04`	The record has been deleted since the last sync.
`fldStatusPENDING`	`0x08`	The record requires some kind of action at the end of the synchronization, depending on other circumstances.
`fldStatusARCHIVE`	`0x80`	The record has been archived since the last sync.

Not only do you need to provide the various **Get** and **Set** methods, you need to override a pair of virtual functions before your subclass of **CBaseRecord** will work properly. You first need to override **Assign**, which copies one record object into another. The conduit passes the address of the record object to copy from, which you should cast to your own subclass of **CBaseRecord** before iterating over the record's fields and copying them.

```
long CMyConduitRecord::Assign (const CBaseRecord& rSubj)
{
    if (!m_Positioned)
        return -1;
    for (int i = myFLDRecordID; i <= myFLDLast; i++) {
        CBaseField* pMyFld = (CBaseField*) m_Fields.GetAt(i);
        CBaseField* pSubjFld = (CBaseField*)
            ( (CMyConduitRecord&) rSubj).m_Fields.GetAt(i);
        if (pMyFld && pSubjFld)
            pMyFld->Assign(*pSubjFld);
    }

    return 0;
}
```

You also need to override the comparison operator (**operator==**) to compare two record objects for equality. This routine should ignore the record ID and status, relying on the record's own fields for the comparison. If you were clever and defined the record ID and status field constants as the first two `#define` statements earlier on, all you need to do is start at the next available field and start iterating through the fields, comparing each one.

```
BOOL CMyConduitRecord::operator== (const CBaseRecord& rSubj)
{
    if (!m_Positioned)
```

```
        return FALSE;
    for (int i = myFLDData; i <= myFLDLast; i++) {
        CBaseField* pMyFld = (CBaseField*) m_Fields.GetAt(i);
        CBaseField* pSubjFld = (CBaseField*)
            ( (CMyConduitRecord&)rSubj).m_Fields.GetAt(i);
        if (!pMyFld || !pSubjFld)
            return FALSE;
        if (pMyFld->Compare(pSubjFld) != 0)
            return FALSE;
    }

    return TRUE;
}
```

Implementing a Link Converter Class

The last class you need to implement is the link converter, which translates records between their handheld format and the format used by your subclass of **CBaseRecord**. To implement the link converter, you must override two functions: **ConvertToRemote** and **ConvertFromRemote**. These functions have the following prototypes:

```
long ConvertToRemote   (CBaseRecord &rRec,
                        CRawRecordInfo &rInfo);

long ConvertFromRemote (CBaseRecord &rRec,
                        CRawRecordInfo &rInfo);
```

The first thing these functions should do is cast `rRec` as your own conduit's subclass of **CBaseRecord**, then use the **Get** and **Set** methods of the record subclass to retrieve or set values in `rRec` as appropriate. When the conduit calls either function, it allocates a buffer in `rInfo.m_pBytes`, which you should write to in **ConvertToRemote**, or read from in **ConvertFromRemote**. The **CRawRecordInfo** class is defined in the CDK header file `Syncmgr.h` as follows:

```
class CRawRecordInfo
{
public:
    BYTE   m_FileHandle;
    DWORD  m_RecId;
    WORD   m_RecIndex;
    BYTE   m_Attribs;
    short  m_CatId;
    int    m_ConduitId;
    DWORD  m_RecSize;
    WORD   m_TotalBytes;
    BYTE*  m_pBytes;
    DWORD  m_dwReserved;
};
```

Table 18-5 describes the data members that make up **CRawRecordInfo** and, by extension, the `rInfo` parameter.

Table 18-5
CRawRecordInfo Data Members

Member	*Description*
`m_FileHandle`	Handle to the database on the handheld
`m_RecId`	Record ID
`m_RecIndex`	Sequential index of the record within the handheld database
`m_Attribs`	Record attribute flags
`m_CatId`	Category ID
`m_ConduitId`	Unused
`m_RecSize`	Actual amount of data in the record, in bytes
`m_TotalBytes`	Size of the buffer allocated for data
`m_pBytes`	Pointer to a buffer containing the record's data
`m_dwReserved`	Reserved for future use

Following earlier examples in this section, here is a **ConvertToRemote** function to convert a **CMyConduitRecord** object to its handheld data format:

```
long CMyConduitDTLinkConverter::ConvertToRemote (
    CBaseRecord& rRec, CRawRecordInfo& rInfo)
{
    long     retval = 0;
    char     *pBuff;
    CString  csTemp;
    DWORD    length;
    int      nTemp, destLen;
    char     *pSrc;

    CMyConduitRecord& rAddrRec = (CMyConduitRecord &)rRec;
    rInfo.m_RecSize = 0;

    // Convert the record ID and category ID.
    retval = rAddrRec.GetRecordId(nTemp);
    rInfo.m_RecId = (long)nTemp;
    retval = rAddrRec.GetCategoryId(nTemp);
    rInfo.m_CatId = nTemp;
```

```
    // Convert record attributes.
    rInfo.m_Attribs = 0;
    if (rAddrRec.IsPrivate())
        rInfo.m_Attribs |= PRIVATE_BIT;
    if (rAddrRec.IsArchived())
        rInfo.m_Attribs |= ARCHIVE_BIT;
    if (rAddrRec.IsDeleted())
        rInfo.m_Attribs |= DELETE_BIT;
    if (rAddrRec.IsModified() || rAddrRec.IsAdded())
        rInfo.m_Attribs |= DIRTY_BIT;

    pBuff = (char*)rInfo.m_pBytes;

    // Convert the body of the record.
    retval = rAddrRec.GetData(csTemp);
    length = csTemp.GetLength();
    if (length != 0) {
        // Strip the carriage returns; StripCRs places its
        // result directly into pBuff.
        pSrc    = csTemp.GetBuffer(length);
        destLen = StripCRs(pBuff, pSrc, length);
        csTemp.ReleaseBuffer(-1);

        // If there were more fields in the record, the
        // following lines would advance the buffer pointer and
        // add to the record's total size:
        //     pBuff += destLen;
        //     rInfo.m_RecSize += destLen;
    }

    return retval;
}
```

Here is the counterpart to the last example, **ConvertFromRemote**:

```
long CAddrCondDTLinkConverter::ConvertFromRemote (
    CBaseRecord& rRec, CRawRecordInfo& rInfo)
{
    long     retval = 0;
    char     *pBuff;
    int      nTemp;
    CString  csTemp;

    CMyConduitRecord& rAddrRec = (CMyConduitRecord &)rRec;

    // Convert record attributes.
    retval = rAddrRec.SetRecordId(rInfo.m_RecId);
    retval = rAddrRec.SetCategoryId(rInfo.m_CatId);
    if (rInfo.m_Attribs & ARCHIVE_BIT)
        retval = rAddrRec.SetArchiveBit(TRUE);
    else
        retval = rAddrRec.SetArchiveBit(FALSE);
```

```
    if (rInfo.m_Attribs & PRIVATE_BIT)
        retval = rAddrRec.SetPrivate(TRUE);
    else
        retval = rAddrRec.SetPrivate(FALSE);

    retval = rAddrRec.SetStatus(fldStatusNONE);
    if (rInfo.m_Attribs & DELETE_BIT)
        retval = rAddrRec.SetStatus(fldStatusDELETE);
    else if (rInfo.m_Attribs & DIRTY_BIT)
        retval = rAddrRec.SetStatus(fldStatusUPDATE);

    // Only convert the body of the record if the remote record
    // is not deleted.
    if (!(rInfo.m_Attribs & DELETE_BIT)) {
        pBuff = (char*)rInfo.m_pBytes;

        // Add in any necessary carriage returns; AddCRs places
        // its result in m_TransBuff.
        AddCRs(pBuff, strlen(pBuff));
        csTemp = m_TransBuff;
        retval = rAddrRec.SetName(csTemp);

        // If there were more fields in the record, the
        // following line would advance the pointer in the
        // buffer to the next field:
        //     pBuff += strlen(pBuff) + 1;
    }

    return retval;
}
```

Though the example above does not require them, the **CBaseDTLinkConverter** class contains methods to easily swap byte orders between the handheld and the desktop, which is quite useful when you consider that a Palm OS handheld uses Motorola big-endian byte order and a Windows desktop machine uses Intel little-endian. The functions available are:

- **SwapDWordToMotor.** This function translates a little-endian Intel `DWORD` into a big-endian Motorola `DWORD`.
- **SwapDWordToIntel.** This function performs the reverse conversion from that made by **SwapDWordToMotor**.
- **FlipWord.** You can use **FlipWord** to flip a `WORD` value back and forth between Intel and Motorola formats.

The **CBaseDTLinkConverter** class also has methods for converting between the date format used in the Palm OS (`DateType` in the Palm OS headers, `TdDateType` in the conduit base classes) and the MFC **CTime** format. The **ConvertToTdDate** translates **CTime** to `DateType`, and **ConvertFromTdDate** reverses the process.

Using the Generic Conduit Base Classes

Just like creating a conduit using the Palm MFC base classes, making a conduit using the generic conduit base classes is a matter of subclassing the base classes and overriding certain functions to provide your conduit with its own custom behavior. This is where the similarity ends between the two sets of base classes; the generic conduit classes have a different structure and flow of control from those of the MFC classes. The following list briefly describes the function of each of the generic conduit base classes. Throughout this section, any class name you see that contains **MyConduit** will actually contain the project name you supplied to the Conduit Wizard instead of **MyConduit**.

- **CSynchronizer.** The synchronizer object is responsible for performing the actual synchronization of records, categories, and the handheld program's application info block. All of the native sync logic is implemented in **CSynchronizer**.
- **CPDbBaseMgr.** This class serves as the foundation for **CPcMgr** and **CHHMgr**. The various manager classes control storage and retrieval of records, from both the desktop and the handheld.
- **CPcMgr.** The **CPcMgr** class deals with storing and retrieving records from the desktop. Your generic conduit will have at least one subclass of **CPcMgr** (named something like **CMyConduitPcMgr**), customized for handling whatever desktop format you choose to implement in the conduit. There might also be subclasses of this subclass, used for archiving (**CMyConduitArchiveMgr**) and backing up (**CMyConduitBackupMgr**) records on the desktop.
- **CHHMgr.** Counterpart to **CPcMgr**, **CHHMgr** stores and retrieves data on the handheld end of the conduit. In general, there should be no need to subclass **CHHMgr**; in fact, the Conduit Wizard does not even bother to generate a **CMyConduitHHMgr** function.
- **CPCategoryMgr** and **CPCategory.** The synchronizer object uses these classes to sync categories between the handheld and the desktop.
- **CPalmRecord.** This class provides a generic record storage format on the desktop. The **CPalmRecord** class contains methods for converting itself to and from the raw record format used on the handheld, as well as routines for making common conversions between Intel and Motorola byte ordering.
- **CMyConduitRecord.** This is not really a base class but rather a class you should define yourself. If you want to store data in something other than the default serialization format used by the base classes, you need to provide your own class to define the data format of your records on the desktop.

Figure 18-13 shows how the various generic conduit base classes, and the subclasses you derive from them, relate to one another.

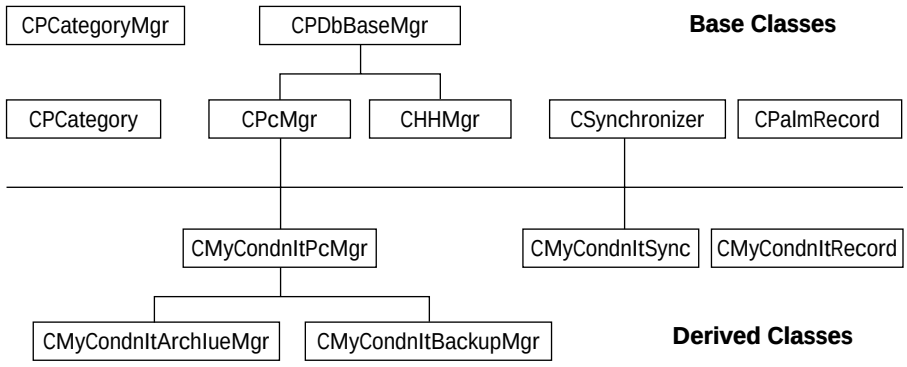


Figure 18-13: How the generic conduit classes relate to one another

Following Generic Conduit Flow of Control

Like any other conduit, a generic conduit starts syncing in its **OpenConduit** function. The **OpenConduit** function instantiates your subclass of **CSynchronizer**, then invokes the synchronizer object's **Perform** method to start the entire sync process. After the synchronizer has finished its job, **OpenConduit** destroys it and exits. The **OpenConduit** function generated by the Conduit Wizard looks like this:

```
ExportFunc long OpenConduit (PROGRESSFN pFn,
                             CSyncProperties& rProps)
{
    long retval = -1;
    if (pFn)
    {
        CLibCondSync* pGeneric;
        pGeneric = new CLibCondSync(rProps,
            GENERIC_FLAG_CATEGORY_SUPPORTED |
            GENERIC_FLAG_APPINFO_SUPPORTED);
        if (pGeneric){
            retval = pGeneric->Perform();

            delete pGeneric;
        }
    }
    return(retval);
}
```

The synchronizer object's **Perform** method creates two manager objects, one for the desktop (**CMyConduitPcMgr**) and one for the handheld (**CHHMgr**). Depending on the type of synchronization that should be performed, the managers retrieve records from the handheld and the desktop (all of them in the case of a SlowSync, only modified records for a FastSync).

Next, the synchronizer creates and uses **CPCategoryMgr** and **CPCategory** objects to synchronize categories between handheld and desktop. Then the synchronizer syncs the database's application info block if it contains any extra information besides just categories, such as the database's sort order.

Once category and application info block synchronization are out of the way, the synchronizer can concentrate on getting the records in sync between the desktop and the handheld, using the native sync logic built into the **CSynchronizer** class. When finished, the synchronizer destroys the managers it created.

To get your own generic conduit up and running, perform the following steps:

1. Describe the format of records on the desktop.
2. Implement desktop record storage and retrieval.
3. Implement conversion between your own data format and **CPalmRecord**.
4. Implement synchronization of the application info block.

Describing the Desktop Record Format

You can use a generic conduit created by the Conduit Wizard "out of the box," without adding a single line of code to it. If you make no modifications, a wizard-generated generic conduit will interface with any Palm OS record database and store its contents on the desktop in MFC serialized format. However, this makes the generic conduit no different from a conduit developed from the Palm MFC base classes. To truly make use of the flexibility of a generic conduit, you need to make some changes to it so that it can read from and write to any arbitrary data format you want.

Because the **CPalmRecord** class is more than adequate to the task of representing records on the handheld, you need to provide only a class to correspond to records on the desktop. This desktop record class is entirely for your own convenience; a generic conduit can get along just fine without it, but having a desktop record class will make your life easier when you have to implement storage and retrieval of desktop records.

The Conduit Wizard does not create even a skeleton for a desktop record class, so you will need to make your own from scratch. The Visual C++ WizardBar's **New Class** option is an easy way to start this task, because it creates new `.h` and `.cpp` files to contain the class, along with a very basic code framework to fill out. What follows is a sample declaration for a very simple record class, one that contains only a single string for its data:

```
class CMyConduitRecord
{
public:
    CMyConduitRecord(void);
    ~CMyConduitRecord();
```

```
public:
    DWORD m_dwRecordID;
    DWORD m_dwStatus;
    DWORD m_dwPosition;
    CPString m_csData;
    DWORD m_dwPrivate;
    DWORD m_dwCategoryID;

public:
    void Reset(void);
};
```

The **CPString** class used above to store the record's data is a string class provided by the CDK. A **CPString** object is essentially a Palm Computing implementation of the standard MFC **CString** class, containing all the useful string-handling methods and operators you might be used to in the **CString** class. You should be sure to `#include "CPString.h"` at the head of your record's `.h` file if you want to take advantage of the **CPString** class.

Here is the actual implementation of the **CMyConduitRecord** class declared above:

```
CMyConduitRecord::CMyConduitRecord (void)
{
    Reset();
}


CMyConduitRecord::~CMyConduitRecord()
{
}


void CMyConduitRecord::Reset(void)
{
    m_dwRecordID = 0;
    m_dwStatus = 0;
    m_dwPosition = 0;
    m_dwPrivate = 0;
    m_dwCategoryID = 0;
    m_csData.Empty();
}
```

There is nothing fancy here; the **CMyConduitRecord** class is pretty much just a structure that initializes its own fields. If you want to, you can declare and implement other methods in your record class to aid in storing and retrieving its contents from whatever storage format you use on the desktop computer.

Implementing Storage and Retrieval

Once you have defined a class that describes your desktop data format, you need to override a couple functions in your conduit's subclass of **CPcMgr** to implement storage and retrieval of desktop records. The two methods you need to concern yourself with are **RetrieveDB** and **StoreDB**. By default, the code generated by the Conduit Wizard simply defers to the **RetrieveDB** and **StoreDB** methods of the actual **CPcMgr** class, which read and write data in the MFC serialization format.

Implementing RetrieveDB

The following partial **RetrieveDB** function takes care of some required bookkeeping tasks, such as reading in the database and application info blocks. The **RetrieveDB** function that follows also leaves a spot in the middle for your own code to retrieve data from the desktop. Much of this code is drawn straight from the **RetrieveDB** method of the base **CPcMgr** class; there is no need to reinvent the wheel here when much of the required code already exists.

```
long CMyConduitPcMgr::RetrieveDB (void)
{
    m_bNeedToSave = FALSE;

    // Check validity of the desktop file name and the handle
    // to that file.
    if (!_tcslen(m_szDataFile))
        return GEN_ERR_INVALID_DB_NAME;
    if (m_hFile == INVALID_HANDLE_VALUE)
        return GEN_ERR_INVALID_DB;

    // Initialize a space for the database info block.
    BOOL bDone = FALSE;
    long retval = 0;
    retval = CreateDBInfo();
    if (retval){
        CloseDB();
        return retval;
    }

    // Read in the database info block from disk.
    retval = ReadInData((LPVOID)m_pDBInfo,
                        CM_STORAGE_HEADER_SIZE);

    if (retval) {
        CloseDB();
        if (retval == GEN_ERR_STORAGE_EOF)
            return 0;
        return GEN_ERR_READING_DBINFO;
    }
```

```
    // Read the application info block from disk.
    if (m_pDBInfo->dwAppInfo > 0) {
        m_pAppInfo = (STORAGE_INFO_PTR)
            malloc(m_pDBInfo->dwAppInfo);
        if (!m_pAppInfo)
            retval =  GEN_ERR_LOW_MEMORY;
        if (!retval) {
            retval = ReadInData((LPVOID)m_pAppInfo,
                                m_pDBInfo->dwAppInfo);
        }
        if (retval) {
            CloseDB();
            return GEN_ERR_READING_APPINFO;
        }
    }

    // Code for reading data from the desktop goes here.

    CloseDB();
    m_bNeedToSave = FALSE;
    return retval;
}
```

The `m_szDataFile` and `m_hFile` data members of the manager class store the file name for the desktop data source and a handle to that file, respectively. When it calls **OpenConduit** to start record synchronization, the HotSync Manager passes the name of the desktop data file as part of a **CSyncProperties** object, which eventually trickles its way down to the desktop manager object and becomes the `m_szDataFile` and `m_hFile` members. You can use the file handle provided in `m_hFile` to read data from the desktop data source.

The generic conduit code assumes that your data source on the desktop is a file. If you want to pipe in data from a different source, you will also need to override the Open, OpenDB, Close, and CloseDB methods of CPcMgr to handle opening and closing the unusual data source.

After checking that `m_szDataFile` and `m_hFile` contain valid data, your **RetrieveDB** method should initialize space in memory for a database information block. The **CreateDBInfo** method of **CPcMgr** takes care of this task, setting a pointer to the initialized memory block in the manager object's `m_pDBInfo` member variable. The pointer stored in `m_pDBInfo` points to a `STORAGE_HEADER` structure, which is defined as follows in the CDK header file `pcmgr.h`:

```
typedef struct StorageHeaderType {
    DWORD   dwStructSize;
    WORD    wVersion;

    char    szName[DB_NAMELEN];
    DWORD   dwDBVersion;
    DWORD   dwDBCreator;
    DWORD   dwDBType;
    WORD    wDBFlags;
```

```
    long    lDBModDate;
    DWORD   dwDBModNumber;
    long    lDBCreateDate;
    long    lDBBackupDate;
    long    lDBRecCount;
    long    dwAppInfo;
    long    dwSortInfo;
} STORAGE_HEADER;
```

The `STORAGE_HEADER` structure contains information about the database itself, including its number of records. It also contains the length of the database's application info block, in bytes, in its `dwAppInfo` field. The **RetrieveDB** method should look in `dwAppInfo` for the length of the database's application info block; armed with this value, **RetrieveDB** can then copy that many bytes of data into a memory block pointed to by the desktop manager object's `m_pAppInfo` variable.

In the previous example, as in the **CPcMgr** base class itself, **RetrieveDB** uses the **ReadInData** method of the manager object to retrieve the database and application info blocks from the desktop data source. The **ReadInData** function provided in **CPcMgr** is set up to read data from an MFC serialized file, but you can supply your own override of **ReadInData** to read data from whatever format you wish to use on the desktop. The arguments **ReadInData** takes are a pointer to a buffer to receive data and the number of bytes to read in.

The structure used in this example, using ReadInData to read raw data from the file, works great if you want to store your application's information in a straight binary format. If you want to use something more human-readable, like comma-delimited values, or write straight to something like an ODBC database, you will need to rework how RetrieveDB does its job. This example is only one possible way of implementing RetrieveDB.

Implementing StoreDB

Many parts of **StoreDB** will look similar to your **RetrieveDB** function, only in reverse, because it writes information to disk instead of reading.

```
long CMyConduitPcMgr::StoreDB (void)
{
    // Check to see if there were any changes to the data. If
    // not, save time by not saving unnecessarily.
    if (!m_bNeedToSave) {
        if ((!m_pCatMgr) || (!m_pCatMgr->IsChanged()))
            return 0;
    }

    // Open the desktop database for writing.
    long retval = OpenDB();
    if (retval)
        return GEN_ERR_UNABLE_TO_SAVE;

    // Store database info block. If there is no info block,
    // then there is nothing to store.
```

```
    if (!m_pDBInfo){
        CloseDB();
        return 0; // No error if nothing to store
    }

    if (m_pDBInfo){
        m_pDBInfo->lDBRecCount = (long)m_dwRecordCount;
        if (m_pAppInfo)
            m_pDBInfo->dwAppInfo = m_pAppInfo->dwStructSize;
        else
            m_pDBInfo->dwAppInfo = 0;

        retval = WriteOutData(m_pDBInfo,
                              m_pDBInfo->dwStructSize);
        if (retval != 0){
            CloseDB();
            return GEN_ERR_UNABLE_TO_SAVE;
        }
    }

    // Store application info block, if it exists.
    if (m_pAppInfo) {
        retval = WriteOutData(m_pAppInfo,
                              m_pAppInfo->dwStructSize);
        if (retval != 0){
            CloseDB();
            return GEN_ERR_UNABLE_TO_SAVE;
        }
    }

    // Code for writing data to the desktop goes here.

    CloseDB();
    m_bNeedToSave = FALSE;
    return 0;
}
```

The first thing **StoreDB** does is to check the desktop manager object's `m_bNeedToSave` variable. If `m_bNeedToSave` is `FALSE`, then there have been no changes to the data that would require saving. In this case, **StoreDB** can simply exit, as it has nothing to do.

If **StoreDB** does, indeed, have something to do, it calls the **CPcMgr** class method **OpenDB** to open the desktop file for writing. Then **StoreDB** writes the database information block to the beginning of the file, followed by the application info block, but only if these pieces of information are available. Once the header information is out of the way, **StoreDB** can get to the work of writing actual records to the desktop.

Like its sibling **ReadInData**, the **WriteOutData** method of **CPcMgr** writes raw bytes of data to the desktop file. You will need to replace **WriteOutData** with your own code to write to something other than MFC serialization format.

Besides overriding **RetrieveDB** and **StoreDB**, if your conduit supports archiving of deleted records or keeping a backup copy of the desktop database, the Conduit Wizard will generate subclasses of your **CPcMgr** subclass to handle archive and backup databases. Unless you need to do something special with the archive and backup databases, such as saving these two databases in a different format from your main desktop database, you can leave the **CMyConduitArchive** and **CMyConduitBackupMgr** classes generated by the Conduit Wizard alone; they will automatically handle archiving and backing up your desktop data.

Converting Data to and from CPalmRecord

The next step in making a generic conduit is implementing conversion between your desktop record format and the generic **CPalmRecord** class. To do this, you must override the **ConvertGenericToPc** and **ConvertPcToGeneric** methods of **CPcMgr**. The following is an example of a **ConvertGenericToPc** method, using the **CMyConduitRecord** class defined earlier in the chapter:

```
long CMyConduitPcMgr::ConvertGenericToPc (CPalmRecord &palmRec,
    CMyConduitRecord &rec, BOOL bClearAttributes)
{
    BYTE   *pBuff;
    long   retval    = 0;
    int    nLength;

    rec.Reset();

    if (palmRec.GetRawDataSize() == 0){
        // This is a deleted record because it has no data.
        return GEN_ERR_EMPTY_RECORD;
    }

    // Fill in the record ID.
    rec.m_dwRecordID = palmRec.GetID();

    // Unless the attributes should be thrown out
    // (bClearAttributes == TRUE), fill in record attributes.
    if (bClearAttributes)
        rec.m_dwStatus = 0;
    else {
        rec.m_dwStatus = palmRec.GetAttribs();
    }

    // Fill in position, category ID, and the private status.
    rec.m_dwPosition   = palmRec.GetIndex();
    rec.m_dwCategoryID = palmRec.GetCategory();
    rec.m_dwPrivate    = palmRec.IsPrivate();

    DWORD dwRawSize = palmRec.GetRawDataSize();
    if (!dwRawSize) {
        // This is an invalid record.
        return 0;
    }
```

```
    pBuff = palmRec.GetRawData();

    // Convert the actual record data. Add carriage returns.
    nLength = _tcslen((char *)pBuff);

    AddCRs((char *)pBuff,
           rec.m_csData.GetBuffer(nLength * 2), nLength);
    rec.m_csData.ReleaseBuffer();

    pBuff += nLength + 1;

    return retval;
}
```

Most of the example **ConvertGenericToPc** function above involves copying members of the **CPalmRecord** object `palmRec`, such as the record ID and attributes, straight into members of the **CMyConduitRecord** object `rec`, using **CPalmRecord** methods to retrieve the appropriate values, because **CPalmRecord** stores just about anything of interest in protected member variables. Retrieving the actual data relies on the **CPalmRecord** class **GetRawData** method, which returns a pointer to the raw data of a record.

The only real conversion performed by **CMyConduitPcMgr::ConvertGenericToPc** is to add carriage return characters to the data, because Windows ends lines of text with both carriage return and linefeed characters, and the Palm OS uses only linefeeds. This conversion is made easy by the use of the **CPcMgr::AddCRs** method. The **CPcMgr** class also has a companion **StripCRs** method for removing carriage returns when converting data back to the handheld database.

Here is the companion **ConvertPcToGeneric** function:

```
long CMyConduitPcMgr::ConvertPCtoGeneric(CMyConduitRecord &rec,
                                         CPalmRecord &palmRec)
{
    long   retval = 0;
    char   *pBuff;
    int    destLen;
    BYTE   szRawData[MAX_RECORD_SIZE];
    DWORD  dwRecSize = 0;


    // Copy over all the record fields and attributes.
    palmRec.SetID(rec.m_dwRecordID);

    palmRec.SetCategory(rec.m_dwCategoryID);

    if (rec.m_dwPrivate)
        palmRec.SetPrivate(TRUE);
    else
        palmRec.SetPrivate(FALSE);
```

```
        palmRec.SetArchived((BOOL)(rec.m_dwStatus &
                                   fldStatusARCHIVE));
        palmRec.SetDeleted((BOOL)(rec.m_dwStatus &
                                  fldStatusDELETE));
        palmRec.SetUpdate((BOOL)(rec.m_dwStatus &
                                 fldStatusUPDATE));

        // Convert the actual record data.
        pBuff = (char*)szRawData;

        int nLength;
        nLength = rec.m_csData.length();
        if (nLength > 0) {
            // Strip carriage returns. StripCRs places its result
            // directly into pBuff.
            destLen = StripCRs(pBuff, rec.m_csData.c_str(),
                               nLength);

            pBuff += destLen;
            dwRecSize += destLen;
        }

        if (dwRecSize == 0) {
            // A record without any data has been deleted.
            palmRec.SetDeleted();
        }

        retval = palmRec.SetRawData(dwRecSize, szRawData);

        return(retval);
    }
```

The `MAX_RECORD_SIZE` constant, used to define the size of the buffer that **ConvertPcToGeneric** puts the record's actual raw data into, is defined in the CDK header file `CPalmRec.h` as the value `0xfff0`, or 65,520 bytes, which is the largest possible record allowed by the Palm OS data manager.

Syncing the Application Info Block

The last thing to implement in a generic conduit is synchronization of the database's application info block. The synchronizer object uses the **CCategoryMgr** object to automatically convert category information stored at the head of an application info block, so only extra information stored in the application info block needs to be synced. Because not every Palm OS application stores extra information in its application info block, not all conduits need to explicitly sync this data. Overriding **CPcMgr::SynchronizeAppInfo** allows you to perform special processing of the application info block data on its way between the handheld and the desktop.

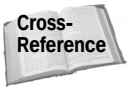
Using the Sync Manager API

If your conduit does not keep records in mirror image sync between the desktop and handheld, or if your conduit is very simple and does not require the record-synchronization logic provided by the Palm MFC or Generic Classes, you can use the Sync Manager API directly. The Sync Manager API contains a large number of very low-level functions for communicating directly with a Palm OS handheld during a HotSync operation. This section is only a brief overview of using Sync Manager functions; for exhaustive details, see the *Conduit Programmer's Reference*, which ships as part of the CDK.

If any call to a Sync Manager function results in an error, treat the other return values from the function as unusable. Using any of these undefined values is likely to cause data corruption.

Registering and Unregistering a Conduit

Before a conduit can make any other Sync Manager calls, it must be registered with the Sync Manager by calling the **SyncRegisterConduit** function.



This registration is not the same as registering the conduit upon installation. See Chapter 17, "Introducing Conduit Mechanics," for more information about registering a conduit when it is installed.

The prototype for **SyncRegisterConduit** looks like this:

```
long SyncRegisterConduit(CONDHANDLE &rHandle)
```

The handle returned in the `rHandle` parameter is required by some other calls your conduit makes to Sync Manager functions. The **SyncRegisterConduit** function returns `0` if it successfully registered your conduit, or `-1` if another conduit that ran before yours did not unregister itself properly. Only one conduit may be registered at a time, so it is imperative that you unregister your conduit before it exits, using the **SyncUnRegisterConduit** function:

```
long SyncUnRegisterConduit(CONDHANDLE handle)
```

The **SyncUnRegisterConduit** function returns `0` if it successfully unregisters your conduit, or `-1` if you provide the function with an invalid `handle` parameter.

Opening and Closing Handheld Databases

Normally, you will use the **SyncOpenDB** function to open an existing database on the handheld. The prototype for **SyncOpenDB** looks like this:

```
long SyncOpenDB(char *pName, int nCardNum, Byte& rHandle,
    Byte openMode)
```

The `pName` parameter is a null-terminated string containing the name of the database to open, and `nCardNum` receives the card number on which the database is located (`0` on all current Palm OS handhelds). You can specify the mode in which to open the database using the `openMode` parameter, which should be one or more `eDbOpenModes` constants combined with the OR operator (`|`). The CDK headers define `eDbOpenMode` like this:

```
enum eDbOpenModes {eDbShowSecret = 0x0010,
                   eDbExclusive  = 0x0020,
                   eDbWrite      = 0x0040,
                   eDbRead       = 0x0080
};
```

In general, you should almost always include the `eDbShowSecret` constant because it enables access to records marked private on the device. The `eDbShowSecret` flag affects how the **SyncReadNextRecInCategory** and **SyncReadNextModifiedRec InCategory** functions (see the "Iterating Over Database Records" section, later in this chapter) operate. These two functions skip over private records if the database was opened without the `eDbShowSecret` flag.

The `eDbExclusive` flag opens the database in exclusive mode, which prevents any other application on the handheld from accessing the database while it is open. This flag is of only limited utility, however, because the Sync Manager does not allow any applications to run during a HotSync operation anyway.

Including the `eDbWrite` flag opens a database for writing, and the `eDbRead` flag opens a database for reading. You can use both of these flags together to allow reading and writing to a database in the same open action. For example, to open a database in read/write mode, with access to private records, pass the following value for `openMode`:

```
(eDbShowSecret | eDbWrite | eDbRead)
```

The **SyncOpenDB** function returns a handle to the opened database in its `rHandle` parameter, as well as a value of `0` in its return value for a successful opening, or an error code if the database could not be opened for some reason.

If the database you want to access does not already exist on the handheld, you can create and open a brand new database using the **SyncCreateDB** function, which has the following prototype:

```
long SyncCreateDB (CDbCreateDB& rDbStats)
```

The `rDbStats` parameter is an object of class **CDbCreateDB**, which looks like this:

```
class CDbCreateDB
{
public:
    BYTE m_FileHandle;
    DWORD m_Creator;
```

```
    eDbFlags m_Flags;
    BYTE m_CardNo;
    char m_Name[SYNC_DB_NAMELEN];
    DWORD m_Type;
    WORD m_Version;
    DWORD m_dwReserved;
};
```

Table 18-6 describes the data members of the **CDbCreateDB** class, as they relate to the **SyncCreateDB** function.

Table 18-6
CDbCreateDB Data Members

Data Member	*Description*
`m_FileHandle`	Receives a handle to the newly created and opened database when SyncCreateDB returns.
`m_Creator`	Creator ID for the new database.
`m_Flags`	Database creation flags, a combination of `eDbFlags` constants.
`m_CardNo`	Number of the card in which to create the database. Use the value `0` for current Palm OS handhelds.
`m_Name`	Null-terminated string containing the name for the new database.
`m_Type`	Four-character type of the new database.
`m_Version`	Database version.
`m_dwReserved`	Reserved for future use.

Although the Sync Manager API contains a number of different classes, none of these classes contains any actual methods, only data members. You can use them much in the same way you would use structures in a C application.

The `eDbFlags` enumerated type, used in the `m_Flags` member of the **CDbCreateDB** class, is defined like this:

```
enum eDbFlags {
    eRecord            = 0x0000,
    eResource          = 0x0001,
    eReadOnly          = 0x0002,
    eAppInfoDirty      = 0x0004,
    eBackupDB          = 0x0008,
    eOkToInstallNewer  = 0x0010,
    eResetAfterInstall = 0x0020,
    eCopyPrevention    = 0x0040,
    eOpenDB            = 0x8000
};
```

With the exception of `eRecord` and `eResource`, which are mutually exclusive, you can OR these values together. Table 18-7 describes the individual flags in `eDbFlags`.

Table 18-7
eDbFlags Enumerated Type

Flag	*Description*
`eRecord`	Creates a record database. This option is the default.
`eResource`	Creates a resource database.
`eReadOnly`	Indicates that the database is read-only, which is usually indicative of databases stored in ROM.
`eAppInfoDirty`	Indicates that the database's application info block has been modified.
`eBackupDB`	Sets the backup flag so the HotSync backup conduit will back up this database if no other conduit has been assigned to handle the database's creator ID.
`eOkToInstallNewer`	Indicates that the backup conduit may install a newer version of this database with a different name if this database is currently open. This is primarily used by the system to update the Graffiti shortcuts database because it is still open when the user starts a HotSync operation.
`eResetAfterInstall`	Tells the system to perform a soft reset once the sync operation that installs this database is complete.
`eCopyPrevention`	Prevents the system launcher from beaming this database to other handhelds.
`eOpenDB`	Indicates that the database is already open. This flag is for system use only; never pass this flag when creating a database.

The **SyncCreateDB** function opens the newly created database in exclusive mode for reading and writing, with private records shown; this is the equivalent of the conduit having opened the database with **SyncOpenDB** by passing the following `openMode` parameter:

```
(eDbShowSecret | eDbExclusive | eDbWrite | eDbRead)
```

Closing databases

When you have finished using a database opened with either **SyncOpenDB** or **SyncCreateDB**, you need to close the database with the **SyncCloseDB** function, whose prototype looks like this:

```
long SyncCloseDB(BYTE fHandle)
```

This function is very simple; it takes only the handle of the database you want to close as an argument. The **SyncCloseDB** function returns `0` if it successfully closed the database and destroyed its handle, or an error code if something went wrong while closing the database.

The Sync Manager allows you to have only one database open at a time. If you forget to close a database with SyncCloseDB, you will prevent other conduits from being able to open their own databases. Be sure to pair every successful SyncOpenDB or SyncCreateDB call with a call to SyncCloseDB.

Iterating Over Database Records

Most conduits need some way to walk through the records in a database, and the Sync Manager contains a variety of functions to handle this task. Before iterating over database records, though, you should call **SyncResetRecordIndex** to make sure the record index points at the first record in the database. The **SyncResetRecordIndex** function has the following prototype:

```
long SyncResetRecordIndex(BYTE fHandle)
```

The `fHandle` parameter is simply the handle to the database, as it was returned by the **SyncOpenDB** or **SyncCreateDB** function.

The Sync Manager automatically resets the record index when it first opens a database, so you can skip calling SyncResetRecordIndex if you know for sure that the database was just opened.

There are three functions for iterating over records using the Sync Manager API:

- **SyncReadNextModifiedRec** retrieves the next modified record from the database.
- **SyncReadNextRecInCategory** retrieves the next record belonging to a specific category.
- **SyncReadNextModifiedRecInCategory** retrieves the next modified record that belongs to a specific category.

The prototypes for these functions look like this:

```
long SyncReadNextModifiedRec (CRawRecordInfo &rInfo);
long SyncReadNextRecInCategory (CRawRecordInfo &rInfo);
long SyncReadNextModifiedRecInCategory (CRawRecordInfo &rInfo);
```

All of the functions use the **CRawRecordInfo** class, which is used by many other functions in the Sync Manager API that read and write record information. The definition of **CRawRecordInfo** looks like this:

```
class CRawRecordInfo
{
```

```
public:
    BYTE m_FileHandle;
    DWORD m_RecId;
    WORD m_RecIndex;
    BYTE m_Attribs;
    short m_CatId;
    int m_ConduitId;
    DWORD m_RecSize;
    WORD m_TotalBytes;
    BYTE* m_pBytes;
    DWORD m_dwReserved;
};
```

Table 18-8 describes the members of the **CRawRecordInfo** class. Notice that some members of the class have subtly different meanings, depending on whether the function using the **CRawRecordInfo** class is reading, writing, or iterating over records.

Table 18-8
CRawRecordInfo Data Members

Data Member	*Description*
`m_FileHandle`	Handle to the database, as returned by SyncOpenDB or SyncCreateDB.
`m_RecId`	Unique record ID for the record. Supply this value when calling a function that reads or deletes a record based on its record ID. Functions that iterate over records return a retrieved record's ID in this field.
`m_RecIndex`	The index of a record. Supply this value if you are calling a function that reads records by index value. Version 2.1 or later of the Sync Manager API fills in this value with the actual index of a record retrieved from the handheld.
`m_Attribs`	Record attribute flags, a combination of constants in the `eSyncRecAttrs` enumerated type. Supply this value if you are calling a function that writes a record. Functions that read a record return the record's attributes in this field.
`m_CatId`	Category index for a record. Supply this value if you are calling a function that writes a record. Functions that read a record return the record's category index in this field.
`m_ConduitId`	Currently unused.

Continued

Table 18-8 *(continued)*

Data Member	*Description*
`m_RecSize`	Actual amount of data in the record, in bytes. If you are calling a function to write a record, you must specify this value, and it should be equal to the `m_TotalBytes` value. Functions that read a record return the record's actual data size in this field. If the data size is larger than the buffer you allocated to contain the incoming record, `m_RecSize` will be larger than `m_TotalBytes`, and depending on the version of the Sync Manager API, one of two things may happen. In 2.1 or later, only `m_TotalBytes` of data are copied from the record; in versions earlier than 2.1, nothing is copied at all.
`m_TotalBytes`	Size of a buffer you have allocated for sending data to or receiving data from a record. If you are writing a record, this field should equal the size of the record in bytes. If you use a function to read a record, `m_TotalBytes` should indicate the size of the buffer pointed to by `m_pBytes`, in bytes.
`m_pBytes`	Pointer to a buffer you have allocated for reading or writing records. You must allocate this buffer before calling any Sync Manager function that reads or writes records using the CRawRecordInfo class.
`m_dwReserved`	Reserved for future use. Set this field to `NULL` before passing a CRawRecordInfo object to one of the Sync Manager functions.

The `eSyncRecAttrs` enumerated type, used in the `m_Attribs` member of **CRawRecordInfo**, has the following definition:

```
enum eSyncRecAttrs {eRecAttrDeleted  = 0x80,
                    eRecAttrDirty    = 0x40,
                    eRecAttrBusy     = 0x20,
                    eRecAttrSecret   = 0x10,
                    eRecAttrArchived = 0x08
};
```

Modifying a database while iterating over its records is not supported by the Sync Manager. Be sure to structure your conduit so it does not intersperse record modification with iteration. The exception to this rule is that the Palm OS version 2.0 and later supports deletion of records during iteration using the SyncDeleteRec function.

As an example of how to use the Sync Manager iteration functions, the following function walks through all the modified records in a database, given a handle to an open database as a parameter:

```
long WalkThroughModifiedRecs (BYTE myHandle)
{
    long retval = 0, err = 0;
```

```
    CRawRecordInfo rawRecord;

    memset(&rawRecord, 0, sizeof(rawRecord));
    rawRecord.m_FileHandle = myHandle;
    rawRecord.m_RecId      = 0;

    // Allocate memory for rawRecord data.
    rawRecord.m_TotalBytes = 0;
    rawRecord.m_pBytes = (BYTE*) new char [8000];
    if (rawRecord.m_pBytes) {
        rawRecord.m_TotalBytes = wRawRecSize;
        memset(rawRecord.m_pBytes, 0, wRawRecSize);
    } else
        return CONDERR_RAW_RECORD_ALLOCATE;

    // Read in each modified remote record one at a time.
    while (!err && !retval) {
        if (!(err = SyncReadNextModifiedRec(rawRecord))) {

            // Do something with each modified record here.

        }

        // Reset data buffer between each record retrieval.
        memset(rawRecord.m_pBytes, 0, rawRecord.m_TotalBytes);
    }

    // Free the memory allocated for the raw record.
    if (rawRecord.m_TotalBytes > 0 && rawRecord.m_pBytes)
        delete rawRecord.m_pBytes;

    return retval;
}
```

Reading and Writing Records

If you know the record ID or index of a record, you can retrieve the record directly using the **SyncReadRecordById** or **SyncReadRecordByIndex** functions, which have the following prototypes:

```
long SyncReadRecordById(CRawRecordInfo &rInfo);
long SyncReadRecordByIndex(CRawRecordInfo &rInfo);
```

Before calling these functions, you will need to fill in the `m_RecId` or `m_RecIndex` member of the **CRawRecordInfo** object that you pass to the functions. Also, just as in the example above that uses **SyncReadNextModifiedRec**, you should allocate a buffer to receive the record's data and point the **CRawRecordInfo** object's `m_pBytes` pointer to the buffer.

Writing to records may be accomplished using the **SyncWriteRec** function, whose prototype looks very similar to all the other record-handling functions in the Sync Manager API:

```
long SyncWriteRec(CRawRecordInfo &rInfo)
```

To write a record, you need to fill a buffer with the record data to write, point the **CRawRecordInfo** object's `m_pBytes` pointer at the buffer, and set `m_RecSize` and `m_TotalBytes` to the length of the record's data in bytes.

Deleting Records

The Sync Manager API provides a number of functions for deleting records from databases. The most basic function, **SyncDeleteRec**, deletes a single record from a database given a now-familiar **CRawRecordInfo** parameter:

```
long SyncDeleteRec (CRawRecordInfo &rInfo)
```

All of the Sync Manager deletion functions, whether they begin with SyncDelete or SyncPurge, immediately and permanently remove a record or records from a database instead of just marking them for deletion. Be very careful that your conduit code has properly archived deleted records before removing them from the handheld.

Aside from deleting a single record, the Sync Manager also provides records for deleting a whole lot of records at once. The **SyncPurgeAllRecs** function wipes out all the records contained in a database, given a handle to the open database:

```
long SyncPurgeAllRecs(BYTE fHandle)
```

If you want to be a little more selective in your mass deletions, you can pass a handle to an open database and a category index to **SyncPurgeAllRecsInCategory** to delete all the records contained in a specific category within that database:

```
long SyncPurgeAllRecsInCategory(BYTE fHandle, short category)
```

Even more selective, and probably the most useful, is the **SyncPurgeDeletedRecs** function, which removes only the records in the database that are marked for deletion. Typically, you would call this function after archiving records to the desktop to clean up the handheld database.

```
long SyncPurgeDeletedRecs(BYTE fHandle)
```

Maintaining a Connection

If your conduit needs to perform an action that could take a while to finish, such as retrieving data over a network, you need to ping the Sync Manager every once in a while so it will keep its connection open and not time out. The **SyncYieldCycles** function sends messages that keep the connection alive, and it also gives the HotSync application a chance to update its progress indicator and process user events, such as a click on the Cancel button. The prototype for **SyncYieldCycles** looks like this:

```
long SyncYieldCycles(WORD wMaxMiliSecs)
```

The `wMaxMiliSecs` parameter specifies the maximum number of milliseconds to spend servicing events. Because the HotSync application does not need a whole lot of time to process its events, any value higher than `1` would just be a waste of time; in fact, the current implementation of **SyncYieldCycles** just ignores this value, anyway. Just use a value of `1`.

You can call SyncYieldCycles without significantly affecting performance because there usually are no events in the HotSync application's queue to process. Palm Computing recommends calling SyncYieldCycles at least once every seven seconds during lengthy operations that might cause a timeout. Sprinkling SyncYieldCycles calls liberally through your code is a good idea.

Summary

In this chapter, you learned how to build conduits using the Palm MFC base classes, the Palm Generic Conduit base classes, and low-level Sync Manager API calls. After reading this chapter, you should understand the following:

- The Conduit Wizard provides the easiest way to start any conduit project in Visual C++.
- Every conduit must implement certain required entry points, including **OpenConduit**, **GetConduitInfo**, **GetConduitName**, and **GetConduitVersion**; in addition, every conduit should also implement the optional **CfgConduit** and **ConfigureConduit** entry points.
- Creating a conduit based on the Palm MFC Base Classes is a matter of subclassing the base monitor, table, schema, record, and link converter classes, then overriding select methods of these classes to produce the desired behavior in your conduit.

- Creating a conduit based on the Palm Generic Conduit Base Classes involves describing a desktop record format, implementing storage and retrieval of records on the desktop, implementing conversion of records between your desktop format and the generic **CPalmRecord** format, and synchronizing the application info block.
- Among other things, the Sync Manager API provides functions for registering and unregistering a conduit, opening and closing handheld databases, iterating over records, reading and writing records, deleting records, and keeping a connection to the handheld alive.

✦ ✦ ✦

CHAPTER 19

Programming in Color

In This Chapter

Determining color depth

Using color tables and palettes

Using color bitmaps

Changing user interface colors

The introduction of color capability in Palm OS 3.5 was like any other addition Palm Computing has made to the Palm OS platform: incremental. Instead of leaping ahead into uncharted territory, Palm Computing carefully added just enough new features to the operating system to support color-enabled applications, in the process maintaining excellent backward compatibility. This meticulous attention to detail means that applications written for earlier versions of the operating system still work great on color Palm OS handhelds, and as an added bonus, writing code to take advantage of Palm OS color capabilities in new applications, or grafting color onto existing programs, is very simple.

This chapter details how color works in the Palm OS, as well as the functions you need to use to add color to your applications. Throughout the chapter, you will find references to Color Test, a sample application that shows off some of the color features in Palm OS 3.5. Because the Color Test application is based entirely on Palm OS 3.5 drawing routines, it will run only on version 3.5 or later of the operating system. Figure 19-1 shows Color Test in action. Note that the actual application actually appears in vibrant color in POSE or on and actual Palm IIIc, not black and white as in this screen shot.

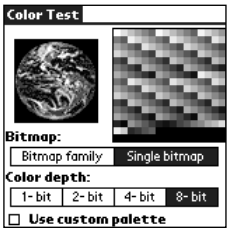


Figure 19-1: The Color Test sample application

Source code for the Color Test application is available on the CD-ROM that accompanies this book.

Determining and Setting Color Depth

Palm OS version 3.5 supports four color depths:

- 1-bit black and white, the only mode officially supported in versions of the Palm OS before 3.5. Every Palm OS device can manage 1-bit color depth.
- 2-bit grayscale, which has four colors: white, light gray, dark gray, and black. Noncolor Palm OS devices that have a non-EZ processor can use this mode, but nothing fancier.
- 4-bit grayscale, which includes sixteen levels of gray from white to black. Noncolor Palm OS devices with a DragonBall EZ processor can handle this mode.
- 8-bit color, which has a somewhat more complex palette to choose from than lesser color depths. The first part of the Palm OS color palette is composed of 216 "Web-safe" colors. These colors have red, green, and blue components at the following levels: `0x00`, `0x33`, `0x66`, `0x99`, `0xCC`, and `0xFF`. Sixteen gray shades, some already part of the first 216 colors, are also part of the palette, including `0x111111`, `0x222222`, and so on, adding another ten colors to the palette. Six named HTML colors round out the regular part of the Palm OS palette, for a total of 232 colors: `0x0C0C0C` (silver), `0x808080` (gray), `0x800000` (maroon), `0x800080` (purple), `0x008000` (green), and `0x008080` (teal). The last 24 color entries are undefined and filled with black in the release version of the operating system; on debug ROM images, these 24 slots are filled with some particularly ugly random colors that are useful for highlighting invalid color choices while testing an application.

Before attempting to use any color drawing routines, your application should first query the system to find out what its color capabilities are. To determine what color depths are available, use the **WinScreenMode** function, which has the following prototype:

```
Err WinScreenMode (WinScreenModeOperation operation,
    UInt32 *widthP, UInt32 *heightP, UInt32 *depthP,
    Boolean *enableColorP)
```

The **WinScreenMode** function is the Swiss Army knife of Palm OS color programming. Not only does **WinScreenMode** allow you to query supported color depths, but it may also modify screen dimensions and color depth, as well as enable color drawing mode. The key to the **WinScreenMode** function's versatility is its first parameter, `operation`, which may be one of the constant values described in Table 19-1.

Table 19-1
WinScreenMode Operation Constants

Constant	*Description*
`winScreenModeGet`	Retrieves the current display settings
`winScreenModeGetDefaults`	Retrieves the default display settings
`winScreenModeGetSupportedDepths`	Returns a value in the `depthP` parameter that describes the color depths supported by the hardware
`winScreenModeGetSupportsColor`	Returns `true` in the `enableColorP` parameter if it is possible to enable color drawing mode
`winScreenModeSet`	Changes display settings
`winScreenModeSetDefaults`	Changes display settings to their default values

Depending on the value of `operation`, the other parameters of **WinScreenMode** serve double duty as both input and return parameters. Table 19-2 outlines how each parameter is used given the `operation` to be performed; "in" means the parameter passes a value into the function, "out" means that **WinScreenMode** returns a value in the parameter, and "ignored" means that **WinScreenMode** ignores the parameter entirely. If you do not wish to change a particular value, or you are not interested in retrieving that value, you may pass `NULL` for any of the `widthP`, `heightP`, `depthP`, or `enableColorP` parameters.

Table 19-2
WinScreenMode Parameter Use

Operation	*widthP*	*heightP*	*depthP*	*enableColorP*
`winScreenModeGet`	out	out	out	out
`winScreenModeGetDefaults`	out	out	out	out
`winScreenModeGetSupported Depths`	in	in	out	in
`winScreenModeGetSupportsColor`	in	in	in	out
`winScreenModeSet`	in	in	in	in
`winScreenModeSetDefaults`	ignored	ignored	ignored	ignored

To find out what color depths the system can handle, pass `winScreenModeGetSupportedDepths` for the `operation` parameter of **WinScreenMode**, and supply a pointer for the `depthP` parameter. For example, the following line from the Color Test application's **StartApplication** routine sets up a global variable, `gSupportedDepths`, with the supported color depth information:

```
WinScreenMode(winScreenModeGetSupportedDepths, NULL, NULL,
              &gSupportedDepths, NULL);
```

The value returned in `depthP` is actually a binary representation of the supported color depths. Each bit in the return value represents support for a given depth if the bit is `1`, or no support if the bit is `0`. The position of a `1` in the bit field determines whether a particular color depth is supported, not the decimal numeric value represented by that bit. For example, a return value of `0x0B`, whose binary representation is `1011`, indicates support for 4-bit, 2-bit, and 1-bit color depths, because its fourth, second, and first bits are set to `1`. Another example is `0x8B` (binary `10001011`), which indicates support for 8-bit, 4-bit, 2-bit, and 1-bit drawing.

To check whether a specific color depth is supported, use a logical and (`&`) to compare `depthP` to a particular bit value. The following sample code tests to see if 8-bit color is supported, using the `gSupportedDepths` value obtained by the last example (`0x80` is `10000000` in binary):

```
if (0x80 & gSupportedDepths) {
    // 8-bit color depth is supported.
}
```

Retrieving Color Depth

If your application changes the color depth, it is good programming practice to ensure that it returns the color depth to its original state before exiting. To accomplish this, you need to retrieve the color depth when your application starts so it knows what state to restore when finished. Use the `winScreenModeGet` operation with **WinScreenMode** to retrieve the current depth, as in the following example from the Color Test application's **StartApplication** routine:

```
WinScreenMode(winScreenModeGet, NULL, NULL, &gOldDepth, NULL);
```

Because Color Test is not interested in the screen dimensions or availability of color drawing, `NULL` values are used for the `widthP`, `heightP`, and `enableColorP` parameters.

The system automatically sets the color depth back to its default when launching a new application, but Palm Computing recommends that you restore the color depth yourself before your application exits.

Setting Color Depth

Setting color depth involves passing the `winScreenModeSet` operation to **WinScreenMode**, along with the desired depth in the `depthP` parameter. As an example, the Color Test application sets the screen depth when the user selects one of the color depth push buttons. The **MainFormSelectColorDepth** function below is responsible for handling the change in color depth:

```
static void MainFormSelectColorDepth (UInt16 controlID)
{
    FormType  *form = FrmGetActiveForm();
    UInt32    depth;
    UInt32    depthHex;
    UInt16    ctlIndex;


    // If the push button is already selected, there is nothing
    // to do, so return.
    if (controlID == gCurrentPushButtonID)
        return;

    switch (controlID) {
        case MainColorDepth1BitPushButton:
            depth = 1;
            depthHex = 0x01;
            break;

        case MainColorDepth2BitPushButton:
            depth = 2;
            depthHex = 0x02;
            break;

        case MainColorDepth4BitPushButton:
            depth = 4;
            depthHex = 0x08;
            break;

        case MainColorDepth8BitPushButton:
            depth = 8;
            depthHex = 0x80;
            break;

        default:
            ErrFatalDisplay("Invalid ID");
    }

    if (depthHex & gSupportedDepths) {
        // Change color depth and refresh the screen.
        WinScreenMode(winScreenModeSet, NULL, NULL, &depth,
                      NULL);
```

```
            FrmUpdateForm(MainForm, frmRedrawUpdateCode);
            gCurrentPushButtonID = controlID;

            // Display the custom palette check box if 8-bit mode
            // is selected, or hide it when another color depth is
            // selected.
            ctlIndex = FrmGetObjectIndex(form,
                MainCustomPaletteCheckbox);
            if (depth == 8)
                FrmShowObject(form, ctlIndex);
            else
                FrmHideObject(form, ctlIndex);

        } else {
            // Alert the user that the selected depth is not
            // supported.
            FrmAlert(UnsupportedDepthAlert);

            // Set push button back to where it was.
            FrmSetControlGroupSelection(form, 1,
                                        gCurrentPushButtonID);
        }
    }
```

The only line in **MainFormSelectColorDepth** that actually performs the change in color depth is the call to **WinScreenMode**; everything else is either code to determine which push button was selected or code to let the user know that the push button selected corresponds to a color depth that the system cannot display.

Palm OS 3.0 first introduced the function ScrDisplayMode, which is the direct predecessor to WinScreenMode. The ScrDisplayMode function operates in much the same fashion as WinScreenMode and allows access to grayscale modes on devices that are running versions of the Palm OS prior to version 3.5.

Using Color Tables

To provide better performance when drawing, the system uses an indexed *color table* to store the available colors. A color table consists of a count of the number of colors in the table, followed by an array of `RGBColorType` structures, one for each color in the table. `RGBColorType` looks like this:

```
typedef struct RGBColorType {
    UInt8 index;
    UInt8 r;
    UInt8 g;
    UInt8 b;
} RGBColorType;
```

The `r`, `g`, and `b` fields in `RGBColorType` represent the level of red, green, and blue in a particular color, respectively. The `index` value is used differently by various parts of the system. Most of the drawing routines in the Palm OS use the index instead of a raw RGB value because it is faster.

Be sure not to confuse a color table, which begins with a count of its entries, with a simple array of `RGBColorType` structures. Some functions, such as BmpGetColorTable, use a full color table, whereas others, such as WinPalette, use just an array of RGB color values.

If you wish to change the current palette used for drawing in your application, use the **WinPalette** function. The most common use for **WinPalette** is to display bitmaps that use a different color table from the system default. See the section titled "Using Color Bitmaps" later in this chapter for more details about color bitmaps.

Like the **WinScreenMode** function, **WinPalette** is multipurpose, and its first parameter specifies what operation it should perform:

```
Err WinPalette (UInt8 operation, Int16 startIndex,
    UInt16 paletteEntries, RGBColorType *tableP)
```

Depending on what `operation` you specify, the `tableP` pointer to an array of `RGBColorType` structures may be used to supply values to **WinPalette** or to retrieve values from the function. Table 19-3 describes the operation constants available for **WinPalette**.

Table 19-3
WinPalette Operation Constants

Constant	*Description*
`winPaletteGet`	Retrieves the current palette; WinPalette reads entries from the palette beginning at `startIndex` and places them in `tableP`, starting at index `0`.
`winPaletteSet`	Sets entries in the current palette; WinPalette reads entries from `tableP`, beginning at index `0`, and sets those entries into the current palette, starting at `startIndex`.
`winPaletteSetToDefault`	Sets the current palette to the default system palette; during this operation, WinPalette does not use the `startIndex` or `tableP` parameters.

The paletteEntries parameter controls how many palette entries should be retrieved or set. Aside from its regular use of indicating the starting index in the palette, you may also pass the constant value WinUseTableIndexes for the startIndex parameter. Doing so tells **WinPalette** to use the individual index values of each RGBColorType structure in tableP to determine which color slot to get or set. Specifying anything other than WinUseTableIndexes for the startIndex parameter causes **WinPalette** to ignore the index values stored in tableP.

To show **WinPalette** in action, the Color Test application displays a Use custom palette check box when it is in 8-bit color depth mode. Checking this box shifts Color Test into a custom palette, which the application assembles with this code in its **StartApplication** routine:

```
if (0x80 & gSupportedDepths) {
    int  i;

    for (i = 0; i < 256; i++) {
        gCustomPalette[i].index = i;
        gCustomPalette[i].r = 0;
        gCustomPalette[i].g = 255 - i;
        gCustomPalette[i].b = 0;
    }
}
```

This custom palette is composed entirely of different shades of green. When the user taps the Use custom palette check box, Color Test calls its **MainFormSwitchPalette** routine to change the palette; **MainFormSwitchPalette** looks like this:

```
static void MainFormSwitchPalette (void)
{
    FormType  *form = FrmGetActiveForm();
    ControlType  *ctl =
        GetObjectPtr(MainCustomPaletteCheckbox);


    if (CtlGetValue(ctl) == 1) {
        // Switch to the custom palette.
        WinPalette(winPaletteSet, 0, 256, gCustomPalette);
    } else {
        // Switch to the default palette.
        WinPalette(winPaletteSetToDefault, NULL, NULL, NULL);
    }
}
```

If the check box is selected, **MainFormSwitchPalette** uses **WinPalette** with the winPaletteSet operation to change every entry in the display palette to an entry in the custom green palette. When the check box is empty, **MainFormSwitchPalette** calls **WinPalette** with the winPaletteSetToDefault operation to set the palette back to the default system palette.

Even though you can specify more than 16 million different colors using the `RGBColorType` structure, in practice the system supports only the 232 colors in the default palette. If you try to use a color that does not exist in the default palette, the system automatically finds the nearest color in the system palette and displays that color instead. This is why the Color Test sample displays only 16 shades of green when the Use custom palette check box is selected; there are only 16 greens to choose from in the system palette.

Translating RGB to Index Values

Your application may need to convert RGB values to index values in the display palette, or vice versa. Two functions exist for making these conversions: **WinRGBToIndex** and **WinIndexToRGB**. The prototypes for these functions look like this:

WinRGBToIndex

```
IndexedColorType WinRGBToIndex (const RGBColorType *rgbP);
```

WinIndexToRGB

```
void WinIndexToRGB (IndexedColorType i, RGBColorType *rgbP);
```

The `IndexedColorType` type is simply a `typedef` for an unsigned 8-bit integer:

```
typedef UInt8 IndexedColorType;
```

If the color requested in **WinRGBToIndex** is not one of the colors present in the system palette, **WinRGBToIndex** finds the nearest matching color and returns its index. If the current display palette is entirely grayscale (1-bit, 2-bit, or 4-bit), **WinRGBToIndex** tries to match the luminosity of the color to an entry in the palette. For a color palette (8-bit), **WinRGBToIndex** matches a color by looking for the nearest available RGB value. This kind of shortest-distance algorithm may not always produce a color that is the closest perceptual match to the requested color, but it is much faster than a more complex algorithm and works reasonably well, given that the default system palette contains only 232 usable colors.

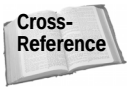
Unlike **WinRGBToIndex**, **WinIndexToRGB** can always return an exact RGB match for the requested color index.

Using Color Bitmaps

With color support added to the operating system, you should consider a few extra things when creating and displaying bitmaps in a Palm OS application. One thing to keep in mind is the color depth at which you intend to display a bitmap. If your

application is to run on different Palm OS handhelds, you should define your bitmap resources as *bitmap families* to provide support for more than one color depth in a single image.

A bitmap family is simply a collection of different versions of the same image, each version intended for a specific color depth. Both Constructor and PilRC allow for creation of bitmap families. At a minimum, an application intended for both color and noncolor devices should include a 1-bit (black and white) version of the bitmap and an 8-bit (color) version.



See Chapter 6, "Creating and Understanding Resources," for the specifics of using Constructor and PilRC to create individual bitmaps and bitmap families.

The Color Test sample application has a bitmap family that contains four bitmaps of the Earth, one for each possible color depth. Color Test also has a bitmap family containing only one 8-bit color image for comparison.

Even if there is only one image included in a form bitmap, it must be contained in a bitmap family.

The **Bitmap family** and **Single bitmap** push buttons toggle the display between showing the complete bitmap family with images in all color depths and the single color bitmap. At 8-bit color depth, both images appear identical, but if you shift through the other color depths, you will notice that the bitmap images differ slightly. This is because the system's drawing routines must convert the single bitmap image into the lower color depths to display it, and the grayscale approximations produced by the operating system do not quite match what is contained in the full bitmap family.

It is more efficient for the system to simply display an image of the appropriate color depth than to convert an image into another depth, so you should always provide bitmaps at whatever color depths an application may need to use.

It is possible to include a color table as part of a bitmap resource. There is no way to attach a color table to a bitmap in Constructor, but PilRC can do so by including the `COLORTABLE` option as part of the `BITMAPCOLOR` or `BITMAPFAMILY` directives, as in the following example:

```
BITMAPCOLOR ID 1000 "color.bmp" COLORTABLE
```

Unfortunately, including a color table with a bitmap will cause application performance to suffer, because the system will convert the bitmap's colors to the current display palette before drawing the bitmap. To keep your application running quickly, use bitmap resources that do not have attached color tables and use the **WinPalette** function to adjust the display palette to properly display the bitmaps that have different palette requirements. Better yet, if you can get away with it, make sure the bitmaps you use are already in the system palette, which will obviate the need for you to fiddle with the display palette at all.

Coloring the User Interface

Along with the main system color table, the system also keeps lists of colors to use for various user interface elements, one list for each supported color depth. These color lists are stored in the system preferences, so that an application could alter the colors the system uses to draw user interface objects. By default, Palm OS 3.5 offers no way for the user to customize user interface colors.

The system uses a number of constants to refer to each user interface color in the list. Table 19-4 lists the user interface color constants and describes where each color is used.

Table 19-4
User Interface Color Constants

Constant	*Description*
`UIObjectFrame`	Border for user interface elements, such as buttons, selector triggers, menus, and check boxes.
`UIObjectFill`	Background color for a solid user interface object.
`UIObjectForeground`	Foreground color for a user interface object; usually applied to the label on the object.
`UIObjectSelectedFill`	Background color for a selected user interface object, whether or not the object itself is solid.
`UIObjectSelectedForeground`	Foreground color for a selected user interface object.
`UIMenuFrame`	Color of the border around a menu.
`UIMenuFill`	Background color of a menu item.
`UIMenuForeground`	Text color in a menu.
`UIMenuSelectedFill`	Background color in a selected menu item.
`UIMenuSelectedForeground`	Text color in a selected menu item.
`UIFieldBackground`	Background color of an editable text field.
`UIFieldText`	Text color in an editable text field.
`UIFieldTextLines`	Underline color in an editable text field.
`UIFieldCaret`	Insertion point cursor color in an editable text field.

Continued

Table 19-4 *(continued)*

Constant	*Description*
`UIFieldTextHighlightBackground`	Background color of highlighted text in an editable text field.
`UIFieldTextHighlightForeground`	Text color of highlighted text in an editable text field.
`UIFieldFepRawText`	Text color of unconverted text in the inline conversion area of a Front End Processor (FEP), such as that used for the Japanese version of the Palm OS. If the FEP colors are identical to normal text field colors (which is usually the case on a monochrome display), unconverted text is indicated with a solid underline.
`UIFieldFepRawBackground`	Background color of unconverted text in the inline conversion area of an FEP.
`UIFieldFepConvertedText`	Text color of converted text in the inline conversion area of an FEP. If the FEP colors are identical to normal text field colors, unconverted text is indicated with a double-thick solid underline.
`UIFieldFepConvertedBackground`	Background color of converted text in the inline conversion area of an FEP.
`UIFieldFepUnderline`	Underline color used in the inline conversion area of an FEP.
`UIFormFrame`	Border and title bar color of a form.
`UIFormFill`	Background color of a form. White is usually the best color to use for `UIFormFill` because it provides the best contrast with user interface elements on the form.
`UIDialogFrame`	Border and title bar color of a modal form.
`UIDialogFill`	Background color of a modal form.
`UIAlertFrame`	Border and title bar color of an alert dialog box.
`UIAlertFill`	Background color of an alert dialog box.
`UIOK`	Color of an information icon.
`UICaution`	Color of a caution icon.
`UIWarning`	Color of a warning icon.

Keep in mind that within a table, objects use the `UIField` colors instead of the regular `UIObject` colors. Also, as of this writing, the Palm OS does not use the `UIOK`, `UICaution`, or `UIWarning` color constants.

In the color debug ROM image for Palm OS 3.5, most of the user interface color constants have a unique color assigned to them from the last 24 entries in the system palette. These color choices make the screen in POSE look like a fashion designer's worst nightmare, because the colors are fairly hideous and seem to be designed to clash with one another. However, this crime against good taste can actually be a debugging bonus if your application changes user interface colors, because you can easily distinguish default system colors from those colors that you assign yourself. If you design a color application, try it on the debug ROM to make sure it does the right thing with the user interface colors, and then test it again on the release version of the ROM, to make sure it looks okay with the actual default system colors.

Within an application, you can change the user interface colors using the **UIColor SetTableEntry** function, which has the following prototype:

```
Err UIColorSetTableEntry (UIColorTableEntries which,
    const RGBColorType *rgbP)
```

The `which` parameter is the symbolic color constant from Table 19-4 that you want to change, and `rgbP` is a standard `RGBColorType` structure containing the RGB color that you wish to assign to the specified user interface element. The **UIColorSetTableEntry** function finds the best fit in the current display palette for the requested color, and then sets the user interface color to the best-fit color.

Be sure that the drawing window is currently on-screen when using UIColor SetTableEntry. If the drawing window is off-screen, the best-fit algorithm used by UIColorSetTableEntry might pick a color that is not actually available in the current display palette.

To find out what color is currently assigned to a particular user interface element, use the **UIColorGetTableEntryIndex** function to retrieve the color's index in the display palette or the **UIColorGetTableEntryRGB** function to retrieve the color's actual RGB value in an `RGBColorType` structure. The prototypes for these two functions look like this:

UIColorGetTableEntryIndex

```
IndexedColorType UIColorGetTableEntryIndex
    (UIColorTableEntries which);
```

UIColorGetTableEntryRGB

```
void UIColorGetTableEntryRGB
    (UIColorTableEntries which, RGBColorType *rgbP);
```

When the system switches to a new application, it applies the default system user interface colors, losing any changes your application has made. If you want changes to user interface colors to stick, you will need to save them as application preferences and reset the user interface colors when your application starts up again.

Summary

In this chapter, you learned how the Palm OS 3.5 color system works and how to exploit it in your own color applications. After reading this chapter, you should understand the following:

- Palm OS 3.5 supports four color depths: 1-bit (black and white), 2-bit (4-color grayscale), 4-bit (16-color grayscale), and 8-bit (256 colors).
- Querying supported color depths, retrieving the current color depth, and setting color depth may all be accomplished with the **WinScreenMode** function.
- The **WinPalette** function allows you to change the current display palette, substituting your own colors for the default colors of the system palette.
- For maximum compatibility on various Palm OS handhelds, color bitmap resources should be created as bitmap families, containing different bitmaps for different color depths.
- The system keeps a list of user interface colors, which you may customize using the **UIColorSetTableEntry** function.

✦ ✦ ✦

CHAPTER 20

Odds and Ends

In This Chapter

Creating large applications

Using custom fonts

Creating a user interface dynamically

Localizing applications

Using the file streaming API

This chapter is a collection of useful Palm OS programming techniques that you may not need to use as frequently as those described elsewhere in this book. Many of the things mentioned in this chapter are not for the timid, and you will need to be comfortable with both the Palm OS and with your development tools before diving too deeply into this part of the book.

Creating Large Applications

The DragonBall processor used in Palm OS handhelds uses 16-bit memory addresses, which limits it to relative jumps of 32KB. If an application tries to call a function that is located more than 32KB away within the same code resource, the jump will fail. For many Palm OS applications, this is not a problem, because many applications consist of a single code resource less than 32KB in size. For larger applications, though, it may be necessary to extend the processor's jump distance, using techniques described later in the "Breaking the 32KB Barrier" section.

Even if you are able to break the 32KB jump limit, the HotSync architecture puts an absolute limit of slightly under 64KB for any resource, which includes the code resources that make up a Palm OS application. If you need to make a Palm OS application that is much larger than 64KB, you will need to break your application into multiple code resources, a process known as *segmenting* your application. The "Segmenting Applications" section later in this chapter describes how to go about breaking your program into smaller bits to work around the 64KB limit.

The official reference for segmenting applications is *CodeWarrior Targeting Palm OS*, which is available in the CodeWarrior Documentation directory of the standard Metrowerks CodeWarrior for Palm Computing platform installation.

Breaking the 32KB Barrier

An application larger than 32KB in size may generate one or more "16-bit reference out of range" errors at compile time if one function in the application calls another that is more than 32KB away in the compiled code. To work around this problem, you have several options:

- ✦ Change the link order of the application's source files.
- ✦ Rearrange your source code.
- ✦ Use the Smart code model in CodeWarrior.
- ✦ Create "code islands" linking pieces of code that are more than 32KB apart.

Changing link order

In CodeWarrior, you may be able to prevent an out of range error by rearranging the order in which CodeWarrior links your application's source files. The Segment view in the project window allows you to change the link order; Figure 20-1 shows the Segment view for a typical application.

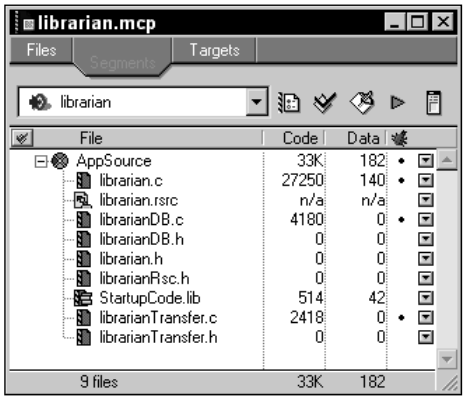


Figure 20-1: The project window's Segment view

To figure out which source files to relocate in the Segment view, take a look at the "16-bit reference out of range" errors produced during compilation. The errors will let you know which functions are attempting to call functions that are more than 32KB away. Find the source files that contain each of the offending routines and drag those source files closer to each other in the Segment view.

The same reordering process can be performed using the PRC Tools by shuffling the order of your source files in the project's makefile. It might take a bit of experimentation to come up with an order that works, however.

This technique is effective only to a certain point. The larger and more complex your application is, the greater the chance that you will not be able to resolve out of range errors by simply rearranging the link order. If that is the case, you will need to either try one of the other methods in this section or consider segmenting your application.

Rearranging your source code

It is also possible to work around some jump limitations by simply rearranging your source code within each source file. Try to group functions that call each other closer together within a source file.

Like changing the application's link order, reordering your source code is effective only to a certain point. Also, this way of circumventing the 32KB jump limit can lead to code that is hard to maintain. If you or some other developer makes changes to the source at a later date, the program may suddenly stop compiling for seemingly mysterious reasons as you shift the source code around in its file.

Using the Smart code model

In CodeWarrior, you can enable the Smart code model option, which tells the compiler to use 32-bit jumps for references that are out of range, rather than the usual 16-bit jumps. To accomplish this task, the compiler and linker have to add a fair amount of code to your application to produce each 32-bit jump, so this option can lead to code bloat very quickly if your application needs to make a lot of long-distance jumps.

If your application is hovering just under 64KB in size, this option can actually bloat the code enough that it will no longer fit in a Palm OS code resource, in which case you are better off segmenting your application.

To enable the Smart code model, open the project's target settings panel with the Edit➪ *target* settings menu command, where *target* is the name of your project. Under Code Generation in the Target Settings Panels list, select the 68k Processor list item, which will change the settings panel to look something like Figure 20-2. In the Code Model drop-down, select Smart to enable the Smart code model.

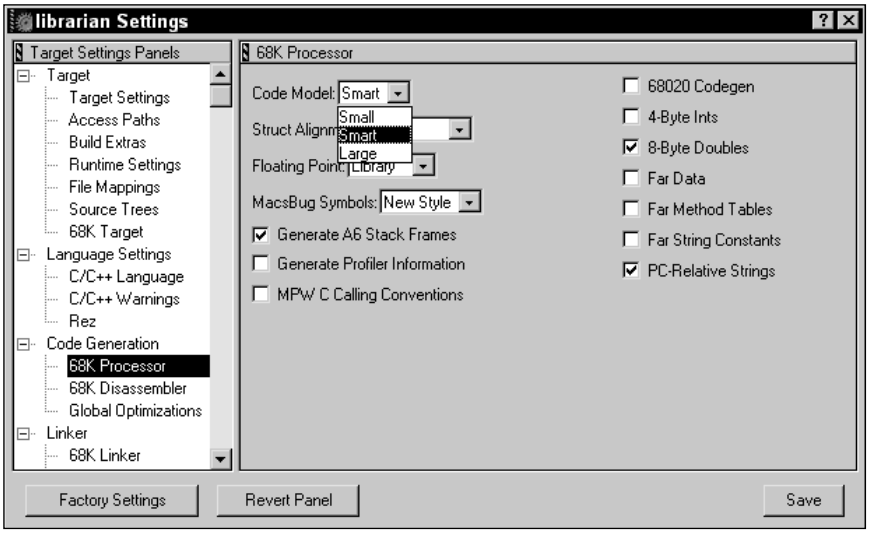


Figure 20-2: Enabling the Smart code model in the Target Settings Panel

Creating code islands

If you have only a few function calls that are out of range, you can work around the 32KB jump limit by creating "code islands," also known as "jump islands," in your application code. A code island is simply a small function, located within 32KB of both a calling function and the distant function that needs to be called, whose only purpose is to call the function that is normally out of range. Such code islands give a long-distance jump a place to "land" before moving on to the actual function the jump is trying to call.

Just as with rearranging your code, making code islands can become a maintainer's nightmare. Be sure to carefully comment both the code island and any calls to the code island to prevent you or another developer from breaking the jump by adding excess code between the calling function and the island, or between the island and the function it calls.

Segmenting Applications

If your application is larger than 64KB, you will need to build it as a *multi-segment application*. A multi-segment application is made of more than one code resource, or *segment*. Of the two major Palm OS development environments, it is much easier to create multi-segment applications with CodeWarrior than with the PRC Tools, but it may be done using either one.

Segmenting applications with CodeWarrior

If you know from the start that you are creating an enormous application to begin with, you can use CodeWarrior's project stationery for a multi-segment application. When you first create a new project, select Palm OS Multi-Segment App from the New Project dialog box, pictured in Figure 20-3, and CodeWarrior will create a multi-segment project for you, including all the linker settings required to properly build the project.

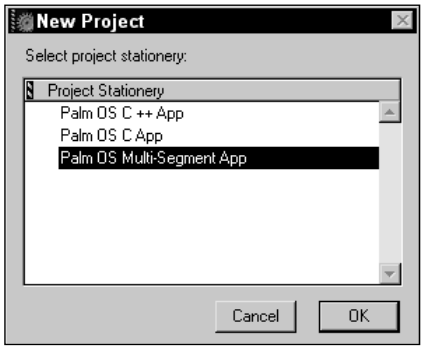


Figure 20-3: CodeWarrior's New Project dialog box

The project window's Segment view is where you can control the different code resources that make up a large project. Figure 20-4 shows the Segment view for a typical multi-segment application.

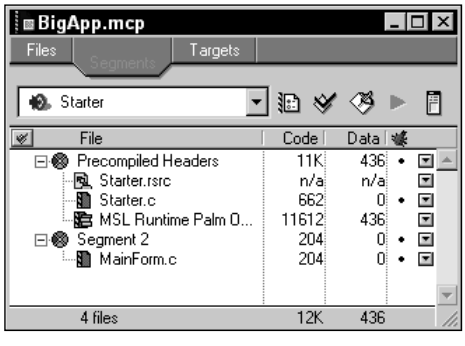


Figure 20-4: The project window's Segment view for a typical multi-segment application

The first segment in a large application is segment 0, which the linker creates to hold startup data and code that the application uses at run time to set up the other segments in the application. Segment 0 does not contain any source code that you can change, so it does not appear in the Segment view.

Segment 1, which must appear first in the Segment view, is where the source file containing the application's **PilotMain** function should go. Also, any functions called directly from **PilotMain** during anything other than a `sysAppLaunchCmdNormalLaunch` launch code must be part of the first segment.

The requirement that such functions must be in the first segment stems from the fact that the system keeps pointers to code resources in the application's global variable space. When global variables are available, which happens during a `sysAppLaunchCmdNormalLaunch` launch code, code outside segment 1 is accessible. If global variables are not available, there is no way for the system to access functions outside the first segment. Functions that PilotMain calls when global variables are not available must, therefore, be part of the first segment, because without a global variable space, there is no way for the system to call code outside segment 1.

Along with the source containing **PilotMain** and its related functions, the first segment must also contain a reference to the file `MSL Runtime Palm OS (`*xx*`).lib`, where *xx* is either `4i` or `2i`, depending on settings in the 68K Processor panel, shown earlier in Figure 20-2. If the 4-Byte Ints check box is selected, the `4i` library should be used; otherwise the `2i` library should be included.

In a multi-segment application created from the Palm OS Multi-Segment App project stationery, segment 1 is labeled in the Segment view with the non-mnemonic title "Precompiled Headers." Rest assured, whatever its name, the first segment listed in the Segment view is actually segment 1.

Later segments in the application may contain any other application code. Palm Computing recommends that you use each segment to contain code that implements related features. For example, a form's event handler and the routines it calls would fit nicely into a single segment.

To figure out exactly which source files can be located outside segment 1, do the following:

1. Comment out the code in your **PilotMain** function that handles the `sysAppLaunchCmdNormalLaunch` launch code.
2. Open the target settings dialog box and select 68K Linker from the Target Settings Panels list to display the 68K Linker panel. Check the Generate Link Map check box. Click Save and close the dialog box.
3. Rebuild your application.

CodeWarrior will create a file named *project*`.map`, where *project* is the name of your project, that shows the link map for your application. Within the link map, any file on a line beginning with `Code:` must be located in segment 1.

You can ignore functions in the link map that begin and end with a double underscore (__); these are system functions.

To add another segment to the application, select Project➪Create New Segment. The Segment Info dialog box, shown in Figure 20-5, appears.

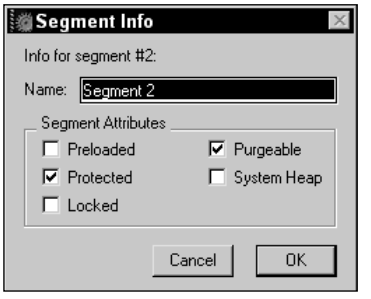


Figure 20-5: The Segment Info dialog box

Enter a name for the segment in the Name text box, and then click OK. You can add source files to a particular segment by selecting the segment in the project window, and then choosing the Project➪Add Files menu command. Moving source files from one segment to another is a simple matter of clicking on and dragging them around the project window.

To rename a segment, double-click its name in the Segment view. The Segment Info dialog box appears, allowing you to change the segment's name.

Converting a small application to a large application

As you add new features to an application during its lifetime, you may find that an application that once fit nicely into the 64KB limit has grown too large to be a single-segment application. Fortunately, it is not very difficult to convert a small application into a multi-segment application. Follow these steps:

1. Select the Targets tab in the project window to display the Target view.
2. Choose Project➪Create New Target. The New Target dialog box, shown in Figure 20-6, appears. Give the target a name, and choose the option to Clone an existing target. Pick a target from the existing project from the drop-down list, and then click OK.
3. In the Target view, pick the newly created target from the drop-down list so it is the current target for the project.
4. Select Edit➪*target* settings, where *target* is the name of the target you just created, or double-click the new target's name in the project window.

Figure 20-6: CodeWarrior's New Target dialog box

5. Select 68K Linker from the Target Settings Panels list. Make sure that the Link Single Segment check box is not selected. Click the Save button and close the dialog box.
6. Select the Segments tab in the project window to display the Segment view.
7. Replace the `StartupCode.lib` library with `MSL Runtime Palm OS (4i).lib` or `MSL Runtime Palm OS (2i).lib`, depending on your project's 4-Byte Ints setting in the 68K Processor panel.
8. Add segments to the application and rearrange the source code files in the project window until you have the code in the desired segments. Keep in mind that certain files must be located in the first segment in the list: the run-time library, any source file containing **PilotMain**, and any source file containing routines that **PilotMain** calls when processing any launch code other than `sysAppLaunchCmdNormalLaunch`. If **PilotMain** and its associated function calls are located in the same source file with other application code, you may need to create a new source file and separate the code into the new file before you can rearrange the link order.

Segmenting applications with the PRC Tools

Creating multi-segment applications with the PRC Tools is somewhat more difficult than it is in CodeWarrior, but it is possible.

The PRC Tools documentation refers to different segments as *code sections*, which is more in keeping with the standard GCC `section` function attribute. This part of the book will continue to call them segments to avoid muddying the waters any further, but if you read through the PRC Tools documentation and see references to code sections, note that segments are being discussed.

Three things are required to build a multi-segment application with the PRC Tools:

- A project *definition file*, which lists the different segments in a `multiple code` clause
- A `section` attribute annotation in the declaration of each function that will be outside the first segment

- ✦ An assembly language stub file that defines pointers to functions in segments outside segment `1`

A definition file is simply a text file with a `.def` extension that contains information about various properties of the project as a whole. The definition file must be the first file you pass to `build-prc` when compiling the application. Although there are many directives you can add to a definition file, the only one that is necessary for segmenting a large application is `multiple code`. The syntax for the `multiple code` clause looks like this:

```
multiple code { section ...}
```

Within the curly braces, list each segment past the first in your application, separated by spaces. For example, the following line declares two extra sections, one for an application's main form and one for its edit form:

```
multiple code { mainform editform }
```

The PRC Tools store only the first eight characters of a section name. If you use larger section names, they will be truncated, which can lead to all kinds of pain; keep section names at eight or fewer characters.

The `multiple code` clause lets `build-prc` know that there are extra segments in the application and what those segments are called. Now you need to mark individual files in your source code so `build-prc` knows that they belong to these extra segments. To do this, you must use the `section` function attribute in the declarations of functions that live outside the first segment. A function declaration with the `section` attribute looks like this:

```
void MyFunc (void) __attribute__ ((section ("section")));
```

The *section* value in double quotes is the name of the segment that the function belongs to, and that name should match one of the segment names from the definition file. To make it easier to define many functions as part of an extra segment, you can `#define` macros to add to the end of function declarations:

```
#define MAIN_SECTION __attribute__ ((section ("mainform")))
#define EDIT_SECTION __attribute__ ((section ("editform")))

Boolean MainFormHandleEvent(EventPtr event) MAIN_SECTION;
   void MainFormInit(FormPtr form) MAIN_SECTION;

...

Boolean EditFormHandleEvent(EventPtr event) EDIT_SECTION;
   void EditFormInit(FormPtr form) EDIT_SECTION;
   void EditFormSaveData(void) EDIT_SECTION;
```

Once you have all the functions in extra segments marked, you need to make an assembly language stub file that contains pointers to all these functions. Creating the assembly stub file is done using the same definition file you created for `build-prc`, only this time you need to pass it to a tool called `multigen`, which also generates a linker script to properly link the extra-segment functions into the application.

First, to generate the assembly code and linker stub, call `multigen` with the `.def` file as its sole argument. The makefile rule should look something like this:

```
myapp-sections.s myapp-sections.ld: myapp.def
        multigen myapp.def
```

The `multigen` tool generates two files, *myapp*`-sections.s` and *myapp*`-sections.ld`, where *myapp* is the name of your application. Compile the *myapp*`_sections.s` file:

```
myapp-sections.o: myapp-sections.s
        m68k-palmos-gcc -c myapp-sections.s
```

This will result in a *myapp*`-sections.o` file, which you should link into your application, along with all the other object files that make up the application. At this time, you should also pass in the *myapp*`_sections.ld` linker script so the linker can place the extra assembly code at the proper addresses:

```
OBJS = ...list of object files... myapp-sections.o

myapp: $(OBJS) myapp-sections.ld
        m68k-palmos-gcc -o myapp $(OBJS) myapp-sections.ld
```

Along with the rules just outlined, you should also remove the `m68k-palmos-obj-res` rule and any mention of the interim `.grc` resource files. Instead, you should build the application's `.prc` file by passing the compiled executable directly to `build-prc`:

```
myapp.prc: myapp
        build-prc myapp.prc "My App" LFlb myapp.def myapp *.bin
```

Listing 20-1 shows a generic makefile for compiling a multi-segment application. All you need to do to customize this makefile is replace the variable definitions at the top with appropriate names for your own application. In particular, make sure you change the `APPID` variable to a unique creator ID that you have registered with Palm Computing.

Listing 20-1: **Generic multi-segment makefile**

```
VERSION         = 1.0
APP             = myapp
```

```
ICONTEXT        = "My App"
APPID           = strt
RCP             = $(APP).rcp
PRC             = $(APP).prc
DEF             = $(APP).def
SRC             = $(APP).c $(APP)Section2.c $(APP)Section3.c
SECTIONS        = $(APP)-sections

CC              = m68k-palmos-gcc
PILRC           = pilrc
BUILDPRC        = build-prc
MULTIGEN        = multigen

# Uncomment this if you want to build a GDB-debuggable version
#CFLAGS = -O2 -g
CFLAGS = -O2

all: $(PRC)

$(PRC): $(APP) bin.stamp
        $(BUILDPRC) $(PRC) $(ICONTEXT) $(APPID) $(DEF) *.bin
        ls -l *.prc

$(APP): $(SRC:.c=.o) $(SECTIONS).ld
        $(CC) $(CFLAGS) -o $@ $^

bin.stamp: $(RCP)
        $(PILRC) $^ $(BINDIR)
        touch $@

%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@
#               touch $<
# Enable the previous line if you want to compile EVERY time.

$(SECTIONS).o: $(SECTIONS).s
        $(CC) $(CFLAGS) -c $< -o $@

$(SECTIONS).s $(SECTIONS).ld: $(DEF)
        $(MULTIGEN) $(DEF)

depend dep:
        $(CC) -M $(SRC) > .dependencies

clean:
        rm -rf *.o $(APP) *.bin *.stamp *.s *.ld

veryclean: clean
        rm -rf *.prc *.rcp *.bak
```

Adding Custom Fonts to Applications

The fonts that come with the Palm OS are normally sufficient for most applications, but they do have their limitations. For one thing, the default fonts all have proportional spacing, which makes them particularly unsuitable for displaying things such as computer source code, which is much easier to read in a monospaced font. If you want some kind of decorative font for use in a game, you are also out of luck with the default fonts, because they are very simple sans-serif fonts, designed to be easy to read and, at the same time, occupy as little screen space as possible.

You can create your own custom fonts for the Palm OS, however. A font is a resource of type `NFNT`. On Palm OS 3.0 or later, you can use an `NFNT` resource in an application by calling **FntDefineFont**. The **FntDefineFont** function's prototype looks like this:

```
Err FntDefineFont (FontID font, FontPtr fontP)
```

The `font` parameter is an application-specific identification number for the font. Values less than 128 are reserved for system use, so a custom font must have an ID of 128 or greater. The `fontP` parameter takes a pointer to the locked font resource. As an example, the following lines of code lock a custom font resource and set it up for use in an application:

```
#define fntCustom 128

FontType  *fontCustom;

fontCustom = MemHandleLock(DmGetResource('NFNT',
                           MyCustomFont));
FntDefineFont(fntCustom, fontCustom);
```

Once you have called **FntDefineFont** to define the custom font's ID, you may use that font ID as you would any other font ID. When the application exits, the system uninstalls the font, so your application will need to reinstall it by calling **FntDefineFont** each time the application runs.

You must keep the custom font resource locked until the application either quits or no longer needs to use the font.

On versions of the Palm OS prior to 3.0, the **FntDefineFont** function does not exist, but you can still use custom fonts with a little judicious hacking. By setting the user interface global variable `UICurrentFontPtr`, you can trick the system into using your custom font:

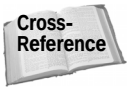
```
void *fontOld;
void *fontCustom;
```

```
// Save the current font.
fontOld = UICurrentFontPtr;

fontCustom = MemHandleLock(DmGetResource('NFNT',
                           MyCustomfont));
UICurrentFontPtr = fontCustom;

// Perform any drawing operations that use the custom font.

// Unlock the pointer to the custom font.
MemPtrUnlock(fontCustom);

// Restore the original font.
UICurrentFontPtr = fontOld;
```

There are limitations to using custom fonts. Normally you may use a custom font only with the various Palm OS text-drawing functions, not as part of any form elements, because the font is not available at build time. The exception to this is if you use PilRC to build your application's resources, and then only in an application that runs on Palm OS 3.0 or later. See below for more details about using custom fonts to build forms in PilRC.

Creating a Custom Font

As of this writing, CodeWarrior's Constructor tool is not able to create custom fonts, but PilRC can. There are also a few shareware and freeware applications, for both Windows and the Mac OS, for creating new fonts on the desktop in a graphical environment, or for converting existing desktop fonts into resources that you can compile into an application using either CodeWarrior or the PRC Tools.



See Appendix B, "Finding Resources for Palm OS Development," for more information about where to find font creation tools.

Unfortunately, even with a good desktop font creation tool, Constructor still cannot use custom fonts in form elements. PilRC, on the other hand, is perfectly capable of building forms that contain custom fonts, provided the font definition comes first in the `.rcp` file, before any use of the custom font in a form. Another requirement for build-time use of custom fonts in PilRC is that the application whose forms contain the custom fonts must be running on Palm OS 3.0 or later.

The PilRC directive for creating a font resource looks like this:

```
FONT ID <resourceID> FONTID <fontID> <font file>
```

The `fontID` should be the same font ID number that you pass to the **FntDefineFont** function (a number between 128 and 255, inclusive), and `font file` specifies a text file that defines the font. For example, the following line creates a font resource from a font definition file called `myfont.txt`:

```
FONT ID 1000 FONTID 128 "myfont.txt"
```

A font definition file is a standard ASCII text file, consisting of some header information followed by the definitions for individual characters, or glyphs, that make up the font. The header for a font looks like this:

```
fontType 36864
ascent <ascent value>
descent <descent value>
fRectWidth <font width>
fRectHeight <font height>
```

The `fontType` is a magic number that is present in all the default Palm OS fonts. David Turnbull, the author of the font compiler in PilRC, created the compiler by reverse-engineering the existing default Palm OS fonts, so the `fontType` number is something of a mystery; rest assured, your font should work fine if you include `fontType 36864` at the head of the font definition.

Both `ascent` and `descent` are values that you do not have to take on faith. The `ascent` value is the number of pixels that make up the part of each glyph above the font's baseline, where the bottoms of most characters are. Most characters in a font sit directly on or above the baseline, but some, such as *q* and *y*, have a part (known in conventional typography as a descender) that dips below the baseline. The `descent` value specifies how many pixels are below the baseline in each glyph. A font's total height is equal to the ascent plus the descent.

The `fRectWidth` and `fRectHeight` values are entirely optional. These two header lines define the width and height of the font, respectively. If you omit `fRectWidth`, PilRC uses the width of the widest character in the font. If you omit `fRectHeight`, PilRC uses the height of the first character in the font.

As an example, the following header section defines a font with an ascent of 9 and a descent of 2, for a total height of 11:

```
fontType 36864
ascent 9
descent 2
```

Following the header are the definitions for individual glyphs in the font. Each glyph definition begins with the word `GLYPH`, followed by either the ASCII number of the character this glyph represents, or the actual character itself, surrounded by single

quotes. For example, the following `GLYPH` statements are equivalent, both representing a space character:

```
GLYPH 32

GLYPH ' '
```

Tip To define a glyph for the single quote character (`'`), use `GLYPH 39`; using `GLYPH ''' ` will not work.

Following the `GLYPH` line are several lines that define what the bitmap image of the glyph looks like. A bitmap line consists of hyphen (`-`) or period (`.`) characters to represent pixels that are turned off, and any other characters to represent pixels that are turned on. For example, the following glyph defines a letter *A*:

```
GLYPH 'A'
----------
----#-----
---#-#----
---#-#----
--#---#---
--######--
-#-----#--
-#-----#--
###---###-
----------
----------
```

Keep in mind that all the glyph definitions in your font should be exactly the same number of lines high. Even a tiny character, such as a period, needs to have empty pixels above and below it so its height is the same as that of other glyphs in the font. For example, the following period glyph would work with the *A* defined previously:

```
GLYPH '.'
---
---
---
---
---
---
---
##-
##-
---
---
```

Tip To make a font more readable, include a pixel of blank space to the right of each glyph to keep the characters from running into each other. You should also leave a pixel of blank space at the top of each character, so descending parts of glyphs do not collide with the tops of tall letters on the next line of text.

Creating User Interface Dynamically

For most applications, the user interface elements on a form are static; there is no need to move them around at run time. Even if you have an element that you want to hide or show in response to user input, the best way to accomplish this is to include the element as part of a form's normal resources, and then make it disappear and reappear at run time by invoking the **FrmHideObject** or **FrmShowObject** functions. Vanishing form elements can be very confusing to a new user and usually make for a bad user interface.

If your application does need to create new forms at run time or add elements to existing forms, note that Palm OS version 3.0 and later offers functions for creating user interface dynamically—while an application is running. An example of an application where this might be useful would be a database program that allows the user to create custom input forms.

The **FrmNewForm** function allows you to create a new form, and it has the following prototype:

```
FormType *FrmNewForm (UInt16 formID, const Char *titleStrP,
    Coord x, Coord y, Coord width, Coord height, Boolean modal,
    UInt16 defaultButton, UInt16 helpRscID, UInt16 menuRscID)
```

As a return value, **FrmNewForm** gives you a pointer to the newly created form, or `0` if the call did not succeed. The most common reason that **FrmNewForm**, or any of the dynamic UI functions, might fail is lack of available memory.

Unlike a normal form resource, there is no need to call FrmInitForm with a dynamically created form.

Table 20-1 describes each of the parameters to **FrmNewForm**.

Table 20-1
FrmNewForm Parameters

Parameter	*Description*
`formID`	ID to assign to the form.
`titleStrP`	Pointer to a string to use for the form's title.
`x`	Horizontal coordinate of the form's upper-left corner.

Parameter	*Description*
`y`	Vertical coordinate of the form's upper-left corner.
`width`	Width of the form in pixels; must be from `1`–`160`, inclusive.
`height`	Height of the form in pixels, must be from `1`–`160`, inclusive.
`modal`	If `true`, the form ignores pen events outside its borders.
`defaultButton`	ID of the form's default button, which the system simulates tapping if the user switches applications while the form is displayed.
`helpRscID`	ID of a string resource that contains help text for the dialog box; only modal dialog boxes can have help text.
`menuRscID`	ID of the menu bar resource that should be attached to the form. Note that you cannot dynamically create menu resources, so if you want menus for a dynamically generated form, you must attach the form to pre-existing menu resources.

When creating any form or UI object dynamically, be careful not to use an ID that already exists for another object. A good way to avoid reusing ID numbers is to reserve a block of numbers that are used only for dynamic UI. When the application starts, set a global variable to the first number in this pool of ID numbers, and any time you create a new object, increment the variable so you will never use the same number twice. Here is a quick example:

```
#define DYNAMIC_UI_START 6000

UInt16 gNextID = DYNAMIC_UI_START;

// When creating a new object, increment gNextID:
newLabelPtr = FrmNewLabel(form, gNextID++, text, x, y,
                          stdFont);
```

You may add user interface elements to forms that you create dynamically, or to existing static forms, using a variety of functions. The only UI element that you cannot create dynamically is a table. Also, some of the routines for creating user interface elements are restricted to certain versions of the Palm OS. Table 20-2 lists the functions for creating individual form elements, along with the earliest version of the Palm OS that supports the function and what element each function creates.

Table 20-2
Dynamic UI Element Functions

Function	*Minimum Palm OS Version*	*UI Element(s)*
CtlNewControl	3.0	Controls: buttons, push buttons, repeating buttons, check boxes, pop-up triggers, or selector triggers
CtlNewGraphicControl	3.5	Graphic controls: buttons, push buttons, repeating buttons, pop-up triggers, or selector triggers
CtlNewSliderControl	3.5	Slider
FldNewField	3.0	Text field
FrmNewBitmap	3.0	Form bitmap
FrmNewGadget	3.0	Gadget
FrmNewGsi	3.5	Graffiti shift indicator
FrmNewLabel	3.0	Label
LstNewList	3.0	List

As an example of a typical member of this group of functions, here is the prototype for **CtlNewControl**:

```
ControlType *CtlNewControl (void **formPP, UInt16 ID,
    ControlStyleType style, const Char *textP, Coord x,
    Coord y, Coord width, Coord height, FontID font,
    UInt8 group, Boolean leftAnchor)
```

All of the UI creation functions share the same first parameter: `formPP`. The `formPP` parameter is a pointer to a pointer to the form where the new element should be installed. Unlike many pointers to pointers in the Palm OS, `formPP` is not actually a handle. To accommodate the addition of another user interface element, the system may need to move the entire form structure in memory, which makes the original value of `formPP`, as you pass it to the function, invalid. Fortunately, if the dynamic UI functions need to move the form, they return the form's new location in the `formPP` parameter. Be sure to always use the new value returned in `formPP`, and discard whatever value you originally passed in for the `formPP` parameter.

A pair of other parameters are shared by all the dynamic UI functions; `x` and `y` always specify the coordinates of the upper-left corner of the user interface element, relative to the form that contains it. Most of the functions also have `width`

and `height` parameters, which contain the width and height of the element, in pixels. Many of the functions also have an `ID` or `id` parameter, which is where you should specify the ID number you will use to refer to the element elsewhere in your application.

When creating a control with **CtlNewControl** or **CtlNewGraphicControl**, you need to specify exactly what type of control you want by supplying the `style` parameter. Values for `style` come from the `ControlStyleType` enumerated type, which looks like this:

```
enum controlStyles {
    buttonCtl,
    pushButtonCtl,
    checkboxCtl,
    popupTriggerCtl,
    selectorTriggerCtl,
    repeatingButtonCtl,
    sliderCtl,
    feedbackSliderCtl
};
typedef enum controlStyles ControlStyleType;
```

Not all of the members of `ControlStyleType` are available for use with CtlNewControl or CtlNewGraphicControl. In particular, `sliderCtl` and `feedbackSliderCtl` may be used only with the CtlNewSliderControl function, and CtlNewGraphicControl cannot have a `style` type of `checkboxCtl`.

Before closing a form containing dynamic UI elements, you need to call **FrmRemoveObject** to remove each dynamic object. The **FrmRemoveObject** function has the following prototype:

```
Err FrmRemoveObject (FormType **formPP, UInt16 objIndex)
```

The `formPP` parameter to **FrmRemoveObject** works in the same fashion as the `formPP` parameter to the UI element creation functions. This `formPP` is also not a handle, and you should discard whatever value you pass for `formPP` in favor of the value returned in `formPP` by the **FrmRemoveObject** function. When **FrmRemoveObject** removes a UI element, it does not free any memory associated with the object itself, such as the string data attached to a text field. The **FrmRemoveObject** function does shrink the memory chunk allocated to a form's data structure, because the function frees the memory occupied by the object within the form structure itself.

Also keep in mind that the **FrmRemoveObject** function's `objIndex` parameter requires the index of an object, not its ID number. As with any user interface object, you can retrieve the index by passing the object's ID to the **FrmGetObjectIndex** function.

For more efficient removal of several form objects, remove them in descending index order; that is, call FrmRemoveObject on objects with higher index values before removing objects with low index values. Removing objects in this order reduces the amount of shuffling that FrmRemoveObject must do when filling in the hole left by the removed object's data, because higher-indexed objects are at the end of the form's data structure.

A common problem when programming dynamic user interface is accidentally using invalid pointers to controls and forms. During debugging, you can use **CtlValidatePointer** and **FrmValidatePtr** to make sure that pointers are valid before trying to use them. The prototypes for these two functions look like this:

```
Boolean CtlValidatePointer (const ControlType *controlP);

Boolean FrmValidatePtr (const FormType *formP);
```

Both functions return `true` if the pointer you pass to them is a valid pointer to a control or a form. You should use these functions only for debugging, though; leaving them in a released application adds bloat to the code.

Localizing Applications

Creating applications that can display text in multiple languages is easily one of the most challenging tasks in software development. Not only do you need to be careful to put display text only in the application's resources instead of hard-coding it, but there are also major problems in creating an application that can support both the standard ASCII text sufficient for display of most Western languages and the multiple-byte character encoding used for most Asian languages. Besides text differences, different countries also format numbers and dates differently, further adding to the confusion.

Fortunately, the folks at Palm Computing have provided tools and functions within the Palm OS to make the often arduous task of localizing an application much easier. The text and international managers, available in most versions of the Palm OS since version 3.1, offer functions for working with localized strings and characters.

Using the Text and International Managers

Since Palm OS 3.1, Palm Computing has included the text and international managers as part of the operating system. Before using any text or international manager functions, check for the existence of the international manager with the **FtrGet** function:

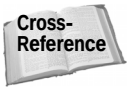
```
UInt32  value;

Err error = FtrGet (sysFtrCreator, sysFtrNumIntlMgr, &value);
```

If the international manager is installed, `value` will be a non-zero value, and the error return value will be `0`. The text and international managers are inseparable companions, so once you ascertain that the international manager exists, you can be certain that the text manager is also available.

The international manager detects the character encoding used by the system and uses this information to set up the text manager. Application code hardly ever interacts with the international manager directly; the manager operates in the background most of the time.

The text manager provides functions for manipulating strings and character data, regardless of what character encoding is in use on the system. In particular, text manager functions allow you to safely deal with both single- and multiple-byte character sets using the same set of text-handling functions. If you stick with using text manager calls instead of directly modifying text data, you should not need to change your application's code to handle different styles of character encoding.

Many of the "functions" included in the text manager are multiple-byte aware versions of the C standard library character macros. For example, the text manager's **TxtCharIsDigit** macro duplicates the effects of the **isdigit** macro, but unlike **isdigit**, **TxtCharIsDigit** knows about multiple-byte character encoding, so it can properly determine whether or not a multiple-byte character is a digit.



For more information about the various character macros, see the "Using Character Macros" section in Chapter 10, "Programming System Elements."

Comparing and finding text

Two particularly useful functions in the text manager are **TxtCompare** and **TxtCaselessCompare**, which allow you to compare the contents of two text buffers. The **TxtCompare** function is a case-sensitive comparison, and **TxtCaselessCompare** ignores case. The prototypes for **TxtCompare** are nearly identical; here is the prototype for **TxtCompare**:

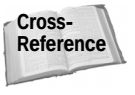
```
Int16 TxtCompare (const Char* s1, UInt16 s1Len,
    UInt16* s1MatchLen, const Char* s2, UInt16 s2Len,
    UInt16* s2MatchLen)
```

Both functions return a value less than zero if the text in `s1` occurs before the text in `s2` alphabetically. The return value is greater than zero if `s2` comes before `s1` in alphabetical order, and the return value is exactly `0` if both text buffers are equal. Along with the pointers to the two text buffers, you should also supply the lengths of the respective buffers in the `s1Len` and `s2Len` parameters.

Both `s1` and `s2` must point to the start of a valid character, which means either the first byte of a multiple-byte character, or a single-byte character. Pointing to the middle of a multiple-byte character will lead to unpredictable (and messy) results. This is true of text input parameters for any text manager function.

The **TxtCompare** and **TxtCaselessCompare** functions return the length in bytes of the text that matches exactly between `s1` and `s2` in the `s1MatchLen` and `s2MatchLen` parameters. The byte lengths in `s1MatchLen` and `s2MatchLen` might differ, because some character encodings, such as Shift-JIS for the Japanese language, can represent the same character as both single- and multiple-byte characters. If you are not interested in the amount of matching text between the two text buffers, pass `NULL` for `s1MatchLen` and `s2MatchLen`.

Another function in the text manager is **TxtFindString**, which finds an occurrence of one string inside another. The search that **TxtFindString** performs is case-insensitive, and it makes a fine multiple-byte aware replacement for **FindStrInStr** when implementing the global find facility.



For more information about implementing global find, take a look at the "Implementing the Global Find Facility" section of Chapter 13, "Manipulating Records."

The prototype for **TxtFindString** looks like this:

```
Boolean TxtFindString (const Char* inSourceStr,
    const Char* inTargetStr, UInt32* outPos, UInt16* outLength)
```

The `inSourceStr` parameter should point to the string that should be searched, and `inTargetStr` should point to the string you want to find in `inSourceStr`. If **TxtFindString** finds `inTargetStr` within `inSourceStr`, the function returns `true`; otherwise, **TxtFindString** returns `false`. On a successful find, **TxtFindString** returns the byte offset of the beginning of `inTargetStr` within `inSourceStr` in the `outPos` parameter, and the length of the matching text in `outLength`. The **TxtFindString** function sets the values of `outPos` and `outLength` to `0` on an unsuccessful search.

Modifying text

The **TxtTransliterate** function allows you to convert all the characters in a text buffer from one form to another. For example, **TxtTransliterate** may be used to change all the characters in a text buffer to uppercase. The prototype for **TxtTransliterate** looks like this:

```
Err TxtTransliterate (const Char* inSrcText,
    UInt16 inSrcLength, Char* outDstText,
    UInt16* ioDstLength, TranslitOpType inOp)
```

The `inSrcText` parameter points to the source text buffer you want to modify, and `inSrcLength` specifies the length of that buffer in bytes. You also need to supply a pointer to an output buffer via `outDstText` and specify the maximum length of that buffer in `ioDstLength`. When **TxtTransliterate** returns, it modifies the value in `ioDstLength` to reflect the actual length of the transformed text in `outDstText`.

You control exactly what **TxtTransliterate** does to the text in `inSrcText` by specifying an operation code in the `inOp` parameter. Operation codes are specific to the character encoding currently in use on the handheld, but two operations are always available: `translitOpUpperCase`, which converts all the text to uppercase, and `translitOpLowerCase`, which converts the text to lowercase. Both case conversion operations are useful for reducing a string to a single case for faster case-insensitive comparison with another string. The Palm OS header file `TextMgr.h` defines a base number for operations that are character encoding–specific:

```
#define translitOpCustomBase 1000
```

Within the header files for specific character encodings, other transliteration operations use `translitOpCustomBase` as a base value. For example, the following operations are defined in `CharShiftJIS.h` for Shift-JIS encoding:

```
#define translitOpFullToHalfKatakana (translitOpCustomBase+0)
#define translitOpHalfToFullKatakana (translitOpCustomBase+1)
#define translitOpFullToHalfRomaji   (translitOpCustomBase+2)
#define translitOpHalfToFullRomaji   (translitOpCustomBase+3)
#define translitOpKatakanaToHiragana (translitOpCustomBase+4)
#define translitOpHiraganaToKatakana (translitOpCustomBase+5)
#define translitOpCombineSoundMark   (translitOpCustomBase+6)
#define translitOpDivideSoundMark    (translitOpCustomBase+7)
#define translitOpRomajiToHiragana   (translitOpCustomBase+8)
#define translitOpHiraganaToRomaji   (translitOpCustomBase+9)
```

Along with the regular operation codes, you may also combine the `translitOpPreprocess` mask constant with any code using the bitwise OR operator (`|`). This causes **TxtTransliterate** to find out how much space is required for the transformed string without actually placing the string in the output buffer; the space required is returned in the `ioDstLength` parameter. If you are not sure whether you have enough space allocated to contain the transliterated text, call **TxtTransliterate** using the `translitOpPreprocess` mask before making the actual call. Here is an example:

```
UInt16  buf1Length, buf2Length, outputSize;
Char    *buffer1, *buffer2;


// Point buffer1 at the source text, allocate an output buffer,
// and point buffer2 at the output buffer.

buf1Length = StrLen(buffer1);
buf2Length = StrLen(buffer2);
outputSize = buf2Length;

TxtTransliterate(buffer1, buf1Length, &buffer2, &outputSize,
    translitOpLowerCase | translitOpPreprocess);
```

```
if (outputSize > buf2Length) {
    // Increase the size of the buffer2 buffer so it can hold
    // outputSize bytes of data.
}

TxtTransliterate(buffer1, buf1Length, &buffer2, &outputSize,
    translitOpLowerCase);
```

Retrieving characters from a text buffer

It can be difficult to do something as simple as retrieving a single character from a text buffer if the buffer contains multiple-byte characters. This phenomenon also makes it difficult to iterate through a text buffer a character at a time, which is necessary to perform many kinds of text modification. With the proper functions from the text manager, this becomes much easier.

The **TxtGetChar** function retrieves a single character from a text buffer, given an offset into the buffer in bytes. Here is the prototype for **TxtGetChar**:

```
WChar TxtGetChar (const Char* inText, UInt32 inOffset)
```

As with other text manager functions, you are responsible for ensuring that the offset specified in `inOffset` points to the beginning of a valid character and not to the middle of a multiple-byte character.

If you need to iterate through a text buffer, use the **TxtGetNextChar** and **TxtGetPreviousChar** functions, which look like this:

```
UInt16 TxtGetNextChar (const Char* inText, UInt32 inOffset,
    WChar* outChar);

UInt16 TxtGetPreviousChar (const Char* inText, UInt32 inOffset,
    WChar* outChar);
```

These functions start at the offset specified in `inOffset` and return either the next or previous character in the `outChar` parameter. The function return value is the size in bytes of the appropriate character.

The TxtGetPreviousChar function can be slower than TxtGetNextChar, because TxtGetPreviousChar sometimes has to work its way backward through the text buffer byte by byte until it finds an unambiguous beginning of a multi-byte character.

If you want to know only the size of the next or previous character, pass `NULL` for the `outChar` parameter in **TxtGetNextChar** or **TxtGetPreviousChar**. Alternatively, you can use the **TxtNextCharSize** and **TxtPreviousCharSize** macros as follows:

```
TxtNextCharSize (inText, inOffset)
TxtPreviousCharSize (inText, inOffset)
```

Like the **TxtGetNextChar** and **TxtGetPreviousChar** functions, **TxtNextCharSize** and **TxtPreviousCharSize** return a `UInt16` value indicating the size in bytes of the appropriate character.

The nature of multiple-byte characters also makes it difficult to find where you need to truncate a text buffer to fit within a certain size. Use the **TxtGetTruncationOffset** function to determine where a text buffer should be chopped so it will fit a given number of bytes of memory:

```
UInt32 TxtGetTruncationOffset (const Char* inText,
    UInt32 inOffset)
```

The return value from **TxtGetTruncationOffset** is the offset into `inText` where the buffer may be safely truncated at an intercharacter boundary; the return value is always less than or equal to the `inOffset` parameter's value.

Finally, the **TxtWordBounds** function is an easy way to find the beginning and end of an actual word in the middle of a text buffer, given an offset into the buffer that points to the beginning of a valid character. The prototype for **TxtWordBounds** looks like this:

```
Boolean TxtWordBounds (const Char* inText, UInt32 inLength,
    UInt32 inOffset, UInt32* outStart, UInt32* outEnd)
```

The `inText` parameter points to the start of the text buffer, `inLength` indicates the length of the buffer, and `inOffset` is the offset around which you want to find a word. The function returns `true` if it finds a word at `inOffset`, or false if `inOffset` is at a punctuation or whitespace character. For example, if you have a string (in ASCII encoding) that contains the string `Find me a word.`, passing an offset of `6` to **TxtWordBounds** will result in `outStart` and `outEnd` values that point to the start and end of the word `me`, because the offset of `6` points to the `e` in `me`.

Determining character encoding

The text manager also contains functions for determining the minimum required encoding to represent a particular character or string. The **TxtCharEncoding** function finds the minimum required encoding system necessary to represent a given character, and **TxtStrEncoding** finds the encoding required to represent a string. Prototypes for these functions look like this:

TxtCharEncoding

```
CharEncodingType TxtCharEncoding (WChar inChar);
```

TxtStrEncoding

```
CharEncodingType TxtStrEncoding (const Char* inStr);
```

The CharEncodingType return value from these functions is an enumerated type defined in the Palm OS header file TextMgr.h as follows:

```
typedef enum {
    charEncodingUnknown = 0, // Unknown to this version of
                             // the Palm OS

    charEncodingAscii,       // ISO 646-1991
    charEncodingISO8859_1,   // ISO 8859 Part 1
    charEncodingPalmLatin,   // Palm OS version of CP1252
    charEncodingShiftJIS,    // Encoding for 0208-1990 + 1-byte
                             //  katakana
    charEncodingPalmSJIS,    // Palm OS version of CP932
    charEncodingUTF8,        // Encoding for Unicode
    charEncodingCP1252,      // Windows variant of 8859-1
    charEncodingCP932        // Windows variant of ShiftJIS
} CharEncodingType;
```

The minimum coding required for a character is the encoding that requires the fewest bytes. Also, the Palm OS supports only a single character encoding at a time, so **TxtCharEncoding** and **TxtStrEncoding** will always return a value equal to or less than the current encoding used by the system, or charEncodingUnknown if the character is completely unrecognizable to the current system.

These functions are not a reliable way to determine the current encoding that the system is using. For that purpose, use FtrGet to retrieve the current encoding:

```
UInt16 encoding;

Err error = FtrGet(sysFtrCreator, sysFtrNumEncoding,
                   &encoding);
```

The error value should be 0, and encoding will contain the CharEncoding Type value for the system's current character encoding.

Compiling with the PalmOSGlue library

Even though the text and international managers are not built into the Palm OS until version 3.1, you can add these useful managers to applications that support earlier versions of the Palm OS, as early as version 2.0. The way to add text and international manager support is to link the PalmOSGlue library into your application.

You need to link the PalmOSGlue.lib file into your application in CodeWarrior, or the libPalmOSGlue.a file if you are building the application with the PRC Tools. The PalmOSGlue library contains all the functions in the text manager, but they

are named differently. Any function beginning with **Txt** in the text manager starts with **TxtGlue** in the PalmOSGlue library. For example, an application using the PalmOSGlue library should call **TxtGlueCompare**, not **TxtCompare**.

PalmOSGlue is a link library, not a shared library, so it will increase the size of your compiled `.prc` file. The amount of the increase varies depending on how many glue functions your application actually calls.

When an application compiled with PalmOSGlue calls one of the glue functions, the Palm OS first tries to use the text and international managers contained in ROM. If those managers do not exist on the system, the system executes a simple ASCII equivalent of the requested function.

Using the File Streaming API

Although the 64KB limit on database size in the Palm OS is not a problem for most applications, it can pose a bit of a challenge for applications that must deal with large amounts of data, such as image viewers or long document readers. For applications that need to handle arbitrarily long data, the Palm OS provides the file streaming API, which is available in version 3.0 and later of the operating system.

A *file stream* is a block of data with no upper limit on its size, other than the available memory on the handheld. File streams are similar to files on a desktop computer, and they provide permanent storage for data, because the mechanism underlying the file streaming API uses standard Palm OS databases for storage. However, the HotSync Manager cannot transfer file streams to the desktop computer during a HotSync operation; you must first convert the file stream data to regular records.

The performance of the file streaming API is considerably slower than the performance of the data manager. If your application makes extensive use of individual records within its database, the file streaming API may be too slow to provide acceptable performance. File streaming does not work well for data that contains many records because of the overhead of parsing the file stream for those records.

The file streaming functions are based on the `stdio` functions from the C standard library, so if you are used to using `stdio` for handling files, most of the file streaming API will operate exactly as you expect it to. Although there are differences between the file streaming functions and those in the `stdio` library, many of the functions in the file streaming API have direct analogs in `stdio`. Table 20-3 shows the connections between file streaming functions and their counterparts in `stdio`.

Table 20-3
File Streaming API and stdio Correspondences

File Streaming Function	*stdio Analog*
FileClearError	clearerr
FileClose	fclose
FileEOF	feof
FileError	ferror
FileOpen	fopen
FileRead	fread
FileRewind	rewind
FileSeek	fseek
FileTell	ftell
FileWrite	fwrite

Opening File Streams

To open an existing file stream or create a brand new one, call **FileOpen**. The **FileOpen** function has the following prototype:

```
FileHand FileOpen (UInt16 cardNo, Char* nameP, UInt32 type,
    UInt32 creator, UInt32 openMode, Err* errP)
```

On a successful call, **FileOpen** returns a handle to the newly opened or created file stream. This handle should be used with other file streaming functions to perform other operations on the file stream. You can pass a pointer to a variable to receive any errors generated by **FileOpen** in the `errP` parameter. If you are not interested in receiving error values, pass `NULL` for the `errP` parameter. You can also retrieve errors produced by **FileOpen** using the **FileError** function, which is described later in this chapter in the section "Retrieving File Stream Errors."

The `cardNo` parameter is the number of the card containing the file stream; use `0` for this parameter, because currently no Palm OS handheld actually supports more than one memory card. Point the `nameP` parameter at a string that holds the name of the file stream. This file name follows the same rules as regular Palm OS database names; it may be a maximum of 31 characters in length, and it should be unique among all the database names on the handheld.

You can use the `type` and `creator` parameters to specify a database type and creator ID for the file stream. Unlike a regular database, type and creator ID are not required for file streams; you can use them to restrict **FileOpen** to opening only existing file streams that were created with a specific type and creator ID. If you

pass 0 for the type and creator parameters, **FileOpen** treats them as wildcards when opening an existing file stream.

When you're creating a new permanent file stream, a 0 value for type will result in a file stream with the constant value sysFileTFileStream (defined as strm in the Palm OS header file SystemResources.h) for its database type. If you specify fileModeTemporary for openMode and 0 for type when creating a new file stream, **FileOpen** creates the new file stream with a database type of sysFileTTemp instead, which is defined as temp in SystemResources.h.

A 0 value for creator when **FileOpen** is creating a new file stream causes the function to use the current application's creator ID as the new file stream's creator ID.

The openMode parameter is where you tell **FileOpen** what mode it should use when opening a file stream. Possible values for openMode may be a one primary mode constant, combined using a bitwise OR operator (|) with one or more secondary mode constants. Table 20-4 outlines the primary mode constants, and Table 20-5 shows the secondary mode constants and what they mean.

Table 20-4
Primary Open Mode Constants

Constant	*Description*
fileModeReadOnly	Opens a file stream for read-only access
fileModeReadWrite	Opens or creates a file stream for read/write access, first deleting any existing file stream that has the same name as the new file stream
fileModeUpdate	Opens or creates a file stream for read/write access, preserving any existing version of the file stream
fileModeAppend	Opens or creates a file stream for read/write access, writing new data to the end of the file stream

Table 20-5
Secondary Open Mode Constants

Constant	*Description*
fileModeDontOverwrite	Prevents the fileModeReadWrite mode from throwing away any existing file stream with the same name as the new file stream. This constant may be used only with the fileModeReadWrite primary mode constant.
fileModeLeaveOpen	Leaves the file stream open when the application exits. Most applications should not use this option.

Continued

Table 20-5 *(continued)*

Constant	*Description*
`fileModeExclusive`	Prevents other applications from opening the stream until this application has closed it.
`fileModeAnyTypeCreator`	Ignores the `type` and `creator` parameters when opening or replacing an existing file stream.
`fileModeTemporary`	Automatically deletes the file stream when it is closed.

As an example, the following code creates a new file stream and opens it for reading and writing, discarding any existing file stream that has the same name as the new file stream:

```
FileHand  newStream;

newStream = FileOpen(0, "MyNewFileStream", 0, 0,
                     fileModeReadWrite, NULL);
```

This call to **FileOpen** opens a file stream for updating, which prevents the removal of an existing file stream with the same name as the new file stream. The file is also opened in exclusive mode to prevent other applications from modifying the file stream until the current application closes the stream:

```
newStream = FileOpen(0, "MyUpdatedFileStream", 0, 0,
                     fileModeUpdate | fileModeExclusive, NULL);
```

Finally, this example creates a temporary file stream, suitable for use in situations where you need a place to cache large amounts of data that will not fit within the system's dynamic memory space:

```
newStream = FileOpen(0, "MyTempFileStream", 0, 0,
                     fileModeReadWrite | fileModeTemporary,
                     NULL);
```

Closing File Streams

When you have finished using a file stream, close it with **FileClose**. The **FileClose** function has the following prototype:

```
Err FileClose (FileHand stream)
```

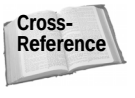
The `stream` parameter is the same handle returned from **FileOpen** when it opens the stream. Not only does **FileClose** close a file stream, but it also destroys the stream's handle for you. If you specified the `fileModeTemporary` secondary mode constant when opening the file stream, **FileClose** deletes the file stream as well. The **FileClose** function returns `0` if it successfully closes the file stream, or one of the error codes described in the next section if there is an error.

Retrieving File Stream Errors

The **FileError** function allows you to retrieve read and write errors for a particular file stream. Whenever an error occurs, an error indicator remains set on the file stream until it is closed by **FileClose**. You can use **FileError**, which has the following prototype, to test for read and write errors:

```
Err FileError (FileHand stream)
```

The **FileError** function returns `0` if there is currently no error set on the file stream, or it returns an error code describing the error. Several file streaming error codes specify everything from a lack of memory to a generic I/O error.



For a complete list of file streaming error codes, see Appendix A, "Palm OS API Quick Reference."

Instead of calling **FileError**, you can also use **FileGetLastError** to retrieve an error code:

```
Err FileGetLastError (FileHand stream)
```

The major difference between **FileError** and **FileGetLastError** is that the latter function clears the error code value, unless the error is an end of file or I/O error.

You can also explicitly clear the error value with the **FileClearerr** function:

```
Err FileClearerr (FileHand stream)
```

The **FileClearerr** function clears any file stream error, including those that **FileGetLastError** cannot clear.

Deleting File Streams

To delete a file stream, call the **FileDelete** function:

```
Err FileDelete (UInt16 cardNo, Char* nameP)
```

You may use **FileDelete** to remove a closed file stream only.

If you want to just truncate a file stream at a certain length, use **FileTruncate**:

```
Err FileTruncate (FileHand stream, Int32 newSize)
```

The `newSize` parameter specifies how large the truncated file stream should be. Be sure to keep `newSize` smaller than the current total size of the file stream.

Setting Position in a File Stream

Every file stream has a current position, which is an offset into the stream that is used for reading and writing data. If you need to change the current position, use the **FileSeek** function, whose prototype looks like this:

```
Err FileSeek (FileHand stream, Int32 offset,
    FileOriginEnum origin)
```

The `offset` is the number of bytes to seek, relative to whatever `origin` you specify. The `origin` may be one of the following constants:

- `fileOriginBeginning`: Beginning of the file stream
- `fileOriginCurrent`: Current position in the file stream
- `fileOriginEnd`: End of the file stream; this position is one byte past the last byte in the file stream

If you need to find out where the current position is within a file stream, call the **FileTell** function:

```
Int32 FileTell (FileHand stream, Int32* fileSizeP, Err* errP)
```

The return value from **FileTell** is the current position, expressed as the offset in bytes from the start of the file stream. If you provide a pointer for the `fileSizeP` parameter, you can also retrieve the current total size of the file stream in bytes. Pass `NULL` for `fileSizeP` to ignore the file stream size information.

You can also reset the position in a file stream back to the beginning of the stream with the **FileRewind** function:

```
Err FileRewind (FileHand stream)
```

The **FileRewind** function also has the side effect of clearing all error codes from the file stream, allowing you to start with a clean slate.

Reading and Writing File Stream Data

Reading data from a file stream may be accomplished by calling the **FileRead** function, which reads data into a buffer:

```
Int32 FileRead (FileHand stream, void* bufP, Int32 objSize,
    Int32 numObj, Err* errP)
```

The `bufP` parameter points to the beginning of a buffer you have allocated to receive data from the file stream. The **FileRead** function reads data in discrete objects, each of which is `objSize` bytes long. You specify the total number of objects to read in the `numObj` parameter. When **FileRead** returns, it places the total number of whole objects read from the file stream in its return value.

You may use FileRead to read data into a buffer only. If you want to read data from a file stream directly into a memory chunk or record, use FileDmRead, described later in this section.

As **FileRead** reads in data, it moves the current position in the file by the number of bytes actually read. If insufficient data is left in the file stream to meet the amount you specified using the `objSize` and `numObj` parameters, **FileRead** will result in a `fileErrEOF` error code, indicating that the end of the file has been reached. You can retrieve this error value from the `errP` parameter, or by calling **FileError**, **FileGetLastError**, or **FileEOF**.

The **FileEOF** function specifically checks the error status for the `fileErrEOF` error, indicating that the current position in the file stream is at the end of the file. The prototype for **FileEOF** looks like this:

```
Err FileEOF (FileHand stream)
```

If the current position is at the end of the file, **FileEOF** returns a non-zero value; otherwise, **FileEOF** returns `0`.

Typically, **FileEOF** is used with **FileRead** as part of a loop, allowing data retrieval until the end of the file stream is reached. As an example, the following code loops through a file stream, reading data from the stream ten characters at a time until **FileRead** hits the end of the file:

```
while(! FileEOF(stream)) {
    count = FileRead(stream, buffer, sizeof(Char), 10, NULL);
    buffer += count;

    // Total up actual bytes read.
    total += count;
}
```

If you want to read data from a file stream directly into a memory chunk or record in a database, use the **FileDmRead** function:

```
Int32 FileDmRead (FileHand stream, void* startOfDmChunkP,
    Int32 destOffset, Int32 objSize, Int32 numObj, Err* errP)
```

The `startOfDmChunkP` parameter points to the start of the memory chunk you want to write data to, and `destOffset` indicates the offset in bytes within that chunk at which **FileDmRead** should begin writing data. Otherwise, **FileDmRead** operates exactly like **FileRead**.

Writing to a file stream requires use of the **FileWrite** function:

```
Int32 FileWrite (FileHand stream, void* dataP, Int32 objSize,
    Int32 numObj, Err* errP)
```

Similar to **FileRead**, the `objSize` and `numObj` parameters tell **FileWrite** how large each write it makes should be and how many writes to attempt. The `dataP` parameter is a pointer to a buffer containing the data to write to the file stream. The **FileWrite** function returns the number of whole objects written to the file stream.

If not enough storage space is available to contain all of the data **FileWrite** has been requested to write, the function writes as much data as possible before quitting, which means that the last object written to the stream might be cut off in the middle.

Summary

In this chapter, you got to see some of the less often-used parts of the Palm OS and how to program them. After reading this chapter, you should know the following:

- Applications larger than 32KB may require some programming or linking tricks to avoid the 16-bit jump limit built into the processor.
- Because no single resource in the Palm OS may be larger than 64KB, you must segment really large applications in order to compile and run them properly.
- You may create and add custom fonts to applications, but you need to use the PRC Tools if you want to use custom fonts in a program's forms at build time.
- It is possible to create forms and most user interface elements at run time, though for most applications, it is completely unnecessary, not to mention more difficult to code than using resources created at build time.
- The text and international managers provide a wealth of functions and macros to help you deal with localizing an application to use different methods of character encoding, particularly multi-byte character sets.
- If you need to store data in larger amounts than the usual 64KB database limit, you can use the file streaming API, which provides functions similar to those in the `stdio` C standard library for reading and writing data to a file stream.

✦ ✦ ✦

Palm OS API Quick Reference

This appendix serves as a quick reference for the functions, data structures, and constants that make up the Palm OS application programming interface (API). Within Appendix A, you will find the following sections:

- **Functional Guide**. A listing of functions, data structures, and constants used in Palm OS programming
- **Events**. A guide to the events that the Palm OS uses within an application
- **Launch Codes**. The launch codes that an application may receive from the system, and what each code means

This appendix is only a quick guide to the parts of the Palm OS API that are mentioned in this book; there are other less frequently used parts of the API. For a complete reference to everything in the Palm OS, see the *Palm OS SDK Reference*, which is available from Palm, Inc., both by itself and as part of the Palm OS SDK.

Functional Guide

Palm OS functions are divided into logical groups called *managers*. Each manager is designed to provide a particular service, such as alarm handling or string manipulation. Almost all the functions in a particular manager begin with the same prefix; for example, Memory Manager functions begin with **Mem**, and Alarm Manager functions begin with **Alm**.

There are also many functions in the Palm OS that do not belong to a specific manager, but do fall within a common group, such as functions devoted to handling forms or other user interface objects. Like the manager functions, functions within a group tend to share the same prefix.

Table A-1 lists the managers and function groups described in this book with their prefixes, and the table also briefly describes the purpose of each manager or function group.

Table A-1
Palm OS Managers and Function Groups

Manager or Group	*Prefix*	*Description*
Alarm Manager	Alm	Sets or retrieves information about an application's alarms
Categories	Category	Implements user-defined categories in an application
Clipboard	Clipboard	Handles moving data to and from the system clipboard
Controls	Ctl	Provides routines for interacting with form controls, including buttons, push buttons, check boxes, pop-up triggers, selector triggers, repeating buttons, sliders, and feedback sliders
Data and Resource Manager	Dm	Handles storage and retrieval of data and resources
Date and Time Selector	Select	Displays date and time selector dialogs
Error Manager	Err	Provides facilities for adding debugging code to your application
Exchange Manager	Exg	Handles high-level IR beaming
Feature Manager	Ftr	Allows publishing and retrieval of special data that persists even after an application quits
Fields	Fld	Handles text field objects
File Streaming	File	Provides support for reading and writing arbitrarily long blocks of data
Find	Find	Implements the global find feature
Fonts	Fnt	Provides utility functions for handling fonts
Forms	Frm	Provides routines for interacting with form objects
Lists	Lst	Handles list objects
Memory Manager	Mem	Handles allocation and manipulation of memory
Menus	Menu	Deals with menus and menu bars

Manager or Group	*Prefix*	*Description*
Miscellaneous	none	Contains functions that do not fall into any other group
New Serial Manager	Srm	Controls serial I/O. This manager adds functions to the original Serial Manager and will eventually take its place.
Password	Pwd	Manipulates the system password
Preferences	Pref	Sets and retrieves system and application preference data
Private Records	Sec	Displays dialogs to allow the user to change whether private records are shown, hidden, or masked
Rectangles	Rct	Provides functions for manipulating rectangle structures
Scroll Bars	Scl	Manages scroll bars
Serial Manager	Ser	Controls serial I/O. This manager will be phased out in favor of the New Serial Manager.
Sound Manager	Snd	Makes sounds through the system speaker and plays standard MIDI files
String Manager	Str	Handles manipulation of strings
System Dialogs	Sys	Displays various system dialogs
System Event Manager	Evt	Directly manipulates the event queue
System Manager	Sys	Allows direct access to many low-level system functions
Tables	Tbl	Provides routines for managing tables and their contents
Text Manager	Txt	Manipulates text in a localization-friendly manner
Time Manager	Date, Tim	Deals with the system clock and converting between different units of time
UI Color List	UIColor	Allows applications to set and retrieve user interface colors
UI Controls	UI	Displays various dialogs for changing user interface settings, such as brightness, contrast, or color
Windows	Win	Handles display and manipulation of windows, as well as drawing in those windows

The rest of this section follows the structure in the table. Each manager or function group has its own major section, arranged alphabetically. Within each major section, there are one or two subsections, "Functions" and sometimes "Structures." The "Functions" subsection lists the functions that make up the manager or group. Each function listing includes a short description, the function's prototype, and any compatibility information, such as the minimum version of the Palm OS that supports the function.

The "Structures" subsection lists a group's associated data structures and enumerated types.

Alarm Manager Functions

The Alarm Manager functions set or retrieve information about an application's alarms.

- **AlmGetAlarm**: Returns the date and time of an application's current alarm, if one has been set.

```
UInt32 AlmGetAlarm (UInt16 cardNo, LocalID dbID, UInt32
*refP)
```

- **AlmSetAlarm**: Sets or cancels an application's current alarm.

```
Err AlmSetAlarm (UInt16 cardNo, LocalID dbID, UInt32 ref,
    UInt32 alarmSeconds, Boolean quiet)
```

Category Functions

The Category functions implement user-defined categories in an application.

- **CategoryEdit**: Displays a dialog that allows the user to edit categories; called by **CategorySelect** when the user chooses the "Edit Categories" item from the category list.

```
Boolean CategoryEdit (DmOpenRef db, UInt16 *category,
    UInt32 titleStrID, UInt8 numUneditableCategories)
```

  Available only on Palm OS 3.0 or later.

- **CategoryEditV20**: Allows the user to edit categories; called by **CategorySelect** when the user chooses the "Edit Categories" item from the category list.

```
Boolean CategoryEditV20 (DmOpenRef db, UInt16 *category,
    UInt32 titleStrID)
```

  For backward compatibility with Palm OS 2.0 only.

- **CategoryEditV10**: Allows the user to edit categories; called by **CategorySelect** when the user chooses the "Edit Categories" item from the category list.

```
Boolean CategoryEditV10 (DmOpenRef db, UInt16 *category)
```

  For backward compatibility with Palm OS 1.0 only.

- **CategoryGetName**: Returns the name of a category, given its index.

```
void CategoryGetName (DmOpenRef db, UInt16 index, Char *name)
```

- **CategoryGetNext**: Returns the index of the next category, given a category index.

```
UInt16 CategoryGetNext (DmOpenRef db, UInt16 index)
```

- **CategoryInitialize**: Initializes the application info block with category names stored in an app info string.

```
void CategoryInitialize (AppInfoPtr appInfoP,
    UInt16 localizedAppInfoStrID)
```

  Available only on Palm OS 2.0 or later.

- **CategorySelect**: Displays a pop-up list to allow the user to select a category.

```
Boolean CategorySelect (DmOpenRef db, const FormType *frm,
    UInt16 ctlID, UInt16 lstID,  Boolean title,
    UInt16 *categoryP, Char *categoryName,
    UInt8 numUneditableCategories, UInt32 editingStrID)
```

  Available only on Palm OS 2.0 or later.

- **CategorySelectV10**: Displays a pop-up list to allow the user to select a category.

```
Boolean CategorySelectV10 (DmOpenRef db, const FormType *frm,
    UInt16 ctlID, UInt16 lstID, Boolean title,
    UInt16 *categoryP, Char *categoryName)
```

  For backward compatibility with Palm OS 1.0 only.

- **CategorySetName**: Sets a category's name or deletes a category programmatically.

```
void CategorySetName (DmOpenRef db, UInt16 index,
    const Char *nameP)
```

  Available only on Palm OS 2.0 or later.

Category Structure

The category functions use one structure to keep track of an application's category information.

- `AppInfoType`: Data structure that should be present at the head of an application info block if an application uses the various **Category** functions to manage and display categories.

```
typedef struct {
    UInt16  renamedCategories;
    Char    categoryLabels [dmRecNumCategories]
                           [dmCategoryLength];
    UInt8   categoryUniqIDs[dmRecNumCategories];
    UInt8   lastUniqID;
```

```
    UInt8   padding;
} AppInfoType;

typedef AppInfoType *AppInfoPtr;
```

Clipboard Functions

The clipboard functions allow an application to communicate with the system clipboard.

- **ClipboardAddItem**: Adds an item to the clipboard.

  ```
  void ClipboardAddItem (const ClipboardFormatType format,
      const void *ptr, UInt16 length)
  ```

- **ClipboardAppendItem**: Appends data to the item currently on the clipboard.

  ```
  Err ClipboardAppendItem (const ClipboardFormatType format,
      const void *ptr, UInt16 length)
  ```

  Available only on Palm OS 3.2 or later.

- **ClipboardGetItem**: Returns a handle to the contents of the clipboard.

  ```
  MemHandle ClipboardGetItem (const ClipboardFormatType format,
      UInt16 *length)
  ```

Clipboard Structure

The clipboard functions use an enumerated type to identify data that may be passed to or from the clipboard.

- `ClipboardFormatType`: Specifies the type of data to add to or retrieve from the clipboard.

  ```
  enum clipboardFormats {
      clipboardText,
      clipboardInk,
      clipboardBitmap
  };

  typedef enum clipboardFormats ClipboardFormatType;
  ```

Control Functions

The Control functions provide routines for interacting with form controls, including buttons, push buttons, check boxes, pop-up triggers, selector triggers, repeating buttons, sliders, and feedback sliders.

- **CtlDrawControl**: Draws a control object on the screen.

```
void CtlDrawControl (ControlType *controlP)
```

- **CtlEnabled**: Returns `true` if the control is enabled.

```
Boolean CtlEnabled (const ControlType *controlP)
```

- **CtlGetLabel**: Returns a pointer to a control's text label.

```
const Char *CtlGetLabel (const ControlType *controlP)
```

- **CtlGetSliderValues**: Returns the current values associated with a slider or feedback slider control.

```
void CtlGetSliderValues(const ControlType *ctlP,
    UInt16 *minValueP, UInt16 *maxValueP, UInt16 *pageSizeP,
    UInt16 *valueP)
```

  Available only on Palm OS 3.5 or later.

- **CtlGetValue**: Retrieves the value of a control.

```
Int16 CtlGetValue (const ControlType *controlP)
```

- **CtlHandleEvent**: Handles an event in a control object; normally, the system calls this function for you to provide default handling of control events.

```
Boolean CtlHandleEvent (ControlType *controlP,
    EventType *pEvent)
```

- **CtlHitControl**: Simulates tapping a control by adding a `ctlSelectEvent` to the event queue.

```
void CtlHitControl (const ControlType *controlP)
```

- **CtlNewControl**: Dynamically creates a new control object.

```
ControlType *CtlNewControl (void **formPP, UInt16 ID,
    ControlStyleType style, const Char *textP, Coord x,
    Coord y, Coord width, Coord height, FontID font,
    UInt8 group, Boolean leftAnchor)
```

  Available only on Palm OS 3.0 or later.

- **CtlNewGraphicControl**: Dynamically creates a new graphic control object.

```
GraphicControlType *CtlNewGraphicControl (void **formPP,
    UInt16 ID, ControlStyleType style, DmResID bitmapID,
    DmResID selectedBitmapID, Coord x, Coord y, Coord width,
    Coord height, UInt8 group, Boolean leftAnchor)
```

  Available only on Palm OS 3.5 or later.

- **CtlNewSliderControl**: Dynamically creates a new slider or feedback slider control object.

```
SliderControlType *CtlNewSliderControl (void **formPP,
    UInt16 ID, ControlStyleType style, DmResID thumbID,
```

```
    DmResID backgroundID, Coord x, Coord y, Coord width,
    Coord height, UInt16 minValue, UInt16 maxValue,
    UInt16 pageSize, UInt16 value)
```

Available only on Palm OS 3.5 or later.

- **CtlSetEnabled**: Enables or disables a control.

```
void CtlSetEnabled (ControlType *controlP, Boolean usable)
```

- **CtlSetGraphics**: Sets the bitmaps for a graphic control and redraws the control if it is visible.

```
void CtlSetGraphics (ControlType *ctlP, DmResID newBitmapID,
    DmResID newSelectedBitmapID)
```

Available only on Palm OS 3.5 or later.

- **CtlSetLabel**: Sets the text label for a control and redraws the control if it is visible.

```
void CtlSetLabel (ControlType *controlP, const Char
*newLabel)
```

- **CtlSetSliderValues**: Sets the values associated with a slider or feedback slider control.

```
void CtlSetSliderValues(ControlType *ctlP,
    const UInt16 *minValueP, const UInt16 *maxValueP,
    const UInt16 *pageSizeP, const UInt16 *valueP)
```

Available only on Palm OS 3.5 or later.

- **CtlSetUsable**: Sets a control to usable or not usable.

```
void CtlSetUsable (ControlType *controlP, Boolean usable)
```

- **CtlSetValue**: Sets the value of a control and redraws the control if it is visible.

```
void CtlSetValue (ControlType *controlP, Int16 newValue)
```

- **CtlValidatePointer**: Returns `true` if passed a valid control pointer.

```
Boolean CtlValidatePointer (const ControlType *controlP)
```

For debugging only; do not use in released applications.

Control Structures

The control functions use a number of structures to keep track of all the data associated with controls.

- `ButtonFrameType`: Specifies the border style for a button; used in the `frame` field of the `ControlAttrType` structure.

```
enum buttonFrames {
    noButtonFrame,
    standardButtonFrame,
    boldButtonFrame,
```

```
    rectangleButtonFrame
};

typedef enum buttonFrames ButtonFrameType;
```

✦ `ControlAttrType`: Bit field defining various attributes of a control.

```
typedef struct {
    UInt8 usable          :1;
    UInt8 enabled         :1;
    UInt8 visible         :1;
    UInt8 on              :1;
    UInt8 leftAnchor      :1;
    UInt8 frame           :3;
    UInt8 drawnAsSelected :1;
    UInt8 graphical       :1;
    UInt8 vertical        :1;
    UInt8 reserved        :5;
} ControlAttrType;
```

The `drawnAsSelected`, `graphical`, and `vertical` fields are present only in Palm OS 3.5 or later.

✦ `ControlStyleType`: Specifies the type of a particular control (button, push button, check box, and so on); used in the `style` field of the `ControlType` structure.

```
enum controlStyles {
    buttonCtl,
    pushButtonCtl,
    checkboxCtl,
    popupTriggerCtl,
    selectorTriggerCtl,
    repeatingButtonCtl,
    sliderCtl,
    feedbackSliderCtl
};

typedef enum controlStyles ControlStyleType;
```

The `sliderCtl` and `feebackSliderCtl` values do not exist prior to Palm OS 3.5.

✦ `ControlType`: Defines a control object.

```
typedef struct ControlType {
    UInt16  id;
    RectangleType  bounds;
    Char    *text;
    ControlAttrType  attr;
    ControlStyleType  style;
    FontID  font;
    UInt8   group;
    UInt8   reserved;
} ControlType;
```

- `GraphicControlType`: Defines a graphic control object. You may cast a pointer to a `GraphicControlType` structure as a pointer to a `ControlType` structure for passing to any **Ctl** function.

```
typedef struct GraphicControlType {
    UInt16   id;
    RectangleType  bounds;
    DmResID  bitmapID;
    DmResID  selectedBitmapID;
    ControlAttrType  attr;
    ControlStyleType  style;
    FontID   unused;
    UInt8    group;
    UInt8    reserved;
} GraphicControlType;
```

  Available only on Palm OS 3.5 or later.

- `SliderControlType`: Defines a slider or feedback slider object. You may cast a pointer to a `SliderControlType` as a pointer to a `ControlType` structure for passing to any **Ctl** function.

```
typedef struct SliderControlType {
    UInt16   id;
    RectangleType  bounds;
    DmResID  thumbID;
    DmResID  backgroundID;
    ControlAttrType  attr;
    ControlStyleType  style;
    UInt8    reserved;
    Int16    minValue;
    Int16    maxValue;
    Int16    pageSize;
    Int16    value;
    MemPtr   activeSliderP;
} SliderControlType;
```

  Available only on Palm OS 3.5 or later.

Data and Resource Manager Functions

The Data and Resource Manager functions handle storage and retrieval of data and resources.

- **DmArchiveRecord**: Archives a record by setting its delete bit but leaving its data intact.

```
Err DmArchiveRecord (DmOpenRef dbP, UInt16 index)
```

- **DmAttachRecord**: Attaches a chunk of memory to a database as a new record.

```
Err DmAttachRecord (DmOpenRef dbP, UInt16 *atP, MemHandle newH,
    MemHandle *oldHP)
```

- **DmAttachResource**: Attaches a chunk of memory to a database as a new resource.

```
Err DmAttachResource (DmOpenRef dbP, MemHandle newH,
    DmResType resType, DmResID resID)
```

- **DmCloseDatabase**: Closes an open database.

```
Err DmCloseDatabase (DmOpenRef dbP)
```

- **DmComparF**: Application-defined callback function that compares two records in a database.

```
typedef Int16 DmComparF (void *rec1, void *rec1, Int16 other,
    SortRecordInfoPtr rec1SortInfo,
    SortRecordInfoPtr rec2SortInfo, MemHandle appInfoH);
```

- **DmCreateDatabase**: Creates a new database.

```
Err DmCreateDatabase (UInt16 cardNo, const Char *nameP,
    UInt32 creator, UInt32 type, Boolean resDB)
```

- **DmDatabaseInfo**: Retrieves information about a database.

```
Err DmDatabaseInfo (UInt16 cardNo, LocalID dbID, Char *nameP,
    UInt16 *attributesP, UInt16 *versionP, UInt32 *crDateP,
    UInt32 *modDateP, UInt32 *bckUpDateP, UInt32 *modNumP,
    LocalID *appInfoIDP, LocalID *sortInfoIDP, UInt32 *typeP,
    UInt32 *creatorP)
```

- **DmDatabaseProtect**: Increments or decrements a database's protection count.

```
Err DmDatabaseProtect (UInt16 cardNo, LocalID dbID,
    Boolean protect)
```

- **DmDatabaseSize**: Retrieves the size of a database and the number of records it contains.

```
Err DmDatabaseSize (UInt16 cardNo, LocalID dbID,
    UInt32 *numRecordsP, UInt32 *totalBytesP,
    UInt32 *dataBytesP)
```

- **DmDeleteCategory**: Deletes all the records in a particular category.

```
Err DmDeleteCategory (DmOpenRef dbR, UInt16 categoryNum)
```

  Available only on Palm OS 2.0 or later.

- **DmDeleteDatabase**: Deletes a database and everything in it.

```
Err DmDeleteDatabase (UInt16 cardNo, LocalID dbID)
```

- **DmDeleteRecord**: Deletes a record by setting its delete bit and destroying its data chunk.

```
Err DmDeleteRecord (DmOpenRef dbP, UInt16 index)
```

- **DmDetachRecord**: Orphans a record from its database, leaving its data chunk intact so it may be reattached to another database.

```
Err DmDetachRecord (DmOpenRef dbP, UInt16 index,
    MemHandle *oldHP)
```

- **DmDetachResource**: Orphans a resource from its database, leaving its data chunk intact so it may be reattached to another database.

```
Err DmDetachResource (DmOpenRef dbP, UInt16 index,
    MemHandle *oldHP)
```

- **DmFindDatabase**: Returns the local ID of a database, given its memory card and name.

```
LocalID DmFindDatabase (UInt16 cardNo, const Char *nameP)
```

- **DmFindRecordByID**: Retrieves the index of a record, given its unique ID.

```
Err DmFindRecordByID (DmOpenRef dbP, UInt32 uniqueID,
    UInt16 *indexP)
```

- **DmFindResource**: Looks in an open database for a resource and returns its index, given the resource's type and ID, or a handle to the resource.

```
UInt16 DmFindResource (DmOpenRef dbP, DmResType resType,
    DmResID resID, MemHandle resH)
```

- **DmFindResourceType**: Looks in an open database for a resource and returns its index, given the resource's type and type index.

```
UInt16 DmFindResourceType (DmOpenRef dbP, DmResType resType,
    UInt16 typeIndex)
```

- **DmFindSortPosition**: Returns the index where a record should be sorted in a database, using a callback comparison function to compare records.

```
UInt16 DmFindSortPosition (DmOpenRef dbP, void *newRecord,
    SortRecordInfoPtr newRecordInfo, DmComparF *compar,
    Int16 other)
```

  Available only on Palm OS 2.0 or later.

- **DmFindSortPositionV10**: Returns the index where a record should be sorted in a database, using a callback comparison function to compare records.

```
UInt16 DmFindSortPositionV10 (DmOpenRef dbP, void *newRecord,
    DmComparF *compar, Int16 other)
```

  For backward compatibility with Palm OS 1.0 only.

- **DmGet1Resource**: Returns a handle to a resource from the most recently opened resource database, given the resource's type and ID.

```
MemHandle DmGet1Resource (DmResType type, DmResID resID)
```

- **DmGetDatabase**: Returns the local ID of a database, given its memory card and index.

```
LocalID DmGetDatabase (UInt16 cardNo, UInt16 index)
```

- **DmGetDatabaseLockState**: Retrieves the number of locked and busy records in a database.

```
void DmGetDatabaseLockState (DmOpenRef dbR, UInt8 *highest,
    UInt32 *count, UInt32 *busy)
```

  Available only on Palm OS 3.2 or later.

- **DmGetLastErr**: Returns the error code generated by the last unsuccessful Data Manager call.

```
Err DmGetLastErr (void)
```

- **DmGetNextDatabaseByTypeCreator**: Retrieves the local ID and memory card of the next database that matches a given type and creator ID combination; may be called successively to return all the databases on the device that have the specified type and creator ID.

```
Err DmGetNextDatabaseByTypeCreator (Boolean newSearch,
    DmSearchStatePtr stateInfoP, UInt32 type, UInt32 creator,
    Boolean onlyLatestVers, UInt16 *cardNoP, LocalID *dbIDP)
```

- **DmGetRecord**: Returns a handle to a record, given its index.

```
MemHandle DmGetRecord (DmOpenRef dbP, UInt16 index)
```

- **DmGetResource**: Returns a handle to a resource, searching through all open resource databases for a given type and ID.

```
MemHandle DmGetResource (DmResType type, DmResID resID)
```

- **DmGetResourceIndex**: Returns a handle to a resource, given its index.

```
MemHandle DmGetResourceIndex (DmOpenRef dbP, UInt16 index)
```

- **DmInsertionSort**: Sorts records in a database using an insertion sort algorithm.

```
Err DmInsertionSort (DmOpenRef dbR, DmComparF *compar,
    Int16 other)
```

- **DmMoveCategory**: Changes the category of all records in a given category to another category.

```
Err DmMoveCategory (DmOpenRef dbP, UInt16 toCategory,
    UInt16 fromCategory, Boolean dirty)
```

- **DmMoveRecord**: Moves a record from one index to another within a database.

```
Err DmMoveRecord (DmOpenRef dbP, UInt16 from, UInt16 to)
```

- **DmNewHandle**: Allocates a new chunk of memory from the same memory card where a given database's header is located.

```
MemHandle DmNewHandle (DmOpenRef dbP, UInt32 size)
```

- **DmNewRecord**: Allocates space for a new record in a database and returns its handle.

```
MemHandle DmNewRecord (DmOpenRef dbP, UInt16 *atP, UInt32 size)
```

- **DmNewResource**: Allocates space for a new resource in a database and returns its handle.

```
MemHandle DmNewResource (DmOpenRef dbP, DmResType resType,
    DmResID resID, UInt32 size)
```

- **DmNumDatabases**: Returns the number of databases on a given memory card.

```
UInt16 DmNumDatabases (UInt16 cardNo)
```

- **DmNumRecords**: Returns the number of records in a database.

```
UInt16 DmNumRecords (DmOpenRef dbP)
```

- **DmNumRecordsInCategory**: Returns the number of records in a database that are part of a given category.

```
UInt16 DmNumRecordsInCategory (DmOpenRef dbP, UInt16
category)
```

- **DmNumResources**: Returns the number of resources in a database.

```
UInt16 DmNumResources (DmOpenRef dbP)
```

- **DmOpenDatabase**: Opens a database and returns a reference to it.

```
DmOpenRef DmOpenDatabase (UInt16 cardNo, LocalID dbID,
    UInt16 mode)
```

- **DmOpenDatabaseByTypeCreator**: Opens the most recent version of a database that matches a given type and creator ID.

```
DmOpenRef DmOpenDatabaseByTypeCreator (UInt32 type,
    UInt32 creator, UInt16 mode)
```

- **DmOpenDatabaseInfo**: Retrieves information about an open database.

```
Err DmOpenDatabaseInfo (DmOpenRef dbP, LocalID *dbIDP,
    UInt16 *openCountP, UInt16 *modeP, UInt16 *cardNoP,
    Boolean *resDBP)
```

- **DmPositionInCategory**: Returns the position of a record within a given category.

```
UInt16 DmPositionInCategory (DmOpenRef dbP, UInt16 index,
    UInt16 category)
```

- **DmQueryNextInCategory**: Returns a handle to the next record in a given category for read-only access.

```
MemHandle DmQueryNextInCategory (DmOpenRef dbP, UInt16
*indexP,
    UInt16 category)
```

- **DmQueryRecord**: Returns a handle to a record for read-only access.

```
MemHandle DmQueryRecord (DmOpenRef dbP, UInt16 index)
```

- **DmQuickSort**: Sorts records in a database using a quicksort algorithm.

```
Err DmQuickSort (DmOpenRef dbP, DmComparF *compar, Int16 other)
```

- **DmRecordInfo**: Retrieves information about a record.

```
Err DmRecordInfo (DmOpenRef dbP, UInt16 index, UInt16 *attrP,
    UInt32 *uniqueIDP, LocalID *chunkIDP)
```

- **DmReleaseRecord**: Releases a record marked busy by **DmGetRecord** or **DmNewRecord**.

```
Err DmReleaseRecord (DmOpenRef dbP, UInt16 index,
    Boolean dirty)
```

- **DmReleaseResource**: Releases a resource acquired by **DmGetResource**.

```
Err DmReleaseResource (MemHandle resourceH)
```

- **DmRemoveRecord**: Removes a record from a database completely, disposing of its data chunk.

```
Err DmRemoveRecord (DmOpenRef dbP, UInt16 index)
```

- **DmRemoveResource**: Removes a resource from a database.

```
Err DmRemoveResource (DmOpenRef dbP, UInt16 index)
```

- **DmRemoveSecretRecords**: Removes from a database all records marked private.

```
Err DmRemoveSecretRecords (DmOpenRef dbP)
```

- **DmResizeRecord**: Resizes a record.

```
MemHandle DmResizeRecord (DmOpenRef dbP, UInt16 index,
    UInt32 newSize)
```

- **DmResizeResource**: Resizes a resource.

```
MemHandle DmResizeResource (MemHandle resourceH,
    UInt32 newSize)
```

- **DmResourceInfo**: Retrieves information about a resource.

```
Err DmResourceInfo (DmOpenRef dbP, UInt16 index,
    DmResType *resTypeP, DmResID *resIDP,
    LocalID *chunkLocalIDP)
```

- **DmSearchRecord**: Searches through all open record databases for a record with a given handle.

```
UInt16 DmSearchRecord (MemHandle recH, DmOpenRef *dbPP)
```

- **DmSearchResource**: Searches through all open resource databases for a resource of a given type and ID, or with a given handle.

```
UInt16 DmSearchResource (DmResType resType, DmResID resID,
    MemHandle resH, DmOpenRef *dbPP)
```

- **DmSeekRecordInCategory**: Retrieves the index of the record closest to a given offset from a given record's index, searching only through records of a given category.

```
Err DmSeekRecordInCategory (DmOpenRef dbP, UInt16 *indexP,
    Int16 offset, Int16 direction, UInt16 category)
```

- **DmSet**: Sets a specified number of bytes in a record to a given byte value.

```
Err DmSet (void *recordP, UInt32 offset, UInt32 bytes,
    UInt8 value)
```

- **DmSetDatabaseInfo**: Sets information about a database.

```
Err DmSetDatabaseInfo (UInt16 cardNo, LocalID dbID,
    const Char *nameP, UInt16 *attributesP, UInt16 *versionP,
    UInt32 *crDateP, UInt32 *modDateP,UInt32 *bckUpDateP,
    UInt32 *modNumP, LocalID *appInfoIDP, LocalID
*sortInfoIDP,
    UInt32 *typeP, UInt32 *creatorP)
```

- **DmSetRecordInfo**: Sets information about a record.

```
Err DmSetRecordInfo (DmOpenRef dbP, UInt16 index,
    UInt16 *attrP, UInt32 *uniqueIDP)
```

- **DmSetResourceInfo**: Sets information about a resource.

```
Err DmSetResourceInfo (DmOpenRef dbP, UInt16 index,
    DmResType *resTypeP, DmResID *resIDP)
```

- **DmStrCopy**: Copies a null-terminated string into a record at a given offset.

```
Err DmStrCopy (void *recordP, UInt32 offset,
const Char  *srcP)
```

- **DmWrite**: Writes a specified number of bytes of data into a record at a given offset.

```
Err DmWrite (void *recordP, UInt32 offset, const void  *srcP,
    UInt32 bytes)
```

- **DmWriteCheck**: Checks that the parameters of a **DmWrite** operation will be successful, without actually writing any data to the record.

```
Err DmWriteCheck (void *recordP, UInt32 offset, UInt32 bytes)
```

Data and Resource Manager Structures

The Data and Resource Manager functions use a number of structures to identify database references, resource types, and sorting information.

- `DmOpenRef`: A pointer to an open database, used by functions that need access to an open database.

```
typedef void *DmOpenRef
```

- `DmResID`: Identifier for a resource; resource ID numbers greater than `10000` are reserved for use by the system.

```
typedef UInt16 DmResID;
```

- `DmResType`: Four-character code specifying a resource type.

```
typedef UInt32 DmResType;
```

- `SortRecordInfoType`: Specifies information about a record that may be useful when an application tries to find a record's proper place in sort order.

```
typedef struct {
    UInt8  attributes;
    UInt8  uniqueID[3];
} SortRecordInfoType;

typedef SortRecordInfoType *SortRecordInfoPtr;
```

Date and Time Selector Functions

The Date and Time Selector functions display date and time selector dialogs.

- **SelectDay**: Displays a date selection dialog.

```
Boolean SelectDay (const SelectDayType selectDayBy,
    Int16 *month, Int16 *day, Int16 *year, const Char *title)
```

  Available only on Palm OS 2.0 or later.

- **SelectDayV10**: Displays a date selection dialog.

```
Boolean SelectDay (Int16 *month, Int16 *day, Int16 *year,
    const Char title)
```

  For backward compatibility with Palm OS 1.0 only.

- **SelectOneTime**: Displays a selection dialog for picking a single time value.

```
Boolean SelectOneTime (Int16 *hour, Int16 *minute,
    const Char *titleP)
```

  Available only on Palm OS 3.1 or later.

- **SelectTime**: Displays a selection dialog for picking start and end times.

```
Boolean SelectTime (TimeType *startTimeP, TimeType *endTimeP,
    Boolean untimed, const Char *titleP, Int16 startOfDay,
    Int16 endOfDay, Int16 startOfDisplay)
```

  Available only on Palm OS 3.5 or later.

- **SelectTimeV33**: Displays a selection dialog for picking start and end times.

```
Boolean SelectTimeV33 (TimeType *startTimeP,
    TimeType *EndTimeP, Boolean untimed, Char *title,
    Int16 startOfDay)
```

  For backward compatibility with Palm OS 3.3 only.

Date and Time Selector Structure

The date and time selector functions use an enumerated type to identify different ways in which the day selection dialog box may appear.

- `SelectDayType`: Indicates the type of day selection dialog box to display.

```
typedef enum  {
    selectDayByDay,
    selectDayByWeek,
    selectDayByMonth
} SelectDayType;
```

Error Manager Functions

The Error Manager functions provide facilities for adding debugging code to your application.

- **ErrAlert**: Macro that displays an alert dialog containing an error string from either the system or an application's `tSTL` resource.

```
ErrAlert (err)
```

  Available only on Palm OS 3.2 or later.

- **ErrDisplay**: Macro that displays an alert dialog containing a text string.

```
ErrDisplay (msg)
```

- **ErrDisplayFileLineMsg**: Displays an alert dialog containing a source code file name, line number, and text string; called by **ErrFatalDisplayIf** and **ErrNonFatalDisplayIf**.

```
void ErrDisplayFileLineMsg (const Char  *const filename,
    UInt16 lineno, const Char  *const msg)
```

- **ErrFatalDisplayIf**: Macro that displays an alert dialog if a given condition is `true` and error checking is set to either partial or full.

```
ErrFatalDisplayIf (condition, msg)
```

- **ErrNonFatalDisplayIf**: Macro that displays an alert dialog if a given condition is `true` and error checking is set to full.

```
ErrNonFatalDisplayIf (condition, msg)
```

Exchange Manager Functions

The Exchange Manager functions handle high-level IR beaming.

- **DeleteProc**: Application-defined callback function used by **ExgDBRead** when a database already exists that shares a name with an incoming database; **DeleteProc** can delete, rename, or move the existing database to resolve the conflict.

```
Boolean DeleteProc (const char *nameP, UInt16 version,
    UInt16 cardNo, LocalID dbID, void *userDataP)
```

- **ExgAccept**: Accepts an IR beaming connection from a remote device.

  ```
  Err ExgAccept (ExgSocketPtr socketP)
  ```

  Available only on Palm OS 3.0 or later.

- **ExgDBRead**: Reads a Palm OS database from its desktop `.prc` or `.pdb` format and writes it to storage using a callback function.

  ```
  Err ExgDBRead (ExgDBReadProcPtr readProcP,
      ExgDBDeleteProcPtr deleteProcP, void *userDataP,
      LocalID *dbIDP, UInt16 cardNo, Boolean *needResetP,
      Boolean keepDates)
  ```

  Available only on Palm OS 3.0 or later.

- **ExgDBWrite**: Reads a Palm OS database from its internal format on the handheld and writes it to the desktop `.prc` or `.pdb` format, using a callback function to perform special write operations, such as beaming the database.

  ```
  Err ExgDBWrite (ExgDBWriteProcPtr writeProcP, void
  *userDataP,
      const char *nameP, LocalID dbID, UInt16 cardNo)
  ```

  Available only on Palm OS 3.0 or later.

- **ExgDisconnect**: Disconnects an IR beaming connection with a remote device.

  ```
  Err ExgDisconnect(ExgSocketPtr socketP, Err error)
  ```

  Available only on Palm OS 3.0 or later.

- **ExgDoDialog**: Displays a dialog box to allow the user to accept or reject incoming beamed data.

  ```
  Boolean ExgDoDialog (ExgSocketPtr socketP,
      ExgDialogInfoType *infoP, Err *errP)
  ```

  Available only on Palm OS 3.5 or later.

- **ExgPut**: Initiates data transfer to a remote device.

  ```
  Err ExgPut (ExgSocketPtr socketP)
  ```

  Available only on Palm OS 3.0 or later.

- **ExgReceive**: Receives data from a remote device.

  ```
  UInt32 ExgReceive (ExgSocketPtr socketP, void *bufP,
      const UInt32 bufLen, Err *err)
  ```

  Available only on Palm OS 3.0 or later.

- **ExgRegisterData**: Registers an application to receive specific types of data.

  ```
  Err ExgRegisterData (const UInt32 creatorID, const UInt16 id,
      const Char  *const dataTypesP)
  ```

  Available only on Palm OS 3.0 or later.

- **ExgSend**: Sends data to a remote device.

```
UInt32 ExgSend (ExgSocketPtr socketP, const void  *const bufP,
    const UInt32 bufLen, Err *err)
```

  Available only on Palm OS 3.0 or later.

- **ReadProc**: Application-defined callback function that **ExgDBRead** calls to read in a database.

```
Err ReadProc (void *dataP, UInt32 *sizeP, void *userDataP)
```

- **WriteProc**: Application-defined callback function that **ExgDBWrite** calls to write out a database.

```
Err WriteProc (const void *dataP, UInt32 *sizeP,
    void *userDataP)
```

Exchange Manager Structures

The Exchange Manager uses a few structures to keep track of important values during infrared beaming operations.

- `ExgAskResultType`: Defines the possible values for the `result` field of a `sysAppLaunchCmdExgAskUser` launch code.

```
typedef enum {
    exgAskDialog,
    exgAskOk,
    exgAskCancel
} ExgAskResultType;
```

- `ExgGoToType`: Contains information about which record should be displayed once a transfer has been completed.

```
typedef struct {
    UInt16   dbCardNo;
    LocalID  dbID;
    UInt16   recordNum;
    UInt32   uniqueID;
    UInt32   matchCustom;
} ExgGoToType;

typedef ExgGoToType *ExgGoToPtr;
```

- `ExgSocketType`: Holds information about a connection and the type of data being transferred.

```
typedef struct ExgSocketType {
    UInt16  libraryRef;
    UInt32  socketRef;
    UInt32  target;
    UInt32  count;
    UInt32  length;
    UInt32  time;
```

```
    UInt32  appData;
    UInt32  goToCreator;
    ExgGoToType  goToParams;
    UInt16  localMode:1;
    UInt16  packetMode:1;
    UInt16  noGoTo:1;
    UInt16  noStatus:1;
    UInt16  reserved:12;
    Char    *description;
    Char    *type;
    Char    *name;
} ExgSocketType;

typedef ExgSocketType *ExgSocketPtr;
```

Feature Manager Functions

The Feature Manager functions allow publishing and retrieval of special data that persists even after an application quits.

- **FtrGet**: Retrieves a feature value.

  ```
  Err FtrGet (UInt32 creator, UInt16 featureNum, UInt32 *valueP)
  ```

- **FtrGetByIndex**: Retrieves a feature, given an index.

  ```
  Err FtrGetByIndex (UInt16 index, Boolean romTable,
      UInt32 *creatorP, UInt16 *numP, UInt32 *valueP)
  ```

- **FtrPtrFree**: Releases memory allocated with **FtrPtrNew**.

  ```
  Err FtrPtrFree (UInt32 creator, UInt16 featureNum)
  ```

  Available only on Palm OS 3.1 or later.

- **FtrPtrNew**: Allocates a chunk of feature memory.

  ```
  Err FtrPtrNew (UInt32 creator, UInt16 featureNum, UInt32 size,
      void **newPtrP)
  ```

  Available only on Palm OS 3.1 or later.

- **FtrPtrResize**: Resizes feature memory.

  ```
  Err FtrPtrResize (UInt32 creator, UInt16 featureNum,
      UInt32 newSize, void **newPtrP)
  ```

  Available only on Palm OS 3.1 or later.

- **FtrSet**: Sets a feature value.

  ```
  Err FtrSet (UInt32 creator, UInt16 featureNum, UInt32 newValue)
  ```

- **FtrUnregister**: Unregisters a feature.

  ```
  Err FtrUnregister (UInt32 creator, UInt16 featureNum)
  ```

Field Functions

The Field functions handle text field objects.

- **FldCalcFieldHeight**: Calculates how many lines tall a field needs to be to contain a given string.

```
UInt16 FldCalcFieldHeight (const Char *chars, UInt16
maxWidth)
```

- **FldCompactText**: Compacts the memory chunk containing a field's text to free up unused space.

```
void FldCompactText (FieldType *fldP)
```

- **FldCopy**: Copies the currently selected text to the clipboard.

```
void FldCopy (const FieldType *fldP)
```

- **FldCut**: Cuts the currently selected text to the clipboard.

```
void FldCut (FieldType *fldP)
```

- **FldDelete**: Deletes a range of characters from a text field.

```
void FldDelete (FieldType *fldP, UInt16 start, UInt16 end)
```

- **FldDirty**: Returns `true` if a field's text has been modified.

```
Boolean FldDirty (const FieldType *fldP)
```

- **FldDrawField**: Draws the text of a field.

```
void FldDrawField (FieldType *fldP)
```

- **FldEraseField**: Erases a field's text and turns off the insertion point.

```
void FldEraseField (FieldType *fldP)
```

- **FldFreeMemory**: Releases the memory that holds a field's text and the memory containing the field's word-wrapping information.

```
void FldFreeMemory (FieldType *fldP)
```

- **FldGetAttributes**: Returns a text field's attributes.

```
void FldGetAttributes (const FieldType *fldP,
    FieldAttrPtr attrP)
```

- **FldGetBounds**: Returns the rectangle occupied by a field.

```
void FldGetBounds (const FieldType *fldP, RectanglePtr rect)
```

- **FldGetFont**: Returns the font ID currently used by a text field.

```
FontID FldGetFont (const FieldType *fldP)
```

- **FldGetInsPtPosition**: Returns the position of a text field's insertion point.

```
UInt16 FldGetInsPtPosition (const FieldType *fldP)
```

- **FldGetMaxChars**: Returns the maximum number of bytes that a text field may contain.

```
UInt16 FldGetMaxChars (const FieldType *fldP)
```

- **FldGetNumberOfBlankLines**: Returns the number of blank lines displayed at the bottom of a text field.

```
UInt16 FldGetNumberOfBlankLines (const FieldType *fldP)
```

- **FldGetScrollPosition**: Returns the offset of the first character in the first visible line of a text field.

```
UInt16 FldGetScrollPosition (const FieldType *fldP)
```

- **FldGetScrollValues**: Retrieves values from a text field for updating a scroll bar.

```
void FldGetScrollValues (const FieldType *fldP,
    UInt16 *scrollPosP, UInt16 *textHeightP,
    UInt16 *fieldHeightP)
```

- **FldGetSelection**: Retrieves the start and end points of a text field's selected text.

```
void FldGetSelection (const FieldType *fldP,
    UInt16 *startPosition, UInt16 *endPosition)
```

- **FldGetTextHandle**: Returns a handle to the memory chunk that contains a field's text.

```
MemHandle FldGetTextHandle (const FieldType *fldP)
```

- **FldGetTextHeight**: Returns the height of a field's text in pixels, skipping empty lines at the end of the field.

```
UInt16 FldGetTextHeight (const FieldType *fldP)
```

- **FldGetTextLength**: Returns the length of a field's text in bytes.

```
UInt16 FldGetTextLength (const FieldType *fldP)
```

- **FldGetTextPtr**: Returns a locked pointer to a field's text.

```
Char *FldGetTextPtr (FieldType *fldP)
```

- **FldGetVisibleLines**: Returns the number of lines in a text field that are visible at the same time.

```
UInt16 FldGetVisibleLines (const FieldType *fldP)
```

- **FldGrabFocus**: Turns the insertion point on and displays a blinking cursor in the given field; usually, you should use **FrmSetFocus** instead.

```
void FldGrabFocus (FieldType *fldP)
```

- **FldHandleEvent**: Handles events for a text field object; normally, the system calls this function for you to provide default handling of field events.

```
Boolean FldHandleEvent (FieldType *fldP, EventType *eventP)
```

- **FldInsert**: Replaces the currently selected text in a field with a given string and redraws the field.

```
Boolean FldInsert (FieldType *fldP, const Char *insertChars,
    UInt16 insertLen)
```

- **FldMakeFullyVisible**: Expands a resizable field's height to display all its text.

```
Boolean FldMakeFullyVisible (FieldType *fldP)
```

- **FldNewField**: Dynamically creates a new text field object.

```
FieldType *FldNewField (void **formPP, UInt16 id, Coord x,
    Coord y, Coord width, Coord height, FontID font,
    UInt32 maxChars, Boolean editable, Boolean underlined,
    Boolean singleLine, Boolean dynamicSize,
    JustificationType justification, Boolean autoShift,
    Boolean hasScrollBar, Boolean numeric)
```

  Available only on Palm OS 3.0 or later.

- **FldPaste**: Writes the current contents of the clipboard into a text field, overwriting selected text, if any.

```
void FldPaste (FieldType *fldP)
```

- **FldRecalculateField**: Updates the word-wrapping information for a text field.

```
void FldRecalculateField (FieldType *fldP, Boolean redraw)
```

- **FldScrollable**: Returns `true` if a text field is scrollable in a given direction.

```
Boolean FldScrollable (const FieldType *fldP,
    WinDirectionType direction)
```

- **FldScrollField**: Scrolls a text field up or down a given number of lines.

```
void FldScrollField (FieldType *fldP, UInt16 linesToScroll,
    WinDirectionType direction)
```

- **FldSetAttributes**: Sets attributes for a text field.

```
void FldSetAttributes (FieldType *fldP,
    const FieldAttrPtr attrP)
```

- **FldSetDirty**: Sets or clears a text field's dirty bit.

```
void FldSetDirty (FieldType *fldP, Boolean dirty)
```

- **FldSetFont**: Sets the font for a text field, updates the field's word-wrapping information, and redraws the field if it is visible.

```
void FldSetFont (FieldType *fldP, FontID fontID)
```

- **FldSetInsertionPoint**: Sets the location of a text field's insertion point without scrolling the field to make the new position visible.

```
void FldSetInsertionPoint (FieldType *fldP, UInt16 pos)
```

- **FldSetInsPtPosition**: Sets the location of a text field's insertion point, scrolling the field to make the new position visible.

```
void FldSetInsPtPosition (FieldType *fldP, UInt16 pos)
```

- **FldSetMaxChars**: Sets the maximum number of bytes' worth of characters that a text field can hold.

```
void FldSetMaxChars (FieldType *fldP, UInt16 maxChars)
```

- **FldSetScrollPosition**: Scrolls a text field so the character at a given offset is visible, redrawing the field if necessary.

```
void FldSetScrollPosition (FieldType *fldP, UInt16 pos)
```

- **FldSetSelection**: Sets the start and end points of the selected text in a field.

```
void FldSetSelection (FieldType *fldP, UInt16 startPosition,
    UInt16 endPosition)
```

- **FldSetText**: Sets a field's text string without updating the display, allowing for in-place editing of text in a database.

```
void FldSetText (FieldType *fldP, MemHandle textHandle,
    UInt16 offset, UInt16 size)
```

- **FldSetTextHandle**: Sets a field's text string without updating the display, using an entire memory chunk for the source string.

```
void FldSetTextHandle (FieldType *fldP, MemHandle textHandle)
```

- **FldSetUsable**: Makes a text field usable or unusable.

```
void FldSetUsable (FieldType *fldP, Boolean usable)
```

- **FldUndo**: Undoes the last change made to a text field.

```
void FldUndo (FieldType *fldP)
```

- **FldWordWrap**: Returns the number of bytes of characters of a given string that may be displayed within a certain width in the current font.

```
UInt16 FldWordWrap (const Char *chars, Int16 maxWidth)
```

Field Structures

The field functions use a few structures to hold information about text field objects.

- `FieldAttrType`: Bit field that defines the attributes for a text field object; used in the `FieldType` structure.

```
typedef struct {
    UInt16 usable        :1;
    UInt16 visible       :1;
    UInt16 editable      :1;
    UInt16 singleLine    :1;
    UInt16 hasFocus      :1;
```

```
    UInt16 dynamicSize   :1;
    UInt16 insPtVisible  :1;
    UInt16 dirty         :1;
    UInt16 underlined    :2;
    UInt16 justification :2;
    UInt16 autoShift     :1;
    UInt16 hasScrollBar  :1;
    UInt16 numeric       :1;
} FieldAttrType;
```

- `FieldType`: Defines a text field object.

```
typedef struct FieldType {
    UInt16  id;
    RectangleType  rect;
    FieldAttrType  attr;
    Char    *text;
    MemHandle  textHandle;
    LineInfoPtr  lines;
    UInt16  textLen;
    UInt16  textBlockSize;
    UInt16  maxChars;
    UInt16  selFirstPos;
    UInt16  selLastPos;
    UInt16  insPtXPos;
    UInt16  insPtYPos;
    FontID  fontID;
    UInt8   reserved;
} FieldType;
```

- `LineInfoType`: Describes a single line in a form object's `lines` array.

```
typedef struct {
    UInt16 start;
    UInt16 length;
} LineInfoType;
```

File Streaming Functions

The File Streaming functions provide support for reading and writing arbitrarily long blocks of data.

- **FileClearerr**: Clears the file streaming error status.

  ```
  Err FileClearerr (FileHand stream)
  ```

  Available only on Palm OS 3.0 or later.

- **FileClose**: Closes a file stream and destroys its handle.

  ```
  Err FileClose (FileHand stream)
  ```

  Available only on Palm OS 3.0 or later.

- **FileDelete**: Deletes a closed file stream.

```
Err FileDelete (UInt16 cardNo, Char *nameP)
```

  Available only on Palm OS 3.0 or later.

- **FileDmRead**: Reads data from a file stream directly into a memory chunk, record, or resource.

```
Int32 FileDmRead (FileHand stream, void *startOfDmChunkP,
    Int32 destOffset, Int32 objSize, Int32 numObj, Err *errP)
```

  Available only on Palm OS 3.0 or later.

- **FileEOF**: Returns `0` if a file stream's read/write position is not at end of file.

```
Err FileEOF (FileHand stream)
```

  Available only on Palm OS 3.0 or later.

- **FileError**: Returns the current I/O error status for a file stream.

```
Err FileError (FileHand stream)
```

  Available only on Palm OS 3.0 or later.

- **FileGetLastError**: Returns the last file stream error code and clears the error status unless the error is end of file or an I/O error.

```
Err FileGetLastError (FileHand stream)
```

  Available only on Palm OS 3.0 or later.

- **FileOpen**: Opens or creates a file stream.

```
FileHand FileOpen (UInt16 cardNo, Char *nameP, UInt32 type,
    UInt32 creator, UInt32 openMode, Err *errP)
```

  Available only on Palm OS 3.0 or later.

- **FileRead**: Reads data from a file stream into a buffer.

```
Int32 FileRead (FileHand stream, void *bufP, Int32 objSize,
    Int32 numObj, Err *errP)
```

  Available only on Palm OS 3.0 or later.

- **FileRewind**: Sets a file stream's read/write position back to the start of the stream and clears all file streaming errors.

```
Err FileRewind (FileHand stream)
```

  Available only on Palm OS 3.0 or later.

- **FileSeek**: Sets the current read/write position within a file stream.

```
Err FileSeek (FileHand stream, Int32 offset,
    FileOriginEnum origin)
```

  Available only on Palm OS 3.0 or later.

- **FileTell**: Returns a file stream's current read/write position, and optionally retrieves the size of the stream in bytes.

  ```
  Int32 FileTell (FileHand stream, Int32 *fileSizeP, Err *errP)
  ```

  Available only on Palm OS 3.0 or later.

- **FileTruncate**: Truncates a file stream to a given number of bytes in size.

  ```
  Err FileTruncate (FileHand stream, Int32 newSize)
  ```

  Available only on Palm OS 3.0 or later.

- **FileWrite**: Writes data to a file stream.

  ```
  Int32 FileWrite (FileHand stream, void *dataP, Int32 objSize,
      Int32 numObj, Err *errP)
  ```

  Available only on Palm OS 3.0 or later.

Find Functions

The Find routines help implement the global find feature.

- **FindDrawHeader**: Draws the header line that separates the list of found records into different database groupings.

  ```
  Boolean FindDrawHeader (FindParamsPtr findParams,
      Char const *title)
  ```

- **FindGetLineBounds**: Retrieves the rectangle defining the next available screen region for drawing a match in the Find Results dialog.

  ```
  void FindGetLineBounds (const FindParamsType *findParams,
      RectanglePtr r)
  ```

- **FindSaveMatch**: Saves the record and text offset within the record of a matching text string, allowing later navigation back to that record.

  ```
  Boolean FindSaveMatch (FindParamsPtr findParams,
      UInt16 recordNum, UInt16 pos, UInt16 fieldNum,
      UInt32 appCustom, UInt16 cardNo, LocalID dbID)
  ```

- **FindStrInStr**: Performs a case-insensitive search to find one string within another string, matching only the beginnings of words.

  ```
  Boolean FindStrInStr (Char const *strToSearch,
      Char const *strToFind, UInt16 *posP)
  ```

Font Functions

The Font functions provide utilities for handling fonts.

- **FntAverageCharWidth**: Returns the average width in pixels of characters in the current font.

  ```
  Int16 FntAverageCharWidth (void)
  ```

- **FntBaseLine**: Returns the distance in pixels from the top of a character cell to its baseline for the current font.

```
Int16 FntBaseLine (void)
```

- **FntCharHeight**: Returns the height in pixels of characters in the current font.

```
Int16 FntCharHeight (void)
```

- **FntCharsInWidth**: Determines the number of bytes of characters in a given string that will fit within a certain pixel width using the current font.

```
void FntCharsInWidth (Char const  *string, Int16
*stringWidthP,
    Int16 *stringLengthP, Boolean *fitWithinWidth)
```

- **FntCharsWidth**: Returns the width in pixels of a string drawn in the current font.

```
Int16 FntCharsWidth (Char const *chars, Int16 len)
```

- **FntCharWidth**: Returns the width in pixels of a given character in the current font.

```
Int16 FntCharWidth (Char ch)
```

- **FntDefineFont**: Makes a custom font available to an application.

```
Err FntDefineFont (FontID font, FontPtr fontP)
```

  Available only on Palm OS 3.0 or later.

- **FntDescenderHeight**: Returns the height in pixels between the baseline and the bottom of the character cell in the current font.

```
Int16 FntDescenderHeight (void)
```

- **FntGetFont**: Returns the ID of the current font.

```
FontID FntGetFont (void)
```

- **FntGetFontPtr**: Returns a pointer to the current font.

```
FontPtr FntGetFontPtr (void)
```

- **FntGetScrollValues**: Retrieves values for updating a scroll bar from a string, based on the position within the string.

```
void FntGetScrollValues (Char const *chars, UInt16 width,
    UInt16 scrollPos, UInt16 *linesP, UInt16 *topLine)
```

  Available only on Palm OS 2.0 or later.

- **FntLineHeight**: Returns the height in pixels of a line of text in the current font.

```
Int16 FntLineHeight (void)
```

- **FntLineWidth**: Returns the width in pixels of a line of text, taking tab characters into account.

```
Int16 FntLineWidth (Char const *pChars, UInt16 length)
```

- **FntSetFont**: Sets the current font, returning the ID of the current font before making the change.

```
FontID FntSetFont (FontID font)
```

- **FntWidthToOffset**: Returns the offset of a character within a string located at a given pixel position.

```
Int16 FntWidthToOffset (Char const *pChars, UInt16 length,
    Int16 pixelWidth, Boolean *leadingEdge, Int16
*truncWidth)
```

  Available only on Palm OS 3.1 or later.

- **FntWordWrap**: Returns the number of bytes of text that can be displayed within a specified width, using the current font.

```
UInt16 FntWordWrap (Char const *chars, UInt16 maxWidth)
```

  Available only on Palm OS 2.0 or later.

- **FontSelect**: Displays a dialog for font selection.

```
FontID FontSelect (FontID fontID)
```

  Available only on Palm OS 3.0 or later.

Form Functions

The Form functions provide routines for interacting with form objects.

- **FormCheckResponseFunc**: Application-defined callback function used by **FrmCustomResponseAlert** to initialize and perform cleanup of a custom response alert dialog, as well as to process and validate user input in the alert's text field.

```
Boolean FormCheckResponseFunc (Int16 button, Char *attempt)
```

  Available only on Palm OS 3.5 or later.

- **FormEventHandler**: Application-defined callback function that handles events for a particular form; **FrmSetEventHandler** installs this callback as a form's event handler.

```
Boolean FormEventHandlerType (EventType *eventP)
```

- **FormGadgetHandler**: Application-defined callback function that handles events for a particular gadget; **FrmSetGadgetHandler** installs this callback as an extended gadget's event handler.

```
Boolean (FormGadgetHandlerType)
    (struct FormGadgetType *gadgetP, UInt16 cmd, void
*paramP)
```

  Available only on Palm OS 3.5 or later.

- **FrmAlert**: Displays an alert dialog.

```
UInt16 FrmAlert (UInt16 alertId)
```

- **FrmCloseAllForms**: Sends a `frmCloseEvent` to all open forms.

```
void FrmCloseAllForms (void)
```

- **FrmCopyLabel**: Copies a string into a label.

```
void FrmCopyLabel (FormType *formP, UInt16 labelID,
    const Char  *newLabel)
```

- **FrmCopyTitle**: Copies a string into a form's title bar.

```
void FrmCopyTitle (FormType *formP, const Char *newTitle)
```

- **FrmCustomAlert**: Displays an alert dialog, substituting placeholders in the alert resource with string values.

```
UInt16 FrmCustomAlert (UInt16 alertId, const Char *s1,
    const Char *s2, const Char *s3)
```

- **FrmCustomResponseAlert**: Displays an alert dialog with a text field, allowing the user to enter text before dismissing the dialog.

```
UInt16 FrmCustomResponseAlert (UInt16 alertId, const Char *s1,
    const Char *s2, const Char *s3, Char *entryStringBuf,
    Int16 entryStringBufLength,
    FormCheckResponseFuncPtr callback)
```

  Available only on Palm OS 3.5 or later.

- **FrmDeleteForm**: Releases the memory allocated to a form.

```
void FrmDeleteForm (FormType *formP)
```

- **FrmDispatchEvent**: Dispatches an event to a form's event handler.

```
Boolean FrmDispatchEvent (EventType *eventP)
```

- **FrmDoDialog**: Displays a modal dialog and retrieves the resource ID of the button used to dismiss the dialog.

```
UInt16 FrmDoDialog (FormType *formP)
```

- **FrmDrawForm**: Draws a form, its border, and all the objects within it.

```
void FrmDrawForm (FormType *formP)
```

- **FrmEraseForm**: Erases a form from the display.

```
void FrmEraseForm (FormType *formP)
```

- **FrmGetActiveForm**: Returns a pointer to the current active form.

```
FormType *FrmGetActiveForm (void)
```

- **FrmGetActiveFormID**: Returns the ID of the current active form.

```
UInt16 FrmGetActiveFormID (void)
```

- **FrmGetControlGroupSelection**: Returns the index of the selected control in an exclusive group of push buttons or check boxes.

```
UInt16 FrmGetControlGroupSelection (FormType *formP,
    UInt8 groupNum)
```

- **FrmGetControlValue**: Returns the value of a control.

```
Int16 FrmGetControlValue (const FormType *formP,
    UInt16 controlID)
```

- **FrmGetFirstForm**: Returns a pointer to the first form in the window list.

```
FormType *FrmGetFirstForm (void)
```

- **FrmGetFocus**: Returns the index of the object in a form that has the focus.

```
UInt16 FrmGetFocus (const FormType *formP)
```

- **FrmGetFormBounds**: Retrieves a rectangle defining the bounds of a form, including its border.

```
void FrmGetFormBounds (const FormType *formP,
    RectangleType *rP)
```

- **FrmGetFormId**: Returns the resource ID of a form, given a pointer to the form.

```
UInt16 FrmGetFormId (FormType *formP)
```

- **FrmGetFormPtr**: Returns a pointer to a form, given the form's ID.

```
FormType *FrmGetFormPtr (UInt16 formId)
```

- **FrmGetGadgetData**: Retrieves the value stored in a gadget.

```
void *FrmGetGadgetData (const FormType *formP, UInt16 objIndex)
```

- **FrmGetLabel**: Returns a pointer to a label's text.

```
const Char *FrmGetLabel (FormType *formP, UInt16 labelID)
```

- **FrmGetNumberOfObjects**: Returns the number of objects contained by a form.

```
UInt16 FrmGetNumberOfObjects (const FormType *formP)
```

- **FrmGetObjectBounds**: Retrieves a rectangle defining an object's bounds within a form.

```
void FrmGetObjectBounds (const FormType *formP,
    UInt16 ObjIndex, RectangleType *rP)
```

- **FrmGetObjectId**: Returns the resource ID of an object in a form, given the object's index.

```
UInt16 FrmGetObjectId (const FormType *formP, UInt16 objIndex)
```

- **FrmGetObjectIndex**: Returns the index of an object in a form, given the object's ID.

```
UInt16 FrmGetObjectIndex (const FormType *formP, UInt16 objID)
```

- **FrmGetObjectPosition**: Retrieves the coordinates of the upper-left corner of an object, relative to the form it occupies.

```
void FrmGetObjectPosition (const FormType *formP,
    UInt16 objIndex, Coord *x, Coord *y)
```

- **FrmGetObjectPtr**: Returns a pointer to an object in a form, given the object's index.

```
void *FrmGetObjectPtr (const FormType *formP, UInt16 objIndex)
```

- **FrmGetObjectType**: Returns the type of an object.

```
FormObjectKind FrmGetObjectType (const FormType *formP,
    UInt16 objIndex)
```

- **FrmGetTitle**: Returns a pointer to a form's title bar string.

```
const Char *FrmGetTitle (const FormType *formP)
```

- **FrmGotoForm**: Closes the current form and opens another.

```
void FrmGotoForm (UInt16 formId)
```

- **FrmHelp**: Displays a help message in a modal dialog.

```
void FrmHelp (UInt16 helpMsgId)
```

- **FrmHideObject**: Erases an object from the display and sets it unusable.

```
void FrmHideObject (FormType *formP, UInt16 objIndex)
```

- **FrmInitForm**: Loads and initializes a form resource.

```
FormType *FrmInitForm (UInt16 rscID)
```

- **FrmNewBitmap**: Creates a new form bitmap at run time.

```
FormBitmapType *FrmNewBitmap (FormType **formPP, UInt16 ID,
    UInt16 rscID, Coord x, Coord y)
```

  Available only on Palm OS 3.0 or later.

- **FrmNewForm**: Creates a new form at run time.

```
FormType *FrmNewForm (UInt16 formID, const Char *titleStrP,
    Coord x, Coord y, Coord width, Coord height, Boolean modal,
    UInt16 defaultButton, UInt16 helpRscID, UInt16 menuRscID)
```

  Available only on Palm OS 3.0 or later.

- **FrmNewGadget**: Creates a new gadget at run time.

```
FormGadgetType *FrmNewGadget (FormType **formPP, UInt16 id,
    Coord x, Coord y, Coord width, Coord height)
```

  Available only on Palm OS 3.0 or later.

- **FrmNewGsi**: Creates a new Graffiti shift indicator at run time.

```
FrmGraffitiStateType *FrmNewGsi (FormType **formPP, Coord x,
    Coord y)
```

  Available only on Palm OS 3.5 or later.

- **FrmNewLabel**: Creates a new label at run time.

```
FormLabelType *FrmNewLabel (FormType **formPP, UInt16 ID,
    const Char *textP, Coord x, Coord y, FontID font)
```

  Available only on Palm OS 3.0 or later.

- **FrmPointInTitle**: Returns `true` if a given coordinate is within a form's title bar.

```
Boolean FrmPointInTitle (const FormType *formP, Coord x,
    Coord y)
```

  Available only on Palm OS 2.0 or later.

- **FrmPopupForm**: Displays a new form without first closing the current form.

```
void FrmPopupForm (UInt16 formId)
```

- **FrmRemoveObject**: Removes an object from a form.

```
Err FrmRemoveObject (FormType **formPP, UInt16 objIndex)
```

- **FrmRestoreActiveState**: Macro that restores the window and form state, as saved by the **FrmSaveActiveState** macro.

```
FrmRestoreActiveState (stateP)
```

  Available only on Palm OS 3.0 or later.

- **FrmReturnToForm**: Erases and deletes the active form and makes another form active.

```
void FrmReturnToForm (UInt16 formId)
```

- **FrmSaveActiveState**: Macro that saves the window and form state before dynamically displaying a new modal form.

```
FrmSaveActiveState (stateP)
```

  Available only on Palm OS 3.0 or later.

- **FrmSaveAllForms**: Sends a `frmSaveEvent` to all open forms.

```
void FrmSaveAllForms (void)
```

- **FrmSetActiveForm**: Sets the active form.

```
void FrmSetActiveForm (FormType *formP)
```

- **FrmSetCategoryLabel**: Sets the category label displayed on the title line of a form.

```
void FrmSetCategoryLabel (FormType *formP, UInt16 objIndex,
    Char *newLabel)
```

- **FrmSetControlGroupSelection**: Sets the selected control in an exclusive group of push buttons or check boxes.

```
void FrmSetControlGroupSelection (const FormType *formP,
    UInt8 groupNum, UInt16 controlID)
```

- **FrmSetControlValue**: Sets the value of a control and redraws the control if it is visible.

```
void FrmSetControlValue (const FormType *formP,
    UInt16 objIndex, Int16 newValue)
```

- **FrmSetEventHandler**: Sets the event handler callback function for a form.

```
void FrmSetEventHandler (FormType *formP,
    FormEventHandlerType *handler)
```

- **FrmSetFocus**: Sets a form's focus to a given object.

```
void FrmSetFocus (FormType *formP, UInt16 fieldIndex)
```

- **FrmSetGadgetData**: Sets a gadget's data value.

```
void FrmSetGadgetData (FormType *formP, UInt16 objIndex,
    const void *data)
```

- **FrmSetGadgetHandler**: Sets the event handler for an extended gadget.

```
void FrmSetGadgetHandler (FormType *formP, UInt16 objIndex,
    FormGadgetHandlerType *attrP)
```

  Available only on Palm OS 3.5 or later.

- **FrmSetMenu**: Sets a form's active menu bar.

```
void FrmSetMenu (FormType *formP, UInt16 menuRscID)
```

  Available only on Palm OS 2.0 or later.

- **FrmSetObjectBounds**: Sets an object's bounds or position based on a given rectangle.

```
void FrmSetObjectBounds (FormType *formP, UInt16 objIndex,
    const RectangleType *bounds)
```

  Available only on Palm OS 2.0 or later.

- **FrmSetObjectPosition**: Sets the window-relative coordinates of an object's upper-left corner.

```
void FrmSetObjectPosition (FormType *formP, UInt16 objIndex,
    Coord x, Coord y)
```

- **FrmSetTitle**: Sets the title for a form, redrawing the title if the form is visible.

```
void FrmSetTitle (FormType *formP, Char *newTitle)
```

- **FrmShowObject**: Sets an object to usable, drawing it if the form it occupies is visible.

```
void FrmShowObject (FormType *formP, UInt16 objIndex)
```

- **FrmUpdateForm**: Sends a `frmUpdateEvent` to a given form.

```
void FrmUpdateForm (UInt16 formId, UInt16 updateCode)
```

- **FrmUpdateScrollers**: Shows, hides, or grays out scroll arrows given a specific scroll state.

```
void FrmUpdateScrollers (FormType *formP, UInt16 upIndex,
    UInt16 downIndex, Boolean scrollableUp,
    Boolean scrollableDown)
```

- **FrmValidatePtr**: Returns `true` if the given pointer references a valid form.

```
Boolean FrmValidatePtr (const FormType *formP)
```

  Available only on Palm OS 3.0 or later. For debugging only; do not use in released applications.

- **FrmVisible**: Returns `true` if a form is visible.

```
Boolean FrmVisible (const FormType *formP)
```

Form Structures

The form functions use several structures to keep track of information about form objects.

- `FormAttrType`: Bit field that describes a form's attributes; used by the `FormType` structure.

```
typedef struct {
    UInt16 usable           :1;
    UInt16 enabled          :1;
    UInt16 visible          :1;
    UInt16 dirty            :1;
    UInt16 saveBehind       :1;
    UInt16 graffitiShift    :1;
    UInt16 globalsAvailable :1;
    UInt16 doingDialog      :1;
    UInt16 exitDialog       :1;
    UInt16 reserved         :7;
    UInt16 reserved2;
} FormAttrType;
```

- `FormBitmapType`: Describes a bitmap object.

```
typedef struct {
    FormObjAttrType  attr;
    PointType  pos;
    UInt16     rscID;
} FormBitmapType;
```

- `FormGadgetAttrType`: Bit field that describes a gadget object's attributes; used by the `FormGadgetType` structure.

```
typedef struct {
    UInt16 usable    :1;
    UInt16 extended  :1;
```

```
    UInt16 visible   :1;
    UInt16 reserved  :13;
} FormGadgetAttrType;
```

✦ `FormGadgetType`: Describes a gadget object.

```
typedef struct {
    UInt16         id;
    FormGadgetAttrType  attr;
    RectangleType  rect;
    const void     *data;
    FormGadgetHandlerType  *handler;
} FormGadgetType;
```

✦ `FormLabelType`: Describes a label object.

```
typedef struct {
    UInt16     id;
    PointType  pos;
    FormObjAttrType  attr;
    FontID     fontID;
    UInt8      reserved;
    Char       *text;
} FormLabelType;
```

✦ `FormObjAttrType`: Bit field that describes an object's attributes; used by the **FormObjectType** union.

```
typedef struct {
    UInt16 usable    :1;
    UInt16 reserved  :15;
} FormObjAttrType;
```

✦ `FormObjectKind`: Specifies the kinds of objects that may be contained in a form.

```
enum formObjects {
    frmFieldObj,
    frmControlObj,
    frmListObj,
    frmTableObj,
    frmBitmapObj,
    frmLineObj,
    frmFrameObj,
    frmRectangleObj,
    frmLabelObj,
    frmTitleObj,
    frmPopupObj,
    frmGraffitiStateObj,
    frmGadgetObj,
    frmScrollBarObj
};

typedef enum formObjects FormObjectKind;
```

- `FormObjectType`: Describes an object on a form.

```
typedef union {
    void            *ptr;
    FieldType       *field;
    ControlType     *control;
    GraphicControlType  *graphicControl;
    SliderControlType  *sliderControl;
    ListType        *list;
    TableType       *table;
    FormBitmapType  *bitmap;
    FormLabelType   *label;
    FormTitleType   *title;
    FormPopupType   *popup;
    FrmGraffitiStateType  *grfState;
    FormGadgetType  *gadget;
    ScrollBarType   *scrollBar;
} FormObjectType;
```

- `FormObjListType`: Describes one member of a form's object list, which is stored in a `FormType` structure's `objects` field.

```
typedef struct {
    FormObjectKind  objectType;
    UInt8  reserved;
    FormObjectType  object;
} FormObjListType;
```

- `FormPopupType`: Describes a pop-up list object.

```
typedef struct {
    UInt16  controlID;
    UInt16  listID;
} FormPopupType;
```

- `FormTitleType`: Describes a form's title.

```
typedef struct {
    RectangleType  rect;
    Char  *text;
} FormTitleType;
```

- `FormType`: Describes a form.

```
typedef struct {
    WindowType  window;
    UInt16     formId;
    FormAttrType  attr;
    WinHandle  bitsBehindForm;
    FormEventHandlerType  *handler;
    UInt16     focus;
    UInt16     defaultButton;
    UInt16     helpRscId;
    UInt16     menuRscId;
```

```
    UInt16     numObjects;
    FormObjListType  *objects;
} FormType;
```

- `FrmGraffitiStateType`: Describes a Graffiti shift indicator.

```
typedef struct {
    PointType  pos;
} FrmGraffitiStateType;
```

List Functions

The List functions handle list objects.

- **LstDrawDataFunc**: Application-defined callback function for performing special drawing within each item in a list.

```
void ListDrawDataFunc (Int16 itemNum, RectangleType *bounds,
    Char **itemsText)
```

- **LstDrawList**: Draws a list object if it is usable.

```
void LstDrawList (ListType *listP)
```

- **LstEraseList**: Erases a list object from the display.

```
void LstEraseList (ListType *listP)
```

- **LstGetNumberOfItems**: Returns the number of items in a list.

```
Int16 LstGetNumberOfItems (const ListType *listP)
```

- **LstGetSelection**: Returns the item number of the currently selected list item.

```
Int16 LstGetSelection (const ListType *listP)
```

- **LstGetSelectionText**: Returns a pointer to the text of the selected list item, or `NULL` if there is no selection.

```
Char  *LstGetSelectionText (const ListType *listP,
    Int16 itemNum)
```

- **LstGetVisibleItems**: Returns the number of visible items in a list.

```
Int16 LstGetVisibleItems (const ListType *listP)
```

  Available only on Palm OS 2.0 or later.

- **LstHandleEvent**: Handles an event in a list object; normally, the system calls this function for you to provide default handling of list events.

```
Boolean LstHandleEvent (ListType *listP,
    const EventType *eventP)
```

- **LstMakeItemVisible**: Scrolls a list to make a given item visible, preferably at the top of the list.

```
void LstMakeItemVisible (ListType *listP, Int16 itemNum)
```

- **LstNewList**: Dynamically creates a new list object.

```
Err LstNewList (void **formPP, UInt16 id, Coord x, Coord y,
    Coord width, Coord height, FontID font, Int16
visibleItems,
    Int16 triggerId)
```

  Available only on Palm OS 3.0 or later.

- **LstPopupList**: Displays a pop-up list and returns the item number of the item the user selects.

```
Int16 LstPopupList (ListType *listP)
```

- **LstScrollList**: Scrolls a list by a given number of items.

```
Boolean LstScrollList (ListType *listP,
    WinDirectionType direction, Int16 itemCount)
```

  Available only on Palm OS 2.0 or later.

- **LstSetDrawFunction**: Sets a callback function to perform special drawing tasks within each member of a list.

```
void LstSetDrawFunction (ListType *listP,
    ListDrawDataFuncPtr func)
```

- **LstSetHeight**: Sets the number of visible items in a list.

```
void LstSetHeight (ListType *listP, Int16 visibleItems)
```

- **LstSetListChoices**: Sets the items in a list to a given array of strings, without affecting the display of the list.

```
void LstSetListChoices (ListType *listP, Char **itemsText,
    UInt16 numItems)
```

- **LstSetPosition**: Sets the coordinates of the upper-left corner of a list.

```
void LstSetPosition (ListType *listP, Coord x, Coord y)
```

- **LstSetSelection**: Sets the selection in a list.

```
void LstSetSelection (ListType *listP, Int16 itemNum)
```

- **LstSetTopItem**: Makes a given list item the top item in the list, unless that item is on the last visible page of the list.

```
void LstSetTopItem (ListType *listP, Int16 itemNum)
```

List Structures

The list functions use a pair of structures to hold list object data.

- `ListAttrType`: Bit field that describes a list's attributes; used by the `ListType` structure.

```
typedef struct {
    UInt16 usable        :1;
    UInt16 enabled       :1;
    UInt16 visible       :1;
    UInt16 poppedUp      :1;
    UInt16 hasScrollBar  :1;
    UInt16 search        :1;
    UInt16 reserved      :2;
} ListAttrType;
```

- `ListType`: Describes a list object.

```
typedef struct ListType {
    UInt16  id;
    RectangleType  bounds;
    ListAttrType  attr;
    Char    **itemsText;
    Int16   numItems;
    Int16   currentItem;
    Int16   topItem;
    FontID  font;
    UInt8   reserved;
    WinHandle  popupWin;
    ListDrawDataFuncPtr  drawItemsCallback;
} ListType;
```

Memory Manager Functions

The Memory Manager functions handle allocation and manipulation of memory.

- **MemCardInfo**: Returns information about a memory card.

```
Err MemCardInfo (UInt16 cardNo, Char *cardNameP,
    Char *manufNameP, UInt16 *versionP, UInt32 *crDateP,
    UInt32 *romSizeP, UInt32 *ramSizeP, UInt32 *freeBytesP)
```

- **MemCmp**: Compares two blocks of memory.

```
Int16 MemCmp (const void *s1, const void *s2, Int32 numBytes)
```

- **MemHandleFree**: Disposes of a movable memory chunk.

```
Err MemHandleFree (MemHandle h)
```

- **MemHandleLock**: Locks a movable memory chunk and returns a pointer to the chunk's data.

```
MemPtr MemHandleLock (MemHandle h)
```

- **MemHandleNew**: Allocates a new movable memory chunk and returns a handle to the chunk.

```
MemHandle MemHandleNew (UInt32 size)
```

- **MemHandleResize**: Resizes a movable memory chunk.

```
Err MemHandleResize (MemHandle h, UInt32 newSize)
```

- **MemHandleSetOwner**: Sets the application that owns a movable memory chunk.

```
Err MemHandleSetOwner (MemHandle h, UInt16 owner)
```

- **MemHandleSize**: Returns the number of bytes allocated for a movable memory chunk.

```
UInt32 MemHandleSize (MemHandle h)
```

- **MemHandleUnlock**: Unlocks a movable memory chunk.

```
Err MemHandleUnlock (MemHandle h)
```

- **MemMove**: Moves one range of memory to another range in the dynamic storage area.

```
Err MemMove (void *dstP, const void *sP, Int32 numBytes)
```

- **MemPtrFree**: Macro that disposes of an unmovable memory chunk.

```
Err MemPtrFree (MemPtr p)
```

- **MemPtrNew**: Allocates a new unmovable memory chunk and returns a pointer to the chunk's data.

```
MemPtr MemPtrNew (UInt32 size)
```

- **MemPtrRecoverHandle**: Returns a handle to a movable memory chunk, given a pointer to the chunk's data.

```
MemHandle MemPtrRecoverHandle (MemPtr p)
```

- **MemPtrResize**: Resizes a memory chunk, given a pointer to the chunk's data.

```
Err MemPtrResize (MemPtr p, UInt32 newSize)
```

- **MemPtrSetOwner**: Sets the application that owns a memory chunk, given a pointer to the chunk's data.

```
Err MemPtrSetOwner (MemPtr p, UInt16 owner)
```

- **MemPtrSize**: Returns the size of a memory chunk.

```
UInt32 MemPtrSize (MemPtr p)
```

- **MemPtrUnlock**: Unlocks a memory chunk, given a pointer to the chunk's data.

```
Err MemPtrUnlock (MemPtr p)
```

- **MemSet**: Sets a given range of memory to a given byte value.

```
Err MemSet (void *dstP, Int32 numBytes, UInt8 value)
```

Menu Functions

The Menu functions deal with menus and menu bars.

- **MenuAddItem**: Adds an item to the active menu.

```
Err MenuAddItem (UInt16 positionId, UInt16 id,
Char cmd, const Char *textP)
```

- **MenuCmdBarAddButton**: Adds a button to the menu command toolbar.

```
Err MenuCmdBarAddButton (UInt8 where, UInt16 bitmapId,
    MenuCmdBarResultType resultType, UInt32 result,
    Char *nameP)
```

  Available only on Palm OS 3.5 or later.

- **MenuCmdBarDisplay**: Displays the menu command toolbar.

```
void MenuCmdBarDisplay (void)
```

  Available only on Palm OS 3.5 or later.

- **MenuCmdBarGetButtonData**: Retrieves information about a button in the menu command toolbar.

```
Boolean MenuCmdBarGetButtonData (Int16 buttonIndex,
    UInt16 *bitmapIdP, MenuCmdBarResultType *resultTypeP,
    UInt32 *resultP, Char *nameP)
```

  Available only on Palm OS 3.5 or later.

- **MenuEraseStatus**: Erases the menu command status message or menu command toolbar.

```
void MenuEraseStatus (MenuBarType *menuP)
```

- **MenuHandleEvent**: Handles an event in a menu object; normally, the system calls this function for you to provide default handling of menu events.

```
Boolean MenuHandleEvent (MenuBarType *menuP, EventType *event,
    Uint16 *error)
```

- **MenuHideItem**: Hides a menu item.

```
Boolean MenuHideItem (UInt16 id)
```

  Available only on Palm OS 3.5 or later.

- **MenuShowItem**: Shows a hidden menu item.

```
Boolean MenuShowItem (UInt16 id)
```

  Available only on Palm OS 3.5 or later.

Menu Structures

The menu functions use several structures to hold information about menu objects.

- `MenuBarAttrType`: Bit field that defines attributes values for a menu bar; used by the `MenuBarType` structure.

```
typedef struct {
    UInt16 visible        :1;
    UInt16 commandPending :1;
    UInt16 insPtEnabled   :1;
    UInt16 needsRecalc    :1;
} MenuBarAttrType;
```

- `MenuBarType`: Defines a menu bar.

```
typedef struct {
    WinHandle  barWin;
    WinHandle  bitsBehind;
    WinHandle  savedActiveWin;
    WinHandle  bitsBehindStatus;
    MenuBarAttrType  attr;
    Int16      curMenu;
    Int16      curItem;
    Int32      commandTick;
    Int16      numMenus;
    MenuPullDownPtr  menus;
} MenuBarType;
```

- `MenuCmdBarButtonType`: Defines a button in a menu command toolbar.

```
typedef struct {
    UInt16  bitmapId;
    Char    name[menuCmdBarMaxTextLength];
    MenuCmdBarResultType  resultType;
    UInt8   reserved;
    UInt32  result;
} MenuCmdBarButtonType;
```

  Available only on Palm OS 3.5 or later.

- `MenuCmdBarResultType`: Specifies the different values for `resultType` in the `MenuCmdBarButtonType` structure.

```
typedef enum {
    menuCmdBarResultNone,
    menuCmdBarResultChar,
    menuCmdBarResultMenuItem,
    menuCmdBarResultNotify
} MenuCmdBarResultType;
```

  Available only on Palm OS 3.5 or later.

✦ `MenuCmdBarType`: Defines a menu command toolbar.

```
typedef struct {
    WinHandle  bitsBehind;
    Int32    timeoutTick;
    Coord    top;
    Int16    numButtons;
    Boolean  insPtWasEnabled;
    Boolean  gsiWasEnabled;
    Boolean  feedbackMode;
    MenuCmdBarButtonType  *buttonsData;
} MenuCmdBarType;
```

Available only on Palm OS 3.5 or later.

✦ `MenuItemType`: Defines an item in a menu.

```
typedef struct {
    UInt16  id;
    Char    command;
    UInt8   hidden   :1;
    UInt8   reserved :7;
    Char    *itemStr;
} MenuItemType;
```

✦ `MenuPullDownType`: Defines a menu pulldown.

```
typedef struct {
    WinHandle  menuWin;
    RectangleType  bounds;
    WinHandle  bitsBehind;
    RectangleType  titleBounds;
    Char       *title;
    UInt16     hidden   :1;
    UInt16     numItems :15;
    MenuItemType  *items;
} MenuPullDownType;
```

Miscellaneous Functions

The Miscellaneous group contains functions that do not fall into any other group.

✦ **LocGetNumberSeparators**: Retrieves localized number separators.

```
void LocGetNumberSeparators (NumberFormatType numberFormat,
    Char *thousandSeparator, Char *decimalSeparator)
```

Available only on Palm OS 2.0 or later.

✦ **PhoneNumberLookup**: Looks up a phone number in the Address Book application.

```
void PhoneNumberLookup (FieldType *fldP)
```

Available only on Palm OS 2.0 or later.

New Serial Manager Functions

The New Serial Manager functions control serial I/O. This manager adds functions to the original Serial Manager and will eventually take its place.

- **SrmClearErr**: Clears the serial port of line errors.

```
Err SrmClearErr(UInt16 portId)
```

  Available only if the New Serial Manager feature set is present.

- **SrmClose**: Closes the serial port.

```
Err SrmClose(UInt16 portID)
```

  Available only if the New Serial Manager feature set is present.

- **SrmControl**: Retrieves or sets various serial communications values.

```
Err SrmControl(UInt16 portId, UInt16 op, void *valueP,
    UInt16 *valueLenP)
```

  Available only if the New Serial Manager feature set is present.

- **SrmGetDeviceCount**: Returns the number of serial devices available.

```
Err SrmGetDeviceCount(UInt16 *numOfDevicesP)
```

  Available only if the New Serial Manager feature set is present.

- **SrmGetDeviceInfo**: Retrieves information about a serial device.

```
Err SrmGetDeviceInfo(UInt32 deviceID,
    DeviceInfoType *deviceInfoP)
```

  Available only if the New Serial Manager feature set is present.

- **SrmGetStatus**: Retrieves the status of the serial hardware.

```
Err SrmGetStatus(UInt16 portId, UInt32 *statusFieldP,
    UInt16 *lineErrsP)
```

  Available only if the New Serial Manager feature set is present.

- **SrmOpen**: Opens a serial port with a foreground connection.

```
Err SrmOpen(UInt32 port, UInt32 baud, UInt16 *newPortIdP)
```

  Available only if the New Serial Manager feature set is present.

- **SrmOpenBackground**: Opens a serial port with a background connection.

```
Err SrmOpenBackground(UInt32 port, UInt32 baud,
    UInt16 *newPortIdP)
```

  Available only if the New Serial Manager feature set is present.

- **SrmReceive**: Receives data from the serial port.

```
UInt32 SrmReceive(UInt16 portId, void *rcvBufP, UInt32 count,
    Int32 timeout, Err *errP)
```

  Available only if the New Serial Manager feature set is present.

- **SrmReceiveCheck**: Retrieves the number of bytes in the serial receive queue.

```
Err SrmReceiveCheck(UInt16 portId, UInt32 *numBytesP)
```

  Available only if the New Serial Manager feature set is present.

- **SrmReceiveFlush**: Flushes the serial receive queue.

```
Err SrmReceiveFlush(UInt16 portId, Int32 timeout)
```

  Available only if the New Serial Manager feature set is present.

- **SrmReceiveWait**: Waits until a given number of bytes have accumulated in the receive queue, and then returns.

```
Err SrmReceiveWait(UInt16 portId, UInt32 bytes, Int32 timeout)
```

  Available only if the New Serial Manager feature set is present.

- **SrmSend**: Sends data through the serial port.

```
UInt32 SrmSend(UInt16 portId, void *bufP, UInt32 count,
    Err *errP)
```

  Available only if the New Serial Manager feature set is present.

- **SrmSendCheck**: Retrieves the number of bytes in the serial send queue.

```
Err SrmSendCheck(UInt16 portId, UInt32 *numBytesP)
```

  Available only if the New Serial Manager feature set is present.

- **SrmSendFlush**: Flushes the serial send queue.

```
Err SrmSendFlush(UInt16 portId)
```

  Available only if the New Serial Manager feature set is present.

- **SrmSendWait**: Waits until the data in the serial send queue has all been sent and then returns.

```
Err SrmSendWait(UInt16 portId)
```

  Available only if the New Serial Manager feature set is present.

- **SrmSetReceiveBuffer**: Installs a new buffer for the serial receive queue.

```
Err SrmSetReceiveBuffer(UInt16 portId, void *bufP,
    UInt16 bufSize)
```

  Available only if the New Serial Manager feature set is present.

New Serial Manager Structures

The New Serial Manager uses a pair of structures to store data for use with serial port transfers.

- `DeviceInfoType`: Holds information about a serial device; returned by the **SrmGetDeviceInfo** function.

```
typedef struct DeviceInfoType {
    UInt32 serDevCreator;
    UInt32 serDevFtrInfo;
    UInt32 serDevMaxBaudRate;
    UInt32 serDevHandshakeBaud;
    Char *serDevPortInfoStr;
    UInt8 reserved[8];
} DeviceInfoType;
```

  Available only if the New Serial Manager feature set is present.

- `SrmCtlEnum`: Specifies the different operations available to the **SrmControl** function.

```
typedef enum SrmCtlEnum {
    srmCtlFirstReserved = 0,
    srmCtlSetBaudRate,
    srmCtlGetBaudRate,
    srmCtlSetFlags,
    srmCtlGetFlags,
    srmCtlSetCtsTimeout,
    srmCtlGetCtsTimeout,
    srmCtlStartBreak,
    srmCtlStopBreak,
    srmCtlStartLocalLoopback,
    srmCtlStopLocalLoopback,
    srmCtlIrDAEnable,
    srmCtlIrDADisable,
    srmCtlRxEnable,
    srmCtlRxDisable,
    srmCtlEmuSetBlockingHook,
    srmCtlUserDef,
    srmCtlGetOptimalTransmitSize,
    srmCtlSetDTRAsserted,
    srmCtlGetDTRAsserted,
    srmCtlLAST
} SrmCtlEnum;
```

  Available only if the New Serial Manager feature set is present.

Password Functions

The Password functions manipulate the system password.

- **PwdExists**: Returns `true` if the system password is set.

  ```
  Boolean PwdExists ()
  ```

- **PwdRemove**: Removes the system password.

  ```
  void PwdRemove (void)
  ```

- **PwdSet**: Sets the system password.

  ```
  void PwdSet (Char *oldPassword, Char *newPassword)
  ```

- **PwdVerify**: Returns `true` if a given string matches the system password.

  ```
  Boolean PwdVerify (Char *string)
  ```

Preferences Functions

The Preferences functions set and retrieve system and application preference data.

- **PrefGetAppPreferences**: Retrieves an application's preference data.

  ```
  Int16 PrefGetAppPreferences (UInt32 creator, UInt16 id,
      void *prefs, UInt16 *prefsSize, Boolean saved)
  ```

  Available only on Palm OS 2.0 or later.

- **PrefGetAppPreferencesV10**: Retrieves an application's preference data.

  ```
  Boolean PrefGetAppPreferencesV10 (UInt32 type, Int16 version,
      void *prefs, UInt16 prefsSize)
  ```

  For backward compatibility with Palm OS 1.0 only.

- **PrefGetPreference**: Returns a single system preference.

  ```
  UInt32 PrefGetPreference (SystemPreferencesChoice choice)
  ```

  Available only on Palm OS 2.0 or later.

- **PrefGetPreferences**: Retrieves all system preferences.

  ```
  void PrefGetPreferences (SystemPreferencesPtr p)
  ```

- **PrefSetAppPreferences**: Sets an application's preference data.

  ```
  void PrefSetAppPreferences (UInt32 creator, UInt16 id,
      Int16 version, void *prefs, UInt16 prefsSize,
      Boolean saved)
  ```

  Available only on Palm OS 2.0 or later.

- **PrefSetAppPreferencesV10**: Sets an application's preference data.

```
void PrefSetAppPreferencesV10 (UInt32 creator, Int16 version,
    void *prefs, UInt16 prefsSize)
```

  For backward compatibility with Palm OS 1.0 only.

- **PrefSetPreference**: Sets a single system preference.

```
void PrefSetPreference (SystemPreferencesChoice choice,
    UInt32 value)
```

  Available only on Palm OS 2.0 or later.

- **PrefSetPreferences**: Sets all system preferences.

```
void PrefSetPreferences (SystemPreferencesPtr p)
```

Preferences Structure

The preferences functions use an enumerated type to identify all the preferences that the system maintains.

- `SystemPreferencesChoice`: Specifies the available system preferences.

```
typedef enum
{
    prefVersion,
    prefCountry,
    prefDateFormat,
    prefLongDateFormat,
    prefWeekStartDay,
    prefTimeFormat,
    prefNumberFormat,
    prefAutoOffDuration,
    prefSysSoundLevelV20,
    prefGameSoundLevelV20,
    prefAlarmSoundLevelV20,
    prefHidePrivateRecordsV33,
    prefDeviceLocked,
    prefLocalSyncRequiresPassword,
    prefRemoteSyncRequiresPassword,
    prefSysBatteryKind,
    prefAllowEasterEggs,
    prefMinutesWestOfGMT,
    prefDaylightSavings,
    prefRonamaticChar,
    prefHard1CharAppCreator,
    prefHard2CharAppCreator,
    prefHard3CharAppCreator,
    prefHard4CharAppCreator,
    prefCalcCharAppCreator,
```

```
    prefHardCradleCharAppCreator,
    prefLauncherAppCreator,
    prefSysPrefFlags,
    prefHardCradle2CharAppCreator,
    prefAnimationLevel,

    // Additions for Palm OS 3.0:
    prefSysSoundVolume,
    prefGameSoundVolume,
    prefAlarmSoundVolume,
    prefBeamReceive,
    prefCalibrateDigitizerAtReset,
    prefSystemKeyboardID,
    prefDefSerialPlugIn,

    // Additions for Palm OS 3.1:
    prefStayOnWhenPluggedIn,
    prefStayLitWhenPluggedIn,

    // Additions for Palm OS 3.2:
    prefAntennaCharAppCreator,

    // Additions for Palm OS 3.3:
    prefMeasurementSystem,

    // Additions for Palm OS 3.5:
    prefShowPrivateRecords,
    prefAutoOffDurationSecs

} SystemPreferencesChoice;
```

Private Records Functions

The Private Records functions display dialogs to allow the user to change whether private records are shown, hidden, or masked.

- **SecSelectViewStatus**: Displays a dialog asking the user whether to show, hide, or mask private records, then returns the user's selection from the dialog.

  ```
  privateRecordViewEnum SecSelectViewStatus (void)
  ```

  Available only on Palm OS 3.5 or later.

- **SecVerifyPW**: Displays a dialog prompting the user for the system password.

  ```
  Boolean SecVerifyPW (privateRecordViewEnum newSecLevel)
  ```

  Available only on Palm OS 3.5 or later.

Private Records Structure

The private records functions use an enumerated type to identify the different types of record security.

- `privateRecordViewEnum`: Specifies different levels of record security.

```
typedef enum {
    showPrivateRecords = 0x00,
    maskPrivateRecords,
    hidePrivateRecords
} privateRecordViewEnum;
```

Available only on Palm OS 3.5 or later.

Rectangle Functions

The Rectangle function group provides routines for manipulating rectangle structures.

- **RctCopyRectangle**: Copies a rectangle structure's values into another rectangle structure.

```
void RctCopyRectangle (const RectangleType *srcRectP,
    RectangleType *dstRectP)
```

- **RctGetIntersection**: Retrieves a rectangle that represents the intersection of two other rectangles.

```
void RctGetIntersection (const RectangleType *r1P,
    const RectangleType *r2P, RectangleType *r3P)
```

- **RctInsetRectangle**: Expands or contracts a rectangle by a given number of pixels.

```
void RctInsetRectangle (RectangleType *rP, Coord insetAmt)
```

- **RctOffsetRectangle**: Moves a rectangle to a different screen position without changing its width and height.

```
void RctOffsetRectangle (RectangleType *rP, Coord deltaX,
    Coord deltaY)
```

- **RctPtInRectangle**: Returns `true` if a given point is within the boundaries of a given rectangle.

```
Boolean RctPtInRectangle (Coord x, Coord y,
    const RectangleType *rP)
```

- **RctSetRectangle**: Sets a rectangle structure's values.

```
void RctSetRectangle (RectangleType *rP, Coord left, Coord top,
    Coord width, Coord height)
```

Rectangle Structures

- `PointType`: Defines a point.

```
typedef struct PointType {
    Coord  x;
    Coord  y;
} PointType;
```

- `RectangleType`: Defines a rectangle.

```
typedef struct RectangleType {
    PointType  topLeft;
    PointType  extent;
} RectangleType;
```

Scroll Bar Functions

The Scroll Bar functions manage scroll bars.

- **SclDrawScrollBar**: Draws a scroll bar.

```
void SclDrawScrollBar (const ScrollBarPtr bar)
```

  Available only on Palm OS 2.0 or later.

- **SclGetScrollBar**: Retrieves a scroll bar's values.

```
void SclGetScrollBar (const ScrollBarPtr bar, Int16 *valueP,
    Int16 *minP, Int16 *maxP, Int16 *pageSizeP)
```

  Available only on Palm OS 2.0 or later.

- **SclHandleEvent**: Handles an event in a scroll bar object; normally, the system calls this function for you to provide default handling of scroll bar events.

```
Boolean SclHandleEvent (const ScrollBarPtr bar,
    const EventType *event)
```

  Available only on Palm OS 2.0 or later.

- **SclSetScrollBar**: Sets a scroll bar's values.

```
void SclSetScrollBar (const ScrollBarPtr bar, Int16 value,
    const Int16 min, const Int16 max, const Int16 pageSize)
```

  Available only on Palm OS 2.0 or later.

Scroll Bar Structures

- `ScrollBarAttrType`: Bit field that defines a scroll bar's attributes; used in the `ScrollBarType` structure.

```
typedef struct {
    UInt16 usable       :1;
```

```
    UInt16 visible      :1;
    UInt16 hilighted    :1;
    UInt16 shown        :1;
    UInt16 activeRegion :4;
} ScrollBarAttrType;
```

Available only on Palm OS 2.0 or later.

- `ScrollBarType`: Defines a scroll bar object.

```
typedef struct {
    RectangleType  bounds;
    UInt16  id;
    ScrollBarAttrType  attr;
    Int16   value;
    Int16   minValue;
    Int16   maxValue;
    Int16   pageSize;
    Int16   penPosInCar;
    Int16   savePos;
} ScrollBarType;
```

Available only on Palm OS 2.0 or later.

Serial Manager Functions

The Serial Manager functions control serial I/O. This manager will be phased out in favor of the New Serial Manager.

- **SerClearErr**: Clears the serial port of line errors.

```
Err SerClearErr (UInt16 refNum)
```

- **SerClose**: Closes the serial port.

```
Err SerClose (UInt16 refNum)
```

- **SerControl**: Retrieves or sets various serial communications values, and also performs special serial port operations.

```
Err SerControl (UInt16 refNum, UInt16 op, void *valueP,
    UInt16 *valueLenP)
```

Available only on Palm OS 2.0 or later.

- **SerGetSettings**: Retrieves information about the serial port.

```
Err SerGetSettings (UInt16 refNum, SerSettingsPtr settingsP)
```

- **SerGetStatus**: Retrieves the status of the serial hardware.

```
UInt16 SerGetStatus (UInt16 refNum, Boolean *ctsOnP,
    Boolean *dsrOnP)
```

- **SerOpen**: Opens a serial port with a foreground connection.

```
Err SerOpen (UInt16 refNum, UInt16 port, UInt32 baud)
```

- **SerReceive**: Receives data from the serial port.

```
UInt32 SerReceive (UInt16 refNum, void *bufP, UInt32 count,
    Int32 timeout, Err *errP)
```

  Available only on Palm OS 2.0 or later.

- **SerReceive10**: Receives data from the serial port.

```
Err SerReceive10 (UInt16 refNum, void *bufP, UInt32 bytes,
    Int32 timeout)
```

  For backward compatibility with Palm OS 1.0 only.

- **SerReceiveCheck**: Retrieves the number of bytes in the serial receive queue.

```
Err SerReceiveCheck (UInt16 refNum, UInt32 *numBytesP)
```

- **SerReceiveFlush**: Flushes the serial receive queue.

```
void SerReceiveFlush (UInt16 refNum, Int32 timeout)
```

- **SerReceiveWait**: Waits until a given number of bytes have accumulated in the receive queue, and then returns.

```
Err SerReceiveWait (UInt16 refNum, UInt32 bytes, Int32 timeout)
```

- **SerSend**: Sends data through the serial port.

```
UInt32 SerSend (UInt16 refNum, void *bufP, UInt32 count,
    Err *errP
```

  Available only on Palm OS 2.0 or later.

- **SerSend10**: Sends data through the serial port.

```
Err SerSend10 (UInt16 refNum, void *bufP, UInt32 size)
```

  For backward compatibility with Palm OS 1.0 only.

- **SerSendFlush**: Flushes the serial send queue.

```
Err SerSendFlush (UInt16 refNum)
```

- **SerSendWait**: Waits until the data in the serial send queue has all been sent, and then returns.

```
Err SerSendWait (UInt16 refNum, Int32 timeout)
```

- **SerSetReceiveBuffer**: Installs a new buffer for the serial receive queue.

```
Err SerSetReceiveBuffer (UInt16 refNum, void *bufP,
    UInt16 bufSize)
```

- **SerSetSettings**: Sets information about the serial port.

```
Err SerSetSettings (UInt16 refNum, SerSettingsPtr settingsP)
```

Serial Manager Structures

- `SerCtlEnum`: Specifies the control operations available for the **SerControl** function.

```
typedef enum SerCtlEnum {
    serCtlFirstReserved = 0,
    serCtlStartBreak,
    serCtlStopBreak,
    serCtlBreakStatus,
    serCtlStartLocalLoopback,
    serCtlStopLocalLoopback,
    serCtlMaxBaud,
    serCtlHandshakeThreshold,
    serCtlEmuSetBlockingHook,
    serCtlIrDAEnable,
    serCtlIrDADisable,
    serCtlIrScanningOn,
    serCtlIrScanningOff,
    serCtlRxEnable,
    serCtlRxDisable,
    serCtlLAST
} SerCtlEnum;
```

  Available only on Palm OS 2.0 or later.

- `SerSettingsType`: Defines a structure used by **SerGetSettings** and **SerSet Settings** to hold information about the serial port.

```
typedef struct SerSettingsType {
    UInt32  baudRate;
    UInt32  flags;
    Int32   ctsTimeout;
} SerSettingsType;
```

Sound Manager Functions

The Sound Manager functions produce sounds through the system speaker and play standard MIDI files.

- **SndDoCmd**: Plays sounds through the system speaker.

```
Err SndDoCmd (void *channelP, SndCommandPtr cmdP,
    Boolean noWait)
```

- **SndGetDefaultVolume**: Retrieves the current volume settings.

```
void SndGetDefaultVolume (UInt16 *alarmAmpP,
UInt16 *sysAmpP, UInt16 *masterAmpP)
```

- **SndPlaySystemSound**: Plays a standard system sound.

```
void SndPlaySystemSound (SndSysBeepType beepID)
```

Sound Manager Structures

- `SndCmdIDType`: Specifies a command to perform for the `SndCommandType` structure.

```
typedef enum {
    sndCmdFreqDurationAmp = 1,
    sndCmdNoteOn,
    sndCmdFrqOn,
    sndCmdQuiet
} SndCmdIDType;
```

- `SndCommandType`: Defines a sound command for use with the **SndDoCmd** function.

```
typedef struct SndCommandType {
    SndCmdIDType cmd;
    UInt8    reserved;
    Int32    param1;
    UInt16   param2;
    UInt16   param3;
} SndCommandType;
```

String Manager Functions

The String Manager functions handle manipulation of strings.

- **StrAToI**: Converts a string to an integer value.

```
Int32 StrAToI (const Char *str)
```

- **StrCaselessCompare**: Performs a case-insensitive comparison of two strings.

```
Int16 StrCaselessCompare (const Char *s1, const Char *s2)
```

- **StrCat**: Concatenates a string with another string.

```
Char* StrCat (Char *dst, const Char *src)
```

- **StrChr**: Looks for a given character within a string.

```
Char* StrChr (const Char *str, WChar chr)
```

- **StrCompare**: Performs a case-sensitive comparison of two strings.

```
Int16 StrCompare (const Char *s1, const Char *s2)
```

- **StrCopy**: Copies one string into another.

```
Char* StrCopy (Char *dst, const Char *src)
```

- **StrDelocalizeNumber**: Converts a number from a localized representation to use the standard U.S. notation (comma for thousands separator, period for decimal point).

```
Char *StrDelocalizeNumber (Char *s, Char thousandSeparator,
Char decimalSeparator)
```

  Available only on Palm OS 2.0 or later.

- **StrIToA**: Converts an integer value to a string.

```
Char* StrIToA (Char *s, Int32 i)
```

- **StrIToH**: Converts an integer value to a string containing the number's hexadecimal representation.

```
Char* StrIToH (Char *s, UInt32 i)
```

- **StrLen**: Returns the length in bytes of a string.

```
UInt16 StrLen (const Char *src)
```

- **StrLocalizeNumber**: Converts a number to a localized format, using the given thousands separator and decimal point characters.

```
Char* StrLocalizeNumber (Char *s, Char thousandSeparator,
    Char decimalSeparator)
```

  Available only on Palm OS 2.0 or later.

- **StrNCaselessCompare**: Performs a case-insensitive comparison of the first `n` bytes of two strings.

```
Int16 StrNCaselessCompare (const Char *s1, const Char *s2,
    Int32 n)
```

  Available only on Palm OS 2.0 or later.

- **StrNCat**: Concatenates one string with another string, truncating the resulting string to `n` bytes in length.

```
Char* StrNCat (Char *dst, const Char *src, Int16 n)
```

  Available only on Palm OS 2.0 or later.

- **StrNCompare**: Performs a case-sensitive comparison of the first `n` bytes of two strings.

```
Int16 StrNCompare (const Char *s1, const Char *s2, UInt32 n)
```

  Available only on Palm OS 2.0 or later.

- **StrNCopy**: Copies up to `n` bytes from a string to another string.

```
Char* StrNCopy (Char *dst, const Char *src, Int16 n)
```

  Available only on Palm OS 2.0 or later.

- **StrPrintF**: Writes formatted output to a string.

```
Int16 StrPrintF (Char *s, const Char *formatStr, ...)
```

- **StrStr**: Finds a string within another string.

```
Char* StrStr (const Char *str, const Char *token)
```

- **StrToLower**: Converts a string to lowercase.

```
Char*.StrToLower (Char *dst, const Char *src)
```

- **StrVPrintF**: Writes formatted output to a string, given a variable number of arguments.

```
Int16 StrVPrintF (Char *s, const Char *formatStr,
    _Palm_va_list argParam)
```

Available only on Palm OS 2.0 or later.

System Dialog Functions

The System Dialog functions display various system dialogs.

- **SysFatalAlert**: Displays a fatal alert dialog.

```
UInt16 SysFatalAlert (const Char *msg)
```

- **SysGraffitiReferenceDialog**: Displays the Graffiti reference dialog.

```
void SysGraffitiReferenceDialog (ReferenceType referenceType)
```

Available only on Palm OS 2.0 or later.

- **SysKeyboardDialog**: Displays the system keyboard dialog.

```
void SysKeyboardDialog (KeyboardType kbd)
```

Available only on Palm OS 2.0 or later.

- **SysKeyboardDialogV10**: Displays the system keyboard dialog.

```
void SysKeyboardDialogV10 ()
```

For backward compatibility with Palm OS 1.0 only.

System Event Manager Functions

The System Event Manager functions directly manipulate the event queue.

- **EvtAddEventToQueue**: Queues a new event.

```
void EvtAddEventToQueue (const EventType *event)
```

- **EvtAddUniqueEventToQueue**: Queues an event, replacing an existing event in the queue that has the same type and ID as the specified event.

```
void EvtAddUniqueEventToQueue (const EventType *eventP,
    UInt32 id, Boolean inPlace)
```

Available only on Palm OS 2.0 or later.

- **EvtCopyEvent**: Copies an event structure into another event structure.

```
void EvtCopyEvent (const EventType *source, EventType *dest)
```

- **EvtEventAvail**: Returns `true` if a high-level event is available in the event queue.

```
Boolean EvtEventAvail (void)
```

Available only on Palm OS 2.0 or later.

- **EvtGetEvent**: Retrieves the next available event from the event queue.

```
void EvtGetEvent (EventType *event, Int32 timeout)
```

- **EvtResetAutoOffTimer**: Resets the auto off timer to prevent the device from entering sleep mode.

```
Err EvtResetAutoOffTimer (void)
```

- **EvtSetAutoOffTimer**: Explicitly sets the auto off timer to a specific value.

```
Err EvtSetAutoOffTimer(EvtSetAutoOffCmd cmd,
    UInt16 timeoutSecs)
```

  Available only on Palm OS 3.5 or later.

- **EvtSysEventAvail**: Returns `true` if a low-level system event is available in the event queue.

```
Boolean EvtSysEventAvail (Boolean ignorePenUps)
```

  Available only on Palm OS 2.0 or later.

System Event Manager Structures

- `EventType`: The `EventType` structure, used by many of the System Event Manager functions, is described later in this appendix in the "Events" section.
- `EvtSetAutoOffCmd`: Specifies different commands used by the **EvtSetAutoOff Timer** function.

```
typedef enum
{
    SetAtLeast,
    SetExactly,
    SetAtMost,
    SetDefault,
    ResetTimer
} EvtSetAutoOffCmd;
```

  Available only on Palm OS 3.5 or later.

System Manager Functions

The System Manager routines allow direct access to many low-level system functions.

- **SysAppLaunch**: Launches an application as a subroutine of the calling application.

```
Err SysAppLaunch (UInt16 cardNo, LocalID dbID,
UInt16 launchFlags, UInt16 cmd, MemPtr cmdPBP, UInt32 *resultP)
```

✦ **SysBatteryInfo**: Retrieves battery information.

```
UInt16 SysBatteryInfo (Boolean set, UInt16 *warnThresholdP,
    UInt16 *criticalThresholdP, UInt16 *maxTicksP,
    SysBatteryKind *kindP, Boolean *pluggedIn, UInt8
*percentP)
```

Available only on Palm OS 3.0 or later.

✦ **SysBatteryInfoV20**: Retrieves battery information.

```
UInt16 SysBatteryInfoV20 (Boolean set, UInt16 *warnThresholdP,
    UInt16 *criticalThresholdP, UInt16 *maxTicksP,
    SysBatteryKind *kindP, Boolean *pluggedIn)
```

For backward compatibility with Palm OS 2.0 only.

✦ **SysBinarySearch**: Performs a binary search of a sorted array for a specific value, using a callback comparison function.

```
Boolean SysBinarySearch (void const *baseP,
    const UInt16 numOfElements, const Int16 width,
    SearchFuncPtr searchF, void const *searchData,
    const Int32 other, Int32 *position,
    const Boolean findFirst)
```

Available only on Palm OS 2.0 or later.

✦ **SysBroadcastActionCode**: Sends a launch code to the latest version of every installed application.

```
Err SysBroadcastActionCode (UInt16 cmd, MemPtr cmdPBP)
```

✦ **SysCopyStringResource**: Copies a string resource into a string.

```
void SysCopyStringResource (Char *string, Int16 theID)
```

✦ **SysFormPointerArrayToStrings**: Converts a packed block of strings into an array of pointers to strings.

```
MemHandle SysFormPointerArrayToStrings (Char *c,
    Int16 stringCount)
```

✦ **SysGetOSVersionString**: Returns a string containing the version of the Palm OS.

```
Char* SysGetOSVersionString()
```

Available only on Palm OS 3.0 or later.

✦ **SysGetROMToken**: Retrieves a value from ROM.

```
Err SysGetROMToken (UInt16 cardNo, UInt32 token, UInt8 **dataP,
    UInt16 *sizeP)
```

Available only on Palm OS 3.0 or later.

✦ **SysHandleEvent**: Performs default processing of system events.

```
Boolean SysHandleEvent (EventPtr eventP)
```

- **SysInsertionSort**: Sorts elements of an array using an insertion sort and a callback comparison function.

```
void SysInsertionSort (void *baseP, Int16 numOfElements,
    Int16 width, const CmpFuncPtr comparF, const Int32 other)
```

- **SysLibFind**: Retrieves a reference to a loaded library.

```
Err SysLibFind (const Char *nameP, UInt16 *refNumP)
```

- **SysLibLoad**: Loads a library.

```
Err SysLibLoad (UInt32 libType, UInt32 libCreator,
    UInt16 *refNumP)
```

  Available only on Palm OS 2.0 or later.

- **SysLibRemove**: Unloads a library loaded with **SysLibLoad**.

```
Err SysLibRemove (UInt16 refNum)
```

  Available only on Palm OS 2.0 or later.

- **SysQSort**: Sorts elements of an array using a quicksort algorithm and a callback comparison function.

```
void SysQSort (void *baseP, Int16 numOfElements, Int16 width,
    const CmpFuncPtr comparF, const Int32 other)
```

- **SysRandom**: Returns a random number between `0` and `sysRandomMax`.

```
Int16 SysRandom (UInt32 newSeed)
```

- **SysReset**: Performs a soft reset.

```
void SysReset (void)
```

- **SysStringByIndex**: Copies a string out of a string list resource, given the string's index within the string list.

```
Char* SysStringByIndex (UInt16 resID, UInt16 index, Char *strP,
    UInt16 maxLen)
```

  Available only on Palm OS 2.0 or later.

- **SysTaskDelay**: Puts the processor in doze mode for a given number of system ticks.

```
Err SysTaskDelay (Int32 delay)
```

- **SysTicksPerSecond**: Returns the number of system ticks per second.

```
UInt16 SysTicksPerSecond (void)
```

  Available only on Palm OS 2.0 or later.

- **SysUIAppSwitch**: Quits the current application and launches another.

```
Err SysUIAppSwitch (UInt16 cardNo, LocalID dbID, UInt16 cmd,
    MemPtr cmdPBP)
```

Table Functions

The Table function group provides routines for managing tables and their contents.

- **TableDrawItemFuncType**: Application-defined callback function that draws a custom table item.

```
void TableDrawItemFuncType (void *tableP, Int16 row,
    Int16 column, RectangleType *bounds)
```

- **TableLoadDataFuncType**: Application-defined callback function that loads data into a table's text fields.

```
Err TableLoadDataFuncType (void *tableP, Int16 row,
    Int16 column, Boolean editable, MemHandle *dataH,
    Int16 *dataOffset, Int16 *dataSize, FieldPtr fld)
```

- **TableSaveDataFuncType**: Application-defined callback function that saves data from a table's text fields.

```
Boolean TableSaveDataFuncType (void *tableP, Int16 row,
    Int16 column)
```

- **TblDrawTable**: Draws a table object.

```
void TblDrawTable (TableType *tableP)
```

- **TblEditing**: Returns `true` if a given table is in editing mode.

```
Boolean TblEditing (const TableType *tableP)
```

- **TblEraseTable**: Erases a table object from the display.

```
void TblEraseTable (TableType *tableP)
```

- **TblFindRowData**: Retrieves the row index from a table that contains a given value.

```
Boolean TblFindRowData (const TableType *tableP, UInt32 data,
    Int16 *rowP)
```

- **TblFindRowID**: Retrieves the row index from a table that has a given ID.

```
Boolean TblFindRowID (const TableType *tableP, UInt16 id,
    Int16 *rowP)
```

- **TblGetBounds**: Retrieves a rectangle defining the bounds of a table.

```
void TblGetBounds (const TableType *tableP, RectangleType *r)
```

- **TblGetColumnSpacing**: Returns the spacing in pixels after a given column.

```
Coord TblGetColumnSpacing (const TableType *tableP,
    Int16 column)
```

- **TblGetColumnWidth**: Returns the width in pixels of a given column.

```
Coord TblGetColumnWidth (const TableType *tableP, Int16 column)
```

- **TblGetCurrentField**: Returns a pointer to the field within a table that the user is currently editing.

```
FieldPtr TblGetCurrentField (const TableType *tableP)
```

- **TblGetItemBounds**: Retrieves a rectangle defining the bounds of a table item.

```
void TblGetItemBounds (const TableType *tableP, Int16 row,
    Int16 column, RectangleType *r)
```

- **TblGetItemFont**: Returns the ID of the font assigned to a given table item.

```
FontID TblGetItemFont (const TableType *tableP, Int16 row,
    Int16 column)
```

  Available only on Palm OS 3.0 or later.

- **TblGetItemInt**: Returns the integer value stored in a table item.

```
Int16 TblGetItemInt (const TableType *tableP, Int16 row,
    Int16 column)
```

- **TblGetItemPtr**: Returns a pointer to the value stored in a table item.

```
void * TblGetItemPtr (const TableType *tableP, Int16 row,
    Int16 column)
```

  Available only on Palm OS 3.5 or later.

- **TblGetLastUsableRow**: Returns the index of the last visible row in a table.

```
Int16 TblGetLastUsableRow (const TableType *tableP)
```

- **TblGetNumberOfRows**: Returns the maximum number of visible rows in a table.

```
Int16 TblGetNumberOfRows (const TableType *tableP)
```

- **TblGetRowData**: Returns the data value of a given row.

```
UInt32 TblGetRowData (const TableType *tableP, Int16 row)
```

- **TblGetRowHeight**: Returns the height in pixels of a given row.

```
Coord TblGetRowHeight (const TableType *tableP, Int16 row)
```

- **TblGetRowID**: Returns the ID of a row, given the row's index.

```
UInt16 TblGetRowID (const TableType *tableP, Int16 row)
```

- **TblGetSelection**: Retrieves the row and column of the currently selected table item and returns `true` if that item is highlighted.

```
Boolean TblGetSelection (const TableType *tableP, Int16 *rowP,
    Int16 *columnP)
```

- **TblGrabFocus**: Sets the table in editing mode.

```
void TblGrabFocus (TableType *tableP, Int16 row, Int16 column)
```

- **TblHandleEvent**: Handles an event in a table object; normally, the system calls this function for you to provide default handling of table events.

  ```
  Boolean TblHandleEvent (TableType *tableP, EventType *event)
  ```

- **TblHasScrollBar**: Sets the `hasScrollBar` attribute of a table object.

  ```
  void TblHasScrollBar (TableType *tableP, Boolean hasScrollBar)
  ```

  Available only on Palm OS 2.0 or later.

- **TblInsertRow**: Inserts a row into a table before a given row.

  ```
  void TblInsertRow (TableType *tableP, Int16 row)
  ```

- **TblMarkRowInvalid**: Marks a row invalid so it will be redrawn by the next **TblRedrawTable** call.

  ```
  void TblMarkRowInvalid (TableType *tableP, Int16 row)
  ```

- **TblMarkTableInvalid**: Marks the entire table invalid so it will be redrawn by the next **TblRedrawTable** call.

  ```
  void TblMarkTableInvalid (TableType *tableP)
  ```

- **TblRedrawTable**: Redraws rows in the table that have been marked invalid.

  ```
  void TblRedrawTable (TableType *tableP)
  ```

- **TblReleaseFocus**: Releases focus from the table.

  ```
  void TblReleaseFocus (TableType *tableP)
  ```

- **TblRemoveRow**: Removes a given row from a table.

  ```
  void TblRemoveRow (TableType *tableP, Int16 row)
  ```

- **TblRowInvalid**: Returns `true` if a given row is marked invalid.

  ```
  Boolean TblRowInvalid (const TableType *tableP, Int16 row)
  ```

- **TblRowMasked**: Returns `true` if a given row is masked.

  ```
  Boolean TblRowMasked (const TableType  *tableP, Int16 row)
  ```

  Available only on Palm OS 3.5 or later.

- **TblRowSelectable**: Returns `true` if a given row highlights when tapped.

  ```
  Boolean TblRowSelectable (const TableType *tableP, Int16 row)
  ```

- **TblRowUsable**: Returns `true` if a given row is usable.

  ```
  Boolean TblRowUsable (const TableType *tableP, Int16 row)
  ```

- **TblSelectItem**: Highlights the given table item, unhighlighting any currently highlighted table item.

  ```
  void TblSelectItem (TableType *tableP, Int16 row, Int16 column)
  ```

- **TblSetBounds**: Sets a table's bounds.

```
void TblSetBounds (TableType *tableP, const RectangleType *rP)
```

  Available only on Palm OS 2.0 or later.

- **TblSetColumnEditIndicator**: Sets whether a given column highlights when a table is in editing mode.

```
void TblSetColumnEditIndicator (TableType *tableP,
    Int16 column, Boolean editIndicator)
```

  Available only on Palm OS 2.0 or later.

- **TblSetColumnMasked**: Specifies whether a given column should be masked when private records are set to be masked on the handheld.

```
void TblSetColumnMasked (TableType *tableP, Int16 column,
    Boolean masked)
```

  Available only on Palm OS 3.5 or later.

- **TblSetColumnSpacing**: Sets the spacing in pixels after a given column.

```
void TblSetColumnSpacing (TableType *tableP, Int16 column,
    Coord spacing)
```

- **TblSetColumnUsable**: Sets a column to usable or unusable.

```
void TblSetColumnUsable (TableType *tableP, Int16 column,
    Boolean usable)
```

- **TblSetColumnWidth**: Sets a column's width in pixels.

```
void TblSetColumnWidth (TableType *tableP, Int16 column,
    Coord width)
```

- **TblSetCustomDrawProcedure**: Sets a callback function for drawing custom table items in a given column.

```
void TblSetCustomDrawProcedure (TableType *tableP,
    Int16 column, TableDrawItemFuncPtr drawCallback)
```

- **TblSetItemFont**: Sets the font for a given table item.

```
void TblSetItemFont (TableType *tableP, Int16 row,
    Int16 column, FontID fontID)
```

- **TblSetItemInt**: Sets the integer value of a table item.

```
void TblSetItemInt (TableType *tableP, Int16 row,
    Int16 column, Int16 value)
```

- **TblSetItemPtr**: Sets the data pointer of a table item.

```
void TblSetItemPtr (TableType  *tableP, Int16 row,
    Int16 column, void *value)
```

✦ **TblSetItemStyle**: Sets the type of item to display in a given table item.

```
void TblSetItemStyle (TableType *tableP, Int16 row,
    Int16 column, TableItemStyleType type)
```

✦ **TblSetLoadDataProcedure**: Sets a callback function for loading data into text fields in a given column.

```
void TblSetLoadDataProcedure (TableType *tableP, Int16 column,
    TableLoadDataFuncPtr loadDataCallback)
```

✦ **TblSetRowData**: Sets the integer value for a given row.

```
void TblSetRowData (TableType *tableP, Int16 row, UInt32
data)
```

✦ **TblSetRowHeight**: Sets the height in pixels of a given row.

```
void TblSetRowHeight (TableType *tableP, Int16 row,
    Coord height)
```

✦ **TblSetRowID**: Sets the ID of a given row.

```
void TblSetRowID (TableType *tableP, Int16 row, UInt16 id)
```

✦ **TblSetRowMasked**: Masks or unmasks a given row.

```
void TblSetRowMasked (TableType *tableP, Int16 row,
    Boolean masked)
```

Available only on Palm OS 3.5 or later.

✦ **TblSetRowSelectable**: Sets whether a row highlights when tapped.

```
void TblSetRowSelectable (TableType *tableP, Int16 row,
    Boolean selectable)
```

✦ **TblSetRowStaticHeight**: Sets a row's static height attribute.

```
void TblSetRowStaticHeight (TableType *tableP, Int16 row,
    Boolean staticHeight)
```

Available only on Palm OS 2.0 or later.

✦ **TblSetRowUsable**: Sets a given row to usable or unusable.

```
void TblSetRowUsable (TableType *tableP, Int16 row,
    Boolean usable)
```

✦ **TblSetSaveDataProcedure**: Sets a callback function for saving data from the text fields in a given column.

```
void TblSetSaveDataProcedure (TableType *tableP, Int16 column,
    TableSaveDataFuncPtr saveDataCallback)
```

✦ **TblUnhighlightSelection**: Unhighlights the currently selected table item.

```
void TblUnhighlightSelection (TableType *tableP)
```

Table Structures

- `TableAttrType`: Bit field that defines a table's attributes.

```
typedef struct {
    UInt16  visible      :1;
    UInt16  editable     :1;
    UInt16  editing      :1;
    UInt16  selected     :1;
    UInt16  hasScrollBar :1;
    UInt16  reserved     :11;
} TableAttrType;
```

- `TableColumnAttrType`: Bit field that defines a column's attributes.

```
typedef struct {
    Coord   width;

    UInt16  reserved1     :5;
    UInt16  masked        :1;
    UInt16  editIndicator :1;
    UInt16  usable        :1;
    UInt16  reserved2     :8;

    Coord   spacing;
    TableDrawItemFuncPtr  drawCallback;
    TableLoadDataFuncPtr  loadDataCallback;
    TableSaveDataFuncPtr  saveDataCallback;
} TableColumnAttrType;
```

- `TableItemStyleType`: Specifies the different styles available for table items.

```
typedef enum {
    checkboxTableItem,
    customTableItem,
    dateTableItem,
    labelTableItem,
    numericTableItem,
    popupTriggerTableItem,
    textTableItem,
    textWithNoteTableItem,
    timeTableItem,
    narrowTextTableItem
} tableItemStyles;
```

- `TableItemType`: Defines a table item.

```
typedef struct {
    TableItemStyleType  itemType;
    FontID  fontID;
    Int16   intValue;
    Char    *ptr;
} TableItemType;
```

✦ `TableRowAttrType`: Bit field that defines a row's attributes.

```
typedef struct {
    UInt16  id;
    Coord   height;
    UInt32  data;

    UInt16  reserved1    :7;
    UInt16  usable       :1;
    UInt16  reserved2    :4;
    UInt16  masked       :1;
    UInt16  invalid      :1;
    UInt16  staticHeight :1;
    UInt16  selectable   :1;

    UInt16  reserved3;

} TableRowAttrType;
```

✦ `TableType`: Defines a table object.

```
typedef struct {
    UInt16  id;
    RectangleType  bounds;
    TableAttrType  attr;
    Int16   numColumns;
    Int16   numRows;
    Int16   currentRow;
    Int16   currentColumn;
    Int16   topRow;
    TableColumnAttrType  *columnAttrs;
    TableRowAttrType  *rowAttrs;
    TableItemPtr  items;
    FieldType  currentField;
} TableType;
```

Text Manager Functions

The Text Manager functions manipulate text in a localization-friendly manner.

✦ **TxtByteAttr**: Retrieves the possible positions of a given byte within a multi-byte character.

```
UInt8 TxtByteAttr (UInt8 inByte)
```

Available only if the International Feature Set is present.

✦ **TxtCaselessCompare**: Performs a case-insensitive comparison of two text buffers.

```
Int16 TxtCaselessCompare (const Char *s1, UInt16 s1Len,
    UInt16 *s1MatchLen, const Char *s2, UInt16 s2Len,
    UInt16 *s2MatchLen)
```

Available only if the International Feature Set is present.

- **TxtCharAttr**: Returns a bit field containing a character's attributes.

```
UInt16 TxtCharAttr (WChar inChar)
```

  Available only if the International Feature Set is present.

- **TxtCharBounds**: Retrieves the beginning and ending positions, within a text buffer, of the multi-byte character that contains a given byte.

```
WChar TxtCharBounds (const Char *inText, UInt32 inOffset,
    UInt32 *outStart, UInt32 *outEnd)
```

  Available only if the International Feature Set is present.

- **TxtCharEncoding**: Returns the minimum encoding system required to represent a given character.

```
CharEncodingType TxtCharEncoding (WChar inChar)
```

  Available only if the International Feature Set is present.

- **TxtCharIsAlNum**: Macro that returns `true` if a given character is alphanumeric.

```
TxtCharIsAlNum (ch)
```

  Available only if the International Feature Set is present.

- **TxtCharIsAlpha**: Macro that returns `true` if a given character is a letter in an alphabet.

```
TxtCharIsAlpha (ch)
```

  Available only if the International Feature Set is present.

- **TxtCharIsCntrl**: Macro that returns `true` if a given character is a control character.

```
TxtCharIsCntrl (ch)
```

  Available only if the International Feature Set is present.

- **TxtCharIsDelim**: Macro that returns `true` if a given character is a delimiter.

```
TxtCharIsDelim (ch)
```

  Available only if the International Feature Set is present.

- **TxtCharIsDigit**: Macro that returns `true` if a given character is a decimal digit.

```
TxtCharIsDigit (ch)
```

  Available only if the International Feature Set is present.

- **TxtCharIsGraph**: Macro that returns `true` if a given character is a graphic character. Graphic characters are those that actually write something to the screen, as opposed to whitespace and control characters.

```
TxtCharIsGraph (ch)
```

  Available only if the International Feature Set is present.

- **TxtCharIsHardKey**: Macro that returns `true` if a given character is one of the hardware buttons on the handheld.

  ```
  TxtCharIsHardKey (m, ch)
  ```

  Available only if the International Feature Set is present.

- **TxtCharIsHex**: Macro that returns `true` if a given character is a hexadecimal digit.

  ```
  TxtCharIsHex (ch)
  ```

  Available only if the International Feature Set is present.

- **TxtCharIsLower**: Macro that returns `true` if a given character is a lowercase letter.

  ```
  TxtCharIsLower (ch)
  ```

  Available only if the International Feature Set is present.

- **TxtCharIsPrint**: Macro that returns `true` if a given character is printable.

  ```
  TxtCharIsPrint (ch)
  ```

  Available only if the International Feature Set is present.

- **TxtCharIsPunct**: Macro that returns `true` if a given character is a punctuation character.

  ```
  TxtCharIsPunct (ch)
  ```

  Available only if the International Feature Set is present.

- **TxtCharIsSpace**: Macro that returns `true` if a given character is a whitespace character.

  ```
  TxtCharIsSpace (ch)
  ```

  Available only if the International Feature Set is present.

- **TxtCharIsUpper**: Macro that returns `true` if a given character is an uppercase letter.

  ```
  TxtCharIsUpper (ch)
  ```

  Available only if the International Feature Set is present.

- **TxtCharIsValid**: Returns `true` if a given character is a valid character in the current character encoding.

  ```
  Boolean TxtCharIsValid (WChar inChar)
  ```

  Available only if the International Feature Set is present.

- **TxtCharSize**: Returns the number of bytes required to represent a character.

  ```
  UInt16 TxtCharSize (WChar inChar)
  ```

  Available only if the International Feature Set is present.

- **TxtCompare**: Performs a case-sensitive comparison of two text buffers.

```
Int16 TxtCompare (const Char *s1, UInt16 s1Len,
    UInt16 *s1MatchLen, const Char *s2, UInt16 s2Len,
    UInt16 *s2MatchLen)
```

  Available only if the International Feature Set is present.

- **TxtEncodingName**: Returns a string containing the official name of a given character encoding.

```
const Char *TxtEncodingName (CharEncodingType inEncoding)
```

  Available only if the International Feature Set is present.

- **TxtFindString**: Performs a case-insensitive search to find one string within another.

```
Boolean TxtFindString (const Char *inSourceStr,
    const Char *inTargetStr, UInt32 *outPos, UInt16 *outLength)
```

  Available only if the International Feature Set is present.

- **TxtGetChar**: Returns the character at a given offset within a text buffer.

```
WChar TxtGetChar (const Char *inText, UInt32 inOffset)
```

  Available only if the International Feature Set is present.

- **TxtGetNextChar**: Retrieves the character at a given offset within a text buffer and returns the size of the character in bytes.

```
UInt16 TxtGetNextChar (const Char *inText, UInt32 inOffset,
    WChar *outChar)
```

  Available only if the International Feature Set is present.

- **TxtGetPreviousChar**: Retrieves the character before a given offset within a text buffer and returns the size of the character in bytes.

```
UInt16 TxtGetPreviousChar (const Char *inText, UInt32 inOffset,
    WChar *outChar)
```

  Available only if the International Feature Set is present.

- **TxtGetTruncationOffset**: Returns the offset for truncating a text buffer so it is at most a given number of bytes in size.

```
UInt32 TxtGetTruncationOffset (const Char *inText,
    UInt32 inOffset)
```

  Available only if the International Feature Set is present.

- **TxtMaxEncoding**: Returns the higher of two character encodings.

```
CharEncodingType TxtMaxEncoding (CharEncodingType a,
    CharEncodingType b)
```

  Available only if the International Feature Set is present.

- **TxtNextCharSize**: Returns the size in bytes of the character at a given offset within a text buffer.

```
TxtNextCharSize (inText, inOffset)
```

  Available only if the International Feature Set is present.

- **TxtPreviousCharSize**: Returns the size in bytes of the character before a given offset within a text buffer.

```
TxtPreviousCharSize (inText, inOffset)
```

  Available only if the International Feature Set is present.

- **TxtSetNextChar**: Replaces a character at a given offset within a text buffer with another character and returns the size in bytes of the new character.

```
UInt16 TxtSetNextChar (Char *ioText, UInt32 inOffset,
    WChar inChar)
```

  Available only if the International Feature Set is present.

- **TxtStrEncoding**: Returns the character encoding required to represent a string.

```
CharEncodingType TxtStrEncoding (const Char *inStr)
```

  Available only if the International Feature Set is present.

- **TxtTransliterate**: Performs a transliteration on a given number of bytes within a text buffer.

```
Err TxtTransliterate (const Char *inSrcText,
    UInt16 inSrcLength, Char *outDstText, UInt16
*ioDstLength,
    TranslitOpType inOp)
```

  Available only if the International Feature Set is present.

- **TxtWordBounds**: Retrieves the beginning and ending positions, within a text buffer, of the word of text that contains a given byte.

```
Boolean TxtWordBounds (const Char *inText, UInt32 inLength,
    UInt32 inOffset, UInt32 *outStart, UInt32 *outEnd)
```

  Available only if the International Feature Set is present.

Text Manager Structure

The Text Manager uses an enumerated type to identify character encodings.

- `CharEncodingType`: Specifies the available character encodings.

```
typedef enum {
    charEncodingUnknown = 0,

    charEncodingAscii,
    charEncodingISO8859_1,
```

```
    charEncodingPalmLatin,
    charEncodingShiftJIS,
    charEncodingPalmSJIS,
    charEncodingUTF8,
    charEncodingCP1252,
    charEncodingCP932
} CharEncodingType;
```

Available only if the International Feature Set is present.

Time Manager Functions

The Time Manager functions deal with the system clock and converting between different units of time.

- **DateAdjust**: Returns a date that is a given number of days before or after a specific date.

  ```
  void DateAdjust (DatePtr dateP, Int32 adjustment)
  ```

- **DateDaysToDate**: Retrieves the date, given the number of days since January 1, 1904.

  ```
  void DateDaysToDate (UInt32 days, DatePtr date)
  ```

- **DateSecondsToDate**: Retrieves the date, given the number of seconds since 12:00 AM on January 1, 1904.

  ```
  void DateSecondsToDate (UInt32 seconds, DatePtr date)
  ```

- **DateTemplateToAscii**: Converts a date to a string, using a given template.

  ```
  UInt16 DateTemplateToAscii(const Char *templateP, UInt8 months,
      UInt8 days, UInt16 years, Char*stringP, Int16 stringLen)
  ```

  Available only on Palm OS 3.5 or later.

- **DateToAscii**: Converts a date to a string, using a given format.

  ```
  void DateToAscii (UInt8 months, UInt8 days, UInt16 years,
      DateFormatType dateFormat, Char *pString)
  ```

- **DateToDays**: Returns the number of days since January 1, 1904, for a specific date.

  ```
  UInt32 DateToDays (DateType date)
  ```

- **DateToDOWDMFormat**: Converts a date to a string in a given format, including the day of the week.

  ```
  void DateToDOWDMFormat (UInt8 month, UInt8 day, UInt16 year,
      DateFormatType dateFormat, Char  *pString)
  ```

- **DayOfMonth**: Returns a value indicating the day of the month for a given date; for example, the date January 28, 2001, returns the constant `domLastSun`, for the last Sunday in the month.

  ```
  Int16 DayOfMonth (Int16 month, Int16 day, Int16 year)
  ```

- **DayOfWeek**: Returns an integer from 0 to 6 representing the day of the week for a given date; 0 represents Sunday.

```
Int16 DayOfWeek (Int16 month, Int16 day, Int16 year)
```

- **DaysInMonth**: Returns the number of days in a month.

```
Int16 DaysInMonth (Int16 month, Int16 year)
```

- **TimAdjust**: Returns a date that is a given number of seconds before or after a specific date and time.

```
void TimAdjust (DateTimePtr dateTimeP, Int32 adjustment)
```

- **TimDateTimeToSeconds**: Returns the number of seconds since January 1, 1904, for a given date and time.

```
UInt32 TimDateTimeToSeconds (DateTimePtr dateTimeP)
```

- **TimeToAscii**: Converts a given time of day to a formatted string.

```
void TimeToAscii (UInt8 hours, UInt8 minutes,
    TimeFormatType timeFormat, Char *pString)
```

- **TimGetSeconds**: Returns the handheld clock's current time, expressed as the number of seconds since 12:00 AM on January 1, 1904.

```
UInt32 TimGetSeconds (void)
```

- **TimGetTicks**: Returns the number of system ticks since the user last performed a soft or hard reset.

```
UInt32 TimGetTicks (void)
```

- **TimSecondsToDateTime**: Retrieves the date and time, given the number of seconds since 12:00 AM on January 1, 1904.

```
void TimSecondsToDateTime (UInt32 seconds,
    DateTimePtr dateTimeP)
```

- **TimSetSeconds**: Sets the handheld's clock, given a date and time expressed as the number of seconds since 12:00 AM on January 1, 1904.

```
void TimSetSeconds (UInt32 seconds)
```

Time Manager Structures

- `DateFormatType`: Specifies the different date formats available to the **DateToAscii** and **DateToDOWDMFormat** functions.

```
typedef enum {
    dfMDYWithSlashes,   // 12/31/95
    dfDMYWithSlashes,   // 31/12/95
    dfDMYWithDots,      // 31.12.95
    dfDMYWithDashes,    // 31-12-95
    dfYMDWithSlashes,   // 95/12/31
    dfYMDWithDots,      // 95.12.31
    dfYMDWithDashes,    // 95-12-31
```

```
    dfMDYLongWithComma, // Dec 31, 1995
    dfDMYLong,          // 31 Dec 1995
    dfDMYLongWithDot,   // 31. Dec 1995
    dfDMYLongNoDay,     // Dec 1995
    dfDMYLongWithComma, // 31 Dec, 1995
    dfYMDLongWithDot,   // 1995.12.31
    dfYMDLongWithSpace, // 1995 Dec 31

    dfMYMed,            // Dec '95
    dfMYMedNoPost       // Dec 95
} DateFormatType;
```

- `DateTimeType`: Defines a date and time value.

```
typedef struct {
    Int16 second;
    Int16 minute;
    Int16 hour;
    Int16 day;
    Int16 month;
    Int16 year;
    Int16 weekDay;  // Days since Sunday (0 to 6)
} DateTimeType;
```

- `DateType`: Defines a date value.

```
typedef struct {
    UInt16 year  :7; // years since 1904 (MAC format)
    UInt16 month :4;
    UInt16 day   :5;
} DateType;
```

- `DaylightSavingsTypes`: Specifies different styles of calculating Daylight Savings Time for different locales.

```
typedef enum {
    dsNone,            // DST not observed
    dsUSA,             // United States DST
    dsAustralia,       // Australian DST
    dsWesternEuropean, // Western European DST
    dsMiddleEuropean,  // Middle European DST
    dsEasternEuropean, // Eastern European DST
    dsGreatBritain,    // Great Britain and Eire DST
    dsRumania,         // Rumanian DST
    dsTurkey,          // Turkish DST
    dsAustraliaShifted // Australian DST with shift in 1986
} DaylightSavingsTypes;
```

- `DayOfWeekType`: Specifies the return values for the **DayOfMonth** function.

```
typedef enum {
    dom1stSun, dom1stMon, dom1stTue, dom1stWen, dom1stThu,
        dom1stFri, dom1stSat,
    dom2ndSun, dom2ndMon, dom2ndTue, dom2ndWen, dom2ndThu,
        dom2ndFri, dom2ndSat,
    dom3rdSun, dom3rdMon, dom3rdTue, dom3rdWen, dom3rdThu,
```

```
        dom3rdFri, dom3rdSat,
    dom4thSun, dom4thMon, dom4thTue, dom4thWen, dom4thThu,
        dom4thFri, dom4thSat,
    domLastSun, domLastMon, domLastTue, domLastWen,
domLastThu,
        domLastFri, domLastSat
} DayOfWeekType;
```

- `TimeFormatType`: Specifies time formats for the **TimeToAscii** function.

```
typedef enum {
    tfColon,
    tfColonAMPM, // 1:00 pm
    tfColon24h,  // 13:00
    tfDot,
    tfDotAMPM,   // 1.00 pm
    tfDot24h,    // 13.00
    tfHoursAMPM, // 1 pm
    tfHours24h,  // 13
    tfComma24h   // 13,00
} TimeFormatType;
```

- `TimeType`: Defines a time value.

```
typedef struct {
    UInt8  hours;
    UInt8  minutes;
} TimeType;
```

UI Color List Functions

The UI Color List function group allows an application to set and retrie user interface colors.

- **UIColorGetTableEntryIndex**: Returns the color index in the current palette for a given user interface color.

```
IndexedColorType UIColorGetTableEntryIndex
    (UIColorTableEntries which)
```

Available only on Palm OS 3.5 or later.

- **UIColorGetTableEntryRGB**: Retrieves the RGB value for a given user interface color.

```
void UIColorGetTableEntryRGB
    (UIColorTableEntries which, RGBColorType *rgbP)
```

Available only on Palm OS 3.5 or later.

- **UIColorSetTableEntry**: Sets a user interface color.

```
Err UIColorSetTableEntry (UIColorTableEntries which,
    const RGBColorType *rgbP)
```

Available only on Palm OS 3.5 or later.

UI Color List Structure

The UI color list functions use an enumerated type to identify the colors that may be assigned to different user interface elements.

- `UIColorTableEntries`: Specifies the color values that the system uses to draw various user interface objects.

```
typedef enum UIColorTableEntries {
    UIObjectFrame = 0,
    UIObjectFill,
    UIObjectForeground,
    UIObjectSelectedFill,
    UIObjectSelectedForeground,

    UIMenuFrame,
    UIMenuFill,
    UIMenuForeground,
    UIMenuSelectedFill,
    UIMenuSelectedForeground,

    UIFieldBackground,
    UIFieldText,
    UIFieldTextLines,
    UIFieldCaret,
    UIFieldTextHighlightBackground,
    UIFieldTextHighlightForeground,
    UIFieldFepRawText,
    UIFieldFepRawBackground,
    UIFieldFepConvertedText,
    UIFieldFepConvertedBackground,
    UIFieldFepUnderline,

    UIFormFrame,
    UIFormFill,

    UIDialogFrame,
    UIDialogFill,

    UIAlertFrame,
    UIAlertFill,

    UIOK,
    UICaution,
    UIWarning,

    UILastColorTableEntry
} UIColorTableEntries;
```

Available only on Palm OS 3.5 or later.

UI Control Functions

The UI Control functions display various dialogs for changing user interface settings, such as brightness, contrast, and color.

- **UIBrightnessAdjust**: Displays a dialog for adjusting the screen's brightness level.

```
void UIBrightnessAdjust()
```

  Available only on Palm OS 3.5 or later.

- **UIContrastAdjust**: Displays a dialog for adjusting the screen's contrast.

```
void UIContrastAdjust()
```

  Available only on Palm OS 3.1 or later.

- **UIPickColor**: Displays a dialog for picking a color.

```
Boolean UIPickColor (IndexedColorType *indexP,
    RGBColorType *rgbP, UIPickColorStartType start,
    const Char *titleP, const Char *tipP)
```

  Available only on Palm OS 3.5 or later.

Windows Functions

The Windows functions handle display and manipulation of windows, as well as drawing in those windows.

- **WinCreateWindow**: Creates a new window and adds it to the window list.

```
WinHandle WinCreateWindow (RectangleType *bounds,
    FrameType frame, Boolean modal, Boolean focusable,
    UInt16 *error)
```

- **WinDeleteWindow**: Frees the memory used by a window and removes the window from the window list.

```
void WinDeleteWindow (WinHandle winHandle, Boolean eraseIt)
```

- **WinDrawBitmap**: Draws a bitmap at a given set of coordinates.

```
void WinDrawBitmap (BitmapPtr bitmapP, Coord x, Coord y)
```

- **WinDrawChar**: Draws a given character in the foreground color.

```
void WinDrawChar (WChar theChar, Coord x, Coord y)
```

- **WinDrawChars**: Draws a string of characters in the foreground color.

```
void WinDrawChars (const Char *chars, Int16 len, Coord x,
    Coord y)
```

- **WinDrawGrayLine**: Draws a dotted line in the foreground color.

```
void WinDrawGrayLine (Coord x1, Coord y1, Coord x2, Coord y2)
```

- **WinDrawGrayRectangleFrame**: Draws a dotted rectangle frame in the foreground color.

```
void WinDrawGrayRectangleFrame (FrameType frame,
    RectangleType *rP)
```

- **WinDrawInvertedChars**: Draws a string of characters with their foreground and background colors reversed.

```
void WinDrawInvertedChars (const Char *chars, Int16 len,
    Coord x, Coord y)
```

- **WinDrawLine**: Draws a line in the foreground color.

```
void WinDrawLine (Coord x1, Coord y1, Coord x2, Coord y2)
```

- **WinDrawPixel**: Draws a single pixel in the foreground color.

```
void WinDrawPixel (Coord x, Coord y)
```

  Available only on Palm OS 3.5 or later.

- **WinDrawRectangle**: Draws a solid rectangle in the foreground color.

```
void WinDrawRectangle (RectangleType *rP, UInt16 cornerDiam)
```

- **WinDrawRectangleFrame**: Draws a rectangle frame in the foreground color.

```
void WinDrawRectangleFrame (FrameType frame, RectangleType *rP)
```

- **WinDrawTruncChars**: Draws a string of characters in the foreground color, truncating them to a given width in pixels.

```
void WinDrawTruncChars (const Char *chars, Int16 len, Coord x,
    Coord y, Coord maxWidth)
```

  Available only on Palm OS 3.1 or later.

- **WinEraseChars**: Erases characters from the display.

```
void WinEraseChars (const Char *chars, Int16 len, Coord x,
    Coord y)
```

- **WinEraseLine**: Draws a line in the background color.

```
void WinEraseLine (Coord x1, Coord y1, Coord x2, Coord y2)
```

- **WinErasePixel**: Draws a pixel in the background color.

```
void WinErasePixel (Coord x, Coord y)
```

  Available only on Palm OS 3.5 or later.

- **WinEraseRectangle**: Draws a solid rectangle in the background color.

```
void WinEraseRectangle (RectangleType *rP, UInt16 cornerDiam)
```

- **WinEraseRectangleFrame**: Draws a rectangle frame in the background color.

```
void WinEraseRectangleFrame (FrameType frame,
    RectangleType *rP)
```

- **WinEraseWindow**: Erases the contents of the draw window.

```
void WinEraseWindow (void)
```

- **WinFillLine**: Draws a line in the current fill pattern.

```
void WinFillLine (Coord x1, Coord y1, Coord x2, Coord y2)
```

- **WinFillRectangle**: Draws a solid rectangle in the current fill pattern.

```
void WinFillRectangle (RectangleType *rP, UInt16 cornerDiam)
```

- **WinGetDisplayExtent**: Retrieves the width and height of the display.

```
void WinGetDisplayExtent (Coord *extentX, Coord *extentY)
```

- **WinGetDrawWindow**: Returns a handle to the current draw window.

```
WinHandle WinGetDrawWindow (void)
```

- **WinGetFramesRectangle**: Retrieves a rectangle that includes both a given rectangle frame and the area within that frame.

```
void WinGetFramesRectangle (FrameType frame, RectangleType *rP,
    RectangleType *obscuredRectP)
```

- **WinGetPattern**: Retrieves the current fill pattern.

```
void WinGetPattern (CustomPatternType *patternP)
```

- **WinGetPatternType**: Returns the current pattern type.

```
PatternType WinGetPatternType (void)
```

  Available only on Palm OS 3.5 or later.

- **WinGetPixel**: Returns the color index of a given pixel.

```
IndexedColorType WinGetPixel (Coord x, Coord y)
```

  Available only on Palm OS 3.5 or later.

- **WinGetWindowBounds**: Retrieves a rectangle defining the bounds of the current draw window in display-relative coordinates.

```
void WinGetWindowBounds (RectangleType *rP)
```

- **WinGetWindowExtent**: Retrieves the width and height of the current draw window.

```
void WinGetWindowExtent (Coord *extentX, Coord *extentY)
```

- **WinGetWindowFrameRect**: Retrieves a rectangle defining the area of the current draw window and its frame.

```
void WinGetWindowFrameRect (WinHandle winHandle,
    RectangleType *r)
```

- **WinIndexToRGB**: Converts a color index in the current palette to an RGB value.

```
void WinIndexToRGB (IndexedColorType i, RGBColorType *rgbP)
```

  Available only on Palm OS 3.5 or later.

- **WinInvertChars**: Inverts a number of characters.

```
void WinInvertChars (const Char *chars, Int16 len, Coord x,
    Coord y)
```

- **WinInvertLine**: Inverts a line.

```
void WinInvertLine (Coord x1, Coord y1, Coord x2, Coord y2)
```

- **WinInvertPixel**: Inverts a pixel.

```
void WinInvertPixel (Coord x, Coord y)
```

  Available only on Palm OS 3.5 or later.

- **WinInvertRectangle**: Inverts a rectangle.

```
void WinInvertRectangle (RectangleType *rP, UInt16
cornerDiam)
```

- **WinInvertRectangleFrame**: Inverts a rectangle frame.

```
void WinInvertRectangleFrame (FrameType frame,
    RectangleType *rP)
```

- **WinModal**: Returns `true` if a given window is modal.

```
Boolean WinModal (WinHandle winHandle)
```

- **WinPaintBitmap**: Draws a bitmap in the current draw state.

```
void WinPaintBitmap (BitmapType *bitmapP, Coord x, Coord y)
```

  Available only on Palm OS 3.5 or later.

- **WinPaintChar**: Draws a character in the current draw state.

```
void WinPaintChar (WChar theChar, Coord x, Coord y)
```

  Available only on Palm OS 3.5 or later.

- **WinPaintChars**: Draws a number of characters in the current draw state.

```
void WinPaintChars (const Char *chars, Int16 len, Coord x,
    Coord y)
```

  Available only on Palm OS 3.5 or later.

- **WinPaintLine**: Draws a line in the current draw state.

```
void WinPaintLine (Coord x1, Coord y1, Coord x2, Coord y2)
```

  Available only on Palm OS 3.5 or later.

- **WinPaintLines**: Draws a number of lines in the current draw state.

```
void WinPaintLines (UInt16 numLines, WinLineType lines[])
```

  Available only on Palm OS 3.5 or later.

- **WinPaintPixel**: Draws a pixel in the current draw state.

```
void WinPaintPixel (Coord x, Coord y)
```

  Available only on Palm OS 3.5 or later.

- **WinPaintPixels**: Draws a number of pixels in the current draw state.

```
void WinPaintPixels (UInt16 numPoints, PointType pts[])
```

  Available only on Palm OS 3.5 or later.

- **WinPaintRectangle**: Draws a solid rectangle in the current draw state.

```
void WinPaintRectangle (RectangleType *rP, UInt16 cornerDiam)
```

  Available only on Palm OS 3.5 or later.

- **WinPaintRectangleFrame**: Draws a rectangle frame in the current draw state.

```
void WinPaintRectangleFrame (FrameType frame,
    RectangleType *rP)
```

  Available only on Palm OS 3.5 or later.

- **WinPalette**: Sets or retrieves the draw window's palette.

```
Err WinPalette (UInt8 operation, Int16 startIndex,
    UInt16 paletteEntries, RGBColorType *tableP)
```

  Available only on Palm OS 3.5 or later.

- **WinPopDrawState**: Retrieves the draw state most recently saved to the stack by **WinPushDrawState**.

```
void WinPopDrawState (void)
```

  Available only on Palm OS 3.5 or later.

- **WinPushDrawState**: Pushes the current draw state onto a stack, for later retrieval by **WinPopDrawState**.

```
void WinPushDrawState (void)
```

  Available only on Palm OS 3.5 or later.

- **WinRestoreBits**: Copies the contents of a given window to the draw window and deletes the passed window.

```
void WinRestoreBits (WinHandle winHandle, Coord destX,
    Coord destY)
```

- **WinRGBToIndex**: Converts an RGB color value to the index of the closest color in the current palette.

```
IndexedColorType WinRGBToIndex (const RGBColorType *rgbP)
```

  Available only on Palm OS 3.5 or later.

- **WinSaveBits**: Saves a rectangular region of the screen to an offscreen window and returns the new window's handle.

```
WinHandle WinSaveBits (RectangleType *sourceP, UInt16 *error)
```

- **WinScreenMode**: Sets or retrieves display parameters.

```
Err WinScreenMode (WinScreenModeOperation operation,
    UInt32 *widthP, UInt32 *heightP, UInt32 *depthP,
    Boolean *enableColorP)
```

  Available only on Palm OS 3.5 or later.

- **WinScrollRectangle**: Scrolls a rectangular region of the draw window.

```
void WinScrollRectangle (RectangleType *rP,
    WinDirectionType direction, Coord distance,
    RectangleType *vacatedP)
```

- **WinSetActiveWindow**: Makes a window the active window.

```
void WinSetActiveWindow (WinHandle winHandle)
```

- **WinSetBackColor**: Sets the background color.

```
IndexedColorType WinSetBackColor (IndexedColorType backColor)
```

  Available only on Palm OS 3.5 or later.

- **WinSetDrawMode**: Sets the drawing transfer mode.

```
WinDrawOperation WinSetDrawMode (WinDrawOperation newMode)
```

  Available only on Palm OS 3.5 or later.

- **WinSetDrawWindow**: Makes a window the current draw window.

```
WinHandle WinSetDrawWindow (WinHandle winHandle)
```

- **WinSetForeColor**: Sets the foreground color.

```
IndexedColorType WinSetForeColor (IndexedColorType foreColor)
```

  Available only on Palm OS 3.5 or later.

- **WinSetPattern**: Sets the current fill pattern.

```
void WinSetPattern (const CustomPatternType *patternP)
```

- **WinSetPatternType**: Sets the current fill pattern type.

```
void WinSetPatternType (PatternType newPattern)
```

  Available only on Palm OS 3.5 or later.

- **WinSetTextColor**: Sets the color to use for drawing characters.

```
IndexedColorType WinSetTextColor (IndexedColorType textColor)
```

  Available only on Palm OS 3.5 or later.

- **WinSetUnderlineMode**: Enables or disables underlining of drawn characters.

```
UnderlineModeType WinSetUnderlineMode (UnderlineModeType mode)
```

Windows Structures

- `CustomPatternType`: Defines a custom fill pattern.

```
typedef UInt8 CustomPatternType [8];
```

- `DrawStateType`: Defines the current draw state.

```
typedef struct DrawStateType {
    WinDrawOperation  transferMode;
    PatternType  pattern;
    UnderlineModeType  underlineMode;
    FontID   fontId;
    FontPtr  font;
    CustomPatternType  patternData;
    IndexedColorType  foreColor;
    IndexedColorType  backColor;
    IndexedColorType  textColor;
    UInt8    reserved;
} DrawStateType;
```

  Available only on Palm OS 3.5 or later.

- `FrameBitsType`: Defines a window frame.

```
typedef union FrameBitsType {
    struct {
        UInt16 cornerDiam  :8;
        UInt16 reserved_3  :3;
        UInt16 threeD      :1;
        UInt16 shadowWidth :2;
        UInt16 width       :2;
    } bits;
    UInt16 word;
} FrameBitsType;
```

- `FrameType`: Defines a window frame.

```
typedef UInt16 FrameType;
```

- `IndexedColorType`: Defines a color index in a palette.

```
typedef UInt8 IndexedColorType;
```

  Available only on Palm OS 3.5 or later.

- `PatternType`: Specifies specific types of patterns.

```
typedef enum {
    blackPattern,
    whitePattern,
    grayPattern,
    customPattern
} PatternType;
```

  Available only on Palm OS 3.5 or later.

- `UnderlineModeType`: Specifies different types of text underlining.

```
typedef enum {
    noUnderline,
    grayUnderline,
    solidUnderline,
    colorUnderline
} UnderlineModeType;
```

- `WindowFlagsType`: Bit field that defines attributes for a window.

```
typedef struct WindowFlagsType {
    UInt16 format:1;
    UInt16 offscreen:1;
    UInt16 modal:1;
    UInt16 focusable:1;
    UInt16 enabled:1;
    UInt16 visible:1;
    UInt16 dialog:1;
    UInt16 freeBitmap:1;
    UInt16 reserved :8;
} WindowFlagsType;
```

- `WindowType`: Defines a window.

```
typedef struct WindowType {
    Coord          displayWidthV20;
    Coord          displayHeightV20;
    void           *displayAddrV20;
    WindowFlagsType  windowFlags;
    RectangleType  windowBounds;
    AbsRectType    clippingBounds;
    BitmapPtr      bitmapP;
    FrameBitsType  frameType;
    DrawStateType  *drawStateP;
    struct WindowType  *nextWindow;
} WindowType;
```

- `WinDrawOperation`: Specifies different transfer modes for color drawing.

```
typedef enum {
    winPaint,
    winErase,
    winMask,
    winInvert,
    winOverlay,
    winPaintInverse,
    winSwap
} WinDrawOperation;
```

  Available only on Palm OS 3.5 or later.

- `WinLineType`: Defines a line.

```
typedef struct WinLineType {
    Coord x1;
    Coord y1;
```

```
    Coord x2;
    Coord y2;
} WinLineType;
```

Available only on Palm OS 3.5 or later.

Events

The Palm OS identifies an event using a constant value from the `enumEvents` enumerated type. Also, the system defines each event with an `EventType` structure, which contains some data common to all events, as well as a union containing data specific to different types of events. This section of Appendix A lists the declarations of both `enumEvents` and `EventType`, followed by a quick guide to each of the events.

- `enumEvents`: The `enumEvents` enumerated type provides constants for all of the events available in the Palm OS:

```
typedef enum {
    nilEvent = 0,           // system level
    penDownEvent,           // system level
    penUpEvent,             // system level
    penMoveEvent,           // system level
    keyDownEvent,           // system level
    winEnterEvent,          // system level
    winExitEvent,           // system level
    ctlEnterEvent,
    ctlExitEvent,
    ctlSelectEvent,
    ctlRepeatEvent,
    lstEnterEvent,
    lstSelectEvent,
    lstExitEvent,
    popSelectEvent,
    fldEnterEvent,
    fldHeightChangedEvent,
    fldChangedEvent,
    tblEnterEvent,
    tblSelectEvent,
    daySelectEvent,
    menuEvent,
    appStopEvent = 22,      // system level
    frmLoadEvent,
    frmOpenEvent,
    frmGotoEvent,
    frmUpdateEvent,
    frmSaveEvent,
    frmCloseEvent,
    frmTitleEnterEvent,
    frmTitleSelectEvent,
    tblExitEvent,
```

```
    sclEnterEvent,
    sclExitEvent,
    sclRepeatEvent,
    tsmConfirmEvent = 35,  // system level
    tsmFepButtonEvent,     // system level
    tsmFepModeEvent,       // system level

    // Add future UI level events in this numeric space
    // to save room for new system level events
    menuCmdBarOpenEvent = 0x0800,
    menuOpenEvent,
    menuCloseEvent,
    frmGadgetEnterEvent,
    frmGadgetMiscEvent,

    // Library events
    firstINetLibEvent = 0x1000,
    firstWebLibEvent = 0x1100,

    // First available user event
    firstUserEvent = 0x6000
} eventsEnum;
```

- `EventType`: **The** `EventType` **structure contains the data for each individual event.**

```
typedef struct EventType {
    eventsEnum  eType;
    Boolean     penDown;
    UInt8       tapCount;
    Int16       screenX;
    Int16       screenY;
    union {
        struct _GenericEventType     generic;
        struct _PenUpEventType       penUp;
        struct _KeyDownEventType     keyDown;
        struct _WinEnterEventType    winEnter;
        struct _WinExitEventType     winExit;
        struct _TSMConfirmType       tsmConfirm;
        struct _TSMFepButtonType     tsmFepButton;
        struct _TSMFepModeEventType  tsmFepMode;

        struct ctlEnter {
            UInt16  controlID;
            struct ControlType  *pControl;
        } ctlEnter;

        struct ctlSelect {
            UInt16   controlID;
            struct ControlType  *pControl;
            Boolean  on;
            UInt8    reserved1;
            UInt16   value;  // used for slider controls only
        } ctlSelect;
```

```
struct ctlRepeat {
    UInt16  controlID;
    struct ControlType  *pControl;
    UInt32  time;
    UInt16  value;  // used for slider controls only
} ctlRepeat;

struct fldEnter {
    UInt16  fieldID;
    struct FieldType  *pField;
} fldEnter;

struct fldHeightChanged {
    UInt16  fieldID;
    struct FieldType  *pField;
    Int16   newHeight;
    UInt16  currentPos;
} fldHeightChanged;

struct fldChanged {
    UInt16  fieldID;
    struct FieldType  *pField;
} fldChanged;

struct fldExit {
    UInt16  fieldID;
    struct FieldType  *pField;
} fldExit;

struct lstEnter {
    UInt16  listID;
    struct ListType  *pList;
    Int16   selection;
} lstEnter;

struct lstExit {
    UInt16  listID;
    struct ListType  *pList;
} lstExit;

struct lstSelect {
    UInt16  listID;
    struct ListType  *pList;
    Int16   selection;
} lstSelect;

struct tblEnter {
    UInt16  tableID;
    struct TableType  *pTable;
    Int16   row;
    Int16   column;
} tblEnter;
```

```
struct tblExit {
    UInt16  tableID;
    struct TableType  *pTable;
    Int16   row;
    Int16   column;
} tblExit;

struct tblSelect {
    UInt16  tableID;
    struct TableType  *pTable;
    Int16   row;
    Int16   column;
} tblSelect;

struct frmLoad {
    UInt16  formID;
} frmLoad;

struct frmOpen {
    UInt16  formID;
} frmOpen;

struct frmGoto {
    UInt16  formID;
    UInt16  recordNum;
    UInt16  matchPos;
    UInt16  matchLen;
    UInt16  matchFieldNum;
    UInt32  matchCustom;
} frmGoto;

struct frmClose {
    UInt16  formID;
} frmClose;

struct frmUpdate {
    UInt16  formID;
    UInt16  updateCode;
} frmUpdate;

struct frmTitleEnter {
    UInt16  formID;
} frmTitleEnter;

struct frmTitleSelect {
    UInt16  formID;
} frmTitleSelect;

struct daySelect {
    struct DaySelectorType  *pSelector;
    Int16    selection;
    Boolean  useThisDate;
    UInt8    reserved1;
```

```
} daySelect;

struct menu {
    UInt16   itemID;
} menu;

struct popSelect {
    UInt16  controlID;
    struct ControlType  *controlP;
    UInt16  listID;
    struct ListType  *listP;
    Int16   selection;
    Int16   priorSelection;
} popSelect;

struct sclEnter {
    UInt16  scrollBarID;
    struct ScrollBarType  *pScrollBar;
} sclEnter;

struct sclExit {
    UInt16  scrollBarID;
    struct ScrollBarType  *pScrollBar;
    Int16   value;
    Int16   newValue;
} sclExit;

struct sclRepeat {
    UInt16  scrollBarID;
    struct ScrollBarType  *pScrollBar;
    Int16   value;
    Int16   newValue;
    Int32   time;
} sclRepeat;

struct menuCmdBarOpen {
    Boolean  preventFieldButtons;
    UInt8    reserved;
} menuCmdBarOpen;

struct menuOpen {
    UInt16  menuRscID;
    Int16   cause;
} menuOpen;

struct gadgetEnter {
    UInt16  gadgetID;
    struct FormGadgetType  *gadgetP;
} gadgetEnter;

struct gadgetMisc {
    UInt16  gadgetID;
    struct FormGadgetType  *gadgetP;
```

```
            UInt16  selector;
            void    *dataP;
        } gadgetMisc;

    } data;

} EventType;
```

Guide to Events

The most commonly used events are listed here in alphabetical order. If an event contains its own special data within the `EventType` structure's `data` union, the description of the event begins with the event's unique data structure.

- `appStopEvent`: This event is a request to the current application to quit. If an application does not respond to this event by exiting, the system cannot start another application.
- `ctlEnterEvent`

  ```
  struct ctlEnter {
      UInt16  controlID;
      struct ControlType  *pControl;
  } ctlEnter;
  ```

  The **CtlHandleEvent** routine queues this event when it receives a `penDown Event` within the borders of a control. The `ctlEnter` structure's fields have the following meanings:

  - `controlID`: ID of the control.
  - `pControl`: Pointer to the control object.
- `ctlExitEvent`: The **CtlHandleEvent** routine queues this event if, after the user sets the stylus down within a control's bounds, the user then moves the stylus out of the control and lifts the stylus from the screen.
- `ctlRepeatEvent`

  ```
  struct ctlRepeat {
      UInt16  controlID;
      struct ControlType  *pControl;
      UInt32  time;
      UInt16  value;
  } ctlRepeat;
  ```

  The **CtlHandleEvent** function queues this event once for every half second that the stylus remains down inside a repeating control's bounds. The `ctlRepeat` structure's fields have the following meanings:

  - `controlID`: ID of the control.
  - `pControl`: Pointer to the control object.

- `time`: System ticks count of the time when the event is added to the queue.
- `value`: Current value if the control is a feedback slider.

✦ `ctlSelectEvent`

```
struct ctlSelect {
    UInt16   controlID;
    struct ControlType  *pControl;
    Boolean  on;
    UInt8    reserved1;
    UInt16   value;  // used for slider controls only
} ctlSelect;
```

The **CtlHandleEvent** routine queues this event if the user taps and lifts the stylus within the bounds of a control. The `ctlSelect` structure's fields have the following meanings:

- `controlID`: ID of the control.
- `pControl`: Pointer to the control object.
- `on`: `true` if the control is selected.
- `reserved1`: Unused.
- `value`: Current value if the control is a slider or feedback slider.

✦ `fldChangedEvent`

```
struct fldChanged {
    UInt16  fieldID;
    struct FieldType  *pField;
} fldChanged;
```

The **FldHandleEvent** routine queues this event when the user scrolls a field by dragging the stylus across its text. The `fldChanged` structure's fields have the following meanings:

- `fieldID`: ID of the field.
- `pField`: Pointer to the field object.

✦ `fldEnterEvent`

```
struct fldEnter {
    UInt16  fieldID;
    struct FieldType  *pField;
} fldEnter;
```

The **FldHandleEvent** routine queues this event when it receives a `penDownEvent` within the borders of a field. The `fldEnter` structure's fields have the following meanings:

- `fieldID`: ID of the field.
- `pField`: Pointer to the field object.

- `fldHeightChangedEvent`

```
struct fldHeightChanged {
    UInt16  fieldID;
    struct FieldType  *pField;
    Int16   newHeight;
    UInt16  currentPos;
} fldHeightChanged;
```

The **FldHandleEvent** routine queues this event when a dynamically resizable text field's height changes because of the addition of text to or removal of it from the field. The `fldHeightChange` structure's fields have the following meanings:

- `fieldID`: ID of the field.
- `pField`: Pointer to the field object.
- `newHeight`: New number of visible lines in the field.
- `currentPos`: Current insertion point position.

- `frmCloseEvent`

```
struct frmClose {
    UInt16  formID;
} frmClose;
```

The **FrmGotoForm** and **FrmCloseAllForms** routines queue this event. If application code does not handle `frmCloseEvent`, the **FrmHandleEvent** routine handles it by erasing the specified form and freeing any memory allocated for the form. If your application does handle a `frmCloseEvent`, it is still a good idea to return `false` from the event handler and allow **FrmHandleEvent** to perform its regular cleanup duties. The `frmCloseEvent` structure's field has the following meaning:

- `formID`: ID of the form to close.

- `frmGadgetEnterEvent`

```
struct gadgetEnter {
    UInt16  gadgetID;
    struct FormGadgetType  *gadgetP;
} gadgetEnter;
```

The **FrmHandleEvent** routine queues this event when there is a `penDownEvent` within the bounds of an extended gadget. The callback event handling function installed for the gadget should handle this event. The `gadgetEnter` structure's fields have the following meanings:

- `gadgetID`: ID of the gadget.
- `gadgetP`: Pointer to the gadget object.

Available only on Palm OS 3.5 or later.

✦ `frmGadgetMiscEvent`

```
struct gadgetMisc {
    UInt16  gadgetID;
    struct FormGadgetType  *gadgetP;
    UInt16  selector;
    void    *dataP;
} gadgetMisc;
```

An application may choose to queue a `frmGadgetMiscEvent`, which the **FrmHandleEvent** routine passes along to the callback event handling function installed for an extended gadget. The `gadgetMisc` structure's fields have the following meanings:

- `gadgetID`: ID of the gadget.
- `gadgetP`: Pointer to the gadget object.
- `selector`: An integer value to pass to the gadget's event handler.
- `dataP`: A pointer to additional data to pass to the gadget's event handler.

Available only on Palm OS 3.5 or later.

✦ `frmGotoEvent`

```
struct frmGoto {
    UInt16  formID;
    UInt16  recordNum;
    UInt16  matchPos;
    UInt16  matchLen;
    UInt16  matchFieldNum;
    UInt32  matchCustom;
} frmGoto;
```

A `frmGotoEvent` is a request to an application to initialize and draw a form, with extra data specifying a text string from one of the application's records that should be highlighted on the screen. Applications can send this event to themselves when handling a `sysAppLaunchCmdGoto` event as part of implementing the global find feature. The `frmGoto` structure's fields have the following meanings:

- `formID`: ID of the form to open.
- `recordNum`: Index of the record to display.
- `matchPos`: Position within the string to start highlighting text.
- `matchLen`: Length of the matching text to highlight.
- `matchFieldNum`: Number of the field within a record that contains the matching text.
- `matchCustom`: Extra parameter for application-defined information.

- `frmLoadEvent`

```
struct frmLoad {
    UInt16  formID;
} frmLoad;
```

The **FrmGotoForm** and **FrmPopupForm** routines queue this event to request that an application load a particular form into memory. An application is responsible for handling this event, since none of the default event handlers takes care of it. The `frmLoad` structure's field has the following meaning:

  - `formID`: ID of the form to load.

- `frmOpenEvent`

```
struct frmOpen {
    UInt16  formID;
} frmOpen;
```

The **FrmGotoForm** and **FrmPopupForm** routines queue this event to request that an application initialize and draw a particular form. An application is responsible for handling this event, since none of the default event handlers takes care of it. The `frmLoad` structure's field has the following meaning:

  - `formID`: ID of the form to draw.

- `frmSaveEvent`: The **FrmSaveAllForms** routine queues this event, which is a request that an application save any data stored in one of its forms. An application is responsible for handling this event, since none of the default event handlers takes care of it.

- `frmTitleEnterEvent`

```
struct frmTitleEnter {
    UInt16  formID;
} frmTitleEnter;
```

The **FrmHandleEvent** routine queues this event when the stylus first comes down within the bounds of a form's title. The `frmTitleEnter` structure's field has the following meaning:

  - `formID`: ID of the form.

- `frmTitleSelectEvent`

```
struct frmTitleSelect {
    UInt16  formID;
} frmTitleSelect;
```

The **FrmHandleEvent** routine queues this event when the user taps and lifts the stylus within a form title's bounds. The `frmTitleSelectEvent` structure's field has the following meaning:

  - `formID`: ID of the form.

- `frmUpdateEvent`

```
struct frmUpdate {
    UInt16  formID;
```

```
    UInt16  updateCode;
} frmUpdate;
```

The **FrmUpdateForm** and **FrmEraseForm** routines queue this event, which is a request to redraw some or all of a form. An application may define its own `updateCode` values; this is useful for communicating changes made in one form to another form. If your application has one or more gadgets on a form, the application should handle a `frmUpdateEvent` by drawing the form first, and then call any gadget-drawing routines. The form event handler should then return `true` to prevent the system from performing its usual drawing behavior, which could draw over the top of the form's gadgets. The `frmUpdate` structure's fields have the following meanings:

- `formID`: ID of the form to update.
- `updateCode`: The reason for redrawing the form; **FrmEraseForm** sends the constant `frmRedrawUpdateCode` to indicate that the entire form needs to be redrawn.

✦ `keyDownEvent`

```
struct _KeyDownEventType {
    WChar   chr;
    UInt16  keyCode;
    UInt16  modifiers;
};
struct _KeyDownEventType  keyDown;
```

The system queues this event when the user enters a character via Graffiti, presses a hardware button, or taps one of the silkscreen buttons. The `_KeyDownEventType` structure's fields have the following meanings:

- `chr`: Character code.
- `keyCode`: Unused.
- `modifiers`: One or more modifiers.

✦ `lstEnterEvent`

```
struct lstEnter {
    UInt16  listID;
    struct ListType  *pList;
    Int16   selection;
} lstEnter;
```

The **LstHandleEvent** routine queues this event when it receives a `penDown Event` within the bounds of a list object. The `lstEnter` structure's fields have the following meanings:

- `listID`: ID of the list.
- `pList`: Pointer to the list object.
- `selection`: Unused.

- `lstExitEvent`

```
struct lstExit {
    UInt16  listID;
    struct ListType  *pList;
} lstExit;
```

The **LstHandleEvent** routine queues this event if, after the user sets the stylus down within a list's bounds, the user then moves the stylus out of the list and lifts the stylus from the screen. The `lstExit` structure's fields have the following meanings:

- `listID`: ID of the list.
- `pList`: Pointer to the list object.

- `lstSelectEvent`

```
struct lstSelect {
    UInt16  listID;
    struct ListType  *pList;
    Int16   selection;
} lstSelect;
```

The **LstHandleEvent** routine queues this event when the user taps and lifts the stylus within a list's bounds. The `lstSelect` structure's fields have the following meanings:

- `listID`: ID of the list.
- `pList`: Pointer to the list object.
- `selection`: Index of the selected list item.

- `menuCmdBarOpenEvent`

```
struct menuCmdBarOpen {
    Boolean  preventFieldButtons;
    UInt8    reserved;
} menuCmdBarOpen;
```

The `menuCmdBarOpenEvent` structure is available only on Palm OS 3.5 or later. The **MenuHandleEvent** routine queues this event when the user enters the command shortcut Graffiti stroke, which causes the command toolbar to appear at the bottom of the screen. An application might respond to this event by calling **MenuCmdBarAddButton** to add a button to the toolbar. The `menuCmdBarOpen` structure's fields have the following meanings:

- `preventFieldButtons`: If `true`, prevents the system from adding standard cut, copy, paste, and undo buttons to the toolbar.
- `reserved`: Unused.

- `menuEvent`

```
struct menu {
    UInt16   itemID;
} menu;
```

The **MenuHandleEvent** routine queues this event when the user selects an item from a menu, executes a menu command via Graffiti command shortcut, or taps a button in the command toolbar. The `menu` structure's field has the following meaning:

- `itemID`: ID of the selected menu command.

✦ `menuOpenEvent`

```
struct menuOpen {
    UInt16  menuRscID;
    Int16   cause;
} menuOpen;
```

The **MenuHandleEvent** routine queues this event when a new menu is initialized, which happens when the user taps the Menu silkscreen button or a form's title to open a menu. Menus remain active until **FrmSetMenu** changes a form's active menu, or a new form becomes active. The `menuOpenEvent` structure's fields have the following meanings:

- `menuRscID`: Resource ID of the menu.
- `cause`: Reason for opening the menu. There are two possible reasons for opening a menu: `menuButtonCause` and `menuCommandCause`. The `menuButtonCause` constant indicates that the user tapped the Menu silkscreen button or a form's title. The `menuCommandCause` constant specifies that the user entered the Graffiti command shortcut, which makes the menu active without displaying it.

✦ `nilEvent`: If **EvtGetEvent** is unable to return an event within the time specified in its timeout parameter, it returns a `nilEvent`.

✦ `penDownEvent`: The event manager queues this event when the stylus first touches the screen.

✦ `penMoveEvent`: The Event Manager queues this event when the user drags the stylus across the screen.

✦ `penUpEvent`

```
struct _PenUpEventType {
    PointType start;
    PointType end;
};
struct _PenUpEventType  penUpEvent;
```

The Event Manager queues this event when the user lifts the stylus from the screen. The `_PenUpEventType` structure's fields have the following meanings:

- `start`: Display-relative coordinates where the stylus first touched the screen.
- `end`: Display-relative coordinates where the user lifted the stylus from the screen.

- `popSelectEvent`

```
struct popSelect {
    UInt16  controlID;
    struct ControlType  *controlP;
    UInt16  listID;
    struct ListType  *listP;
    Int16   selection;
    Int16   priorSelection;
} popSelect;
```

The **FrmHandleEvent** routine queues this event when the user selects a pop-up list item. You application may need to override this event if a list item's field happens to be empty, as can happen when populating a list from a database. The `popSelect` structure's fields have the following meanings:

- `controlID`: ID of the pop-up trigger.
- `pControl`: Pointer to the pop-up trigger object.
- `listID`: ID of the list.
- `listP`: Pointer to the list object.
- `selection`: Index of the newly selected list item.
- `priorSelection`: Index of the selected list item before the user selected a new one.

- `sclEnterEvent`

```
struct sclEnter {
    UInt16  scrollBarID;
    struct ScrollBarType  *pScrollBar;
} sclEnter;
```

The **SclHandleEvent** routine queues this event when it receives a `penDown Event` within the borders of a scroll bar. The `sclEnter` structure's fields have the following meanings:

- `scrollBarID`: ID of the scroll bar.
- `pScrollBar`: Pointer to the scroll bar object.

- `sclExitEvent`

```
struct sclExit {
    UInt16  scrollBarID;
    struct ScrollBarType  *pScrollBar;
    Int16   value;
    Int16   newValue;
} sclExit;
```

The **SclHandleEvent** routine queues this event if, after the user set the stylus down within a scroll bar's bounds, the user then moved the stylus out of the scroll bar and lifted the stylus from the screen. Applications using scroll bars

for nondynamic scrolling should handle this event and use its `value` and `newValue` fields to determine how far the bar was scrolled. Applications that implement dynamic scrolling can ignore this event in favor of `sclRepeatEvent`. The `sclExit` structure's fields have the following meanings:

- `scrollBarID`: ID of the scroll bar.
- `pScrollBar`: Pointer to the scroll bar object.
- `value`: Initial value of the scroll bar.
- `newValue`: New value of the scroll bar.

✦ `sclRepeatEvent`

```
struct sclRepeat {
    UInt16  scrollBarID;
    struct ScrollBarType  *pScrollBar;
    Int16   value;
    Int16   newValue;
    Int32   time;
} sclRepeat;
```

The **SclHandleEvent** routine sends this event repeatedly while the user holds the stylus down on a scroll bar control. Applications that implement dynamic scrolling should watch for this event and respond accordingly. The `sclRepeat` structure's fields have the following meanings:

- `scrollBarID`: ID of the scroll bar.
- `pScrollBar`: Pointer to the scroll bar object.
- `value`: Initial value of the scroll bar.
- `newValue`: New value of the scroll bar.
- `time`: System ticks count when the event is added to the queue.

✦ `tblEnterEvent`

```
struct tblEnter {
    UInt16  tableID;
    struct TableType  *pTable;
    Int16   row;
    Int16   column;
} tblEnter;
```

The **TblHandleEvent** routine queues this event when it receives a `penDown Event` within the borders of a table. The `tblEnter` structure's fields have the following meanings:

- `tableID`: ID of the table.
- `pTable`: Pointer to the table object.
- `row`: Row where the stylus entered the table.
- `column`: Column where the stylus entered the table.

- `tblExitEvent`

```
struct tblExit {
    UInt16  tableID;
    struct TableType  *pTable;
    Int16   row;
    Int16   column;
} tblExit;
```

The **TblHandleEvent** routine queues this event if, after the user sets the stylus down within a table's bounds, the user then moves the stylus out of the table item where the stylus came down and lifts the stylus from the screen. The `tblExit` structure's fields have the following meanings:

- `tableID`: ID of the table.
- `pTable`: Pointer to the table object.
- `row`: Row where the stylus entered the table.
- `column`: Column where the stylus entered the table.

- `tblSelectEvent`

```
struct tblSelect {
    UInt16  tableID;
    struct TableType  *pTable;
    Int16   row;
    Int16   column;
} tblSelect;
```

The **TblHandleEvent** routine queues this event when the user taps and lifts the stylus within a table item. The `tblSelect` structure's fields have the following meanings:

- `tableID`: ID of the table.
- `pTable`: Pointer to the table object.
- `row`: Row of the selected table item.
- `column`: Column of the selected table item.

- `winEnterEvent`

```
struct _WinEnterEventType {
    WinHandle enterWindow;
    WinHandle exitWindow;
};
struct _WinEnterEventType  winEnterEvent;
```

The Event Manager queues this event when a window becomes the active window, which happens as a result of a call to **WinSetActiveWindow**, or when the user taps within the bounds of a window that is not the active window. The `_WinEnterEventType` structure's fields have the following meanings:

- `enterWindow`: Handle to the newly active window.
- `exitWindow`: Handle to the formerly active window.

✦ `winExitEvent`

```
struct _WinExitEventType {
    WinHandle enterWindow;
    WinHandle exitWindow;
};
struct _WinExitEventType  winExitEvent;
```

The Event Manager queues this event when a window ceases to be the active window. The `_WinExitEventType` structure's fields have the following meanings:

- `enterWindow`: Handle to the newly active window.
- `exitWindow`: Handle to the formerly active window.

Launch Codes

This section describes the launch codes available in the Palm OS.

✦ `sysAppLaunchCmdAddRecord`: This launch code allows an application to add a record to its database. On a Palm VII, you can use this launch code to add a new message to the Mail or iMessenger applications' outboxes. This launch code is available for Mail only on Palm OS 3.0 or later, and available for iMessenger only if the Wireless Internet Feature Set is present.

✦ `sysAppLaunchCmdAlarmTriggered`: This launch code allows an application to perform a quick action such as scheduling another alarm or playing a sound. The system sends this launch code right after an alarm is triggered.

✦ `sysAppLaunchCmdCountryChange`: This launch code allows an application to respond to a change in country settings on the handheld.

✦ `sysAppLaunchCmdDisplayAlarm`: This launch code allows an application to perform a long action in response to a triggered alarm, such as displaying an alarm dialog.

✦ `sysAppLaunchCmdExgAskUser`: The Exchange Manager sends this launch code to an application when incoming data is available for the application. This launch code allows an application to display its own custom dialog to prompt the user to select a category in which to store incoming data, or to skip showing a dialog entirely.

✦ `sysAppLaunchCmdExgReceiveData`: The Exchange Manager sends this launch code to an application when incoming data is available for the application, right after sending a `sysAppLaunchCmdExgAskUser` launch code. This launch code gives the application an opportunity to actually receive the data. This launch code is available only on Palm OS 3.0 or later.

✦ `sysAppLaunchCmdFind`: The system sends this launch code to each application during a global find. Applications may support the global find feature by responding to this launch code, along with `sysAppLaunchCmdGoto` and `sysAppLaunchCmdSaveData`.

- `sysAppLaunchCmdGoto`: The system sends this launch code along with `sysAppLaunchCmdFind` or `sysAppLaunchCmdExgReceiveData` to ask an application to display a specific record from its database. Global variables are available to an application when this launch code is received.
- `sysAppLaunchCmdGoToURL`: Sending this launch code to the Clipper application causes Clipper to display a specific URL. This launch code is available only if the Wireless Internet Feature Set is present.
- `sysAppLaunchCmdInitDatabase`: The Desktop Link server sends this launch code in response to a request to create a database. This launch code gives an application an opportunity to initialize the database before the HotSync Manager begins populating the new database with records.
- `sysAppLaunchCmdLookup`: The system or an application can send this launch code to request information from another application. The built-in Address Book application responds to this launch code by looking up a phone number and returning an appropriate address book entry. This launch code is similar to `sysAppLaunchCmdGoto`, but instead of launching itself and displaying a certain record, an application that responds to `sysAppLaunchCmdLookup` should provide a level of indirection, returning something from its database based on the requested information.
- `sysAppLaunchCmdNotify`: This launch code notifies applications that an event has occurred. You can use this launch code to implement your own notifications, or you can use it to respond to notifications from the system. Unlike most launch codes, which are broadcast to all the applications on the device, an application must register to receive specific `sysAppLaunchCmdNotify` launch codes before it will receive any. This launch code is available only if the Notification Feature Set is present.
- `sysAppLaunchCmdOpenDB`: Sending this launch code to the Clipper application causes it to open and display a query application. This launch code is available only if the Wireless Internet Feature Set is present.
- `sysAppLaunchCmdPanelCalledFromApp`: An application can send this launch code to a panel in the system Preferences application to pop up the panel, allow the user to make changes to some settings, and then return to the application. A panel should respond to this launch code by displaying a "Done" button and by not displaying the pop-up list of preference panels. This launch code is available only on Palm OS 2.0 or later.
- `sysAppLaunchCmdReturnFromPanel`: This launch code lets an application know that the user is done with a preferences panel that was launched from within an application. This launch code is available only on Palm OS 2.0 or later.
- `sysAppLaunchCmdSaveData`: This launch code tells an application to save any unsaved data. Usually, an application needs to respond to this event only when it is currently running, since at other times it does not have any data that has not been saved to storage yet. Applications that implement the global find facility should respond to this launch code.

- `sysAppLaunchCmdSyncNotify`: This launch code allows applications to respond to a HotSync operation. The system sends this launch code only to applications whose databases were changed during the HotSync operation, which includes newly installed applications. This launch code is available only on Palm OS 2.0 or later.
- `sysAppLaunchCmdSystemLock`: The system security application responds to this launch code when the user chooses to lock the handheld. Only applications designed to replace the internal security application need to handle this launch code. This launch code is available only on Palm OS 2.0 or later.
- `sysAppLaunchCmdSystemReset`: The system sends this launch code to every application after a hard or soft system reset.
- `sysAppLaunchCmdTimeChange`: When the user changes the time or date, the system sends this launch code to every application. Any application that handles alarms should respond to this launch code by rescheduling its current alarm.
- `sysAppLaunchCmdURLParams`: The Clipper application sends this launch code to launch another application, supplying that application with a pointer to a special URL string. This launch code is implemented only if the Wireless Internet Feature Set is present.

✦ ✦ ✦

APPENDIX B

Finding Resources for Palm OS Development

The Palm OS is large and complex, but fortunately a lot of resources are available to help Palm OS developers. This appendix is a guide to useful sources of information about Palm OS programming.

On the Web

A wealth of information about Palm OS programming is available on the World Wide Web.

Palm OS Programming Bible Site

The Web site for this book, maintained by the author, is at:

```
http://www.palmosbible.com
```

At this site, you can find errata for the book's text and source code, useful development tips, and up-to-date links to Palm OS development resources on the Web.

If you have any questions or comments for the author, feel free to send them to:

```
author@palmosbible.com
```

Official Palm OS Developer Site

Palm Computing offers a fantastic level of support for the Palm OS at the official Palm OS developer Web site:

```
http://www.palmos.com/dev
```

On this Web site, you can find documentation, tools, example source code, technical support, news, marketing support, and just about anything you could possibly need for Palm OS application development. In particular, a few parts of the site merit special attention.

Knowledge Base

The Palm OS Knowledge Base is a searchable repository of technical information about the Palm OS. To get to the Knowledge Base, select "Knowledge Base" from the Quick Index in the sidebar of the Palm OS developer site, or go straight to the following URL:

```
http://oasis.palm.com/dev/kb
```

Within the Knowledge Base are FAQs, presentations, white papers, manuals, and sample code, all available from a quick search form. This is an excellent first place to look if you have questions about some aspect of Palm OS development.

Creator ID Database

Since each application, shared library, and feature in the Palm OS must have a unique four-character creator ID, Palm Computing maintains a database of registered creator ID codes. To get to the Creator ID Database, select "Creator ID" from the Quick Index in the sidebar of the Palm OS developer site, or go straight to the following URL:

```
http://www.palmos.com/dev/tech/palmos/creatorid
```

From here, you can search for a creator ID to see if it is already taken, or you can reserve your own creator ID by registering it in the database. One of the first things you should do when making a new application is to register a creator ID for it.

Developer Tools

The Developer Tools page contains links to the most up-to-date versions of CodeWarrior for Palm Computing Platform, the PRC-Tools, the Palm OS SDK, the Palm OS Emulator, the Conduit Development Kits, and Web Clipping tools. You can get to this page by selecting "Tools" from the Quick Index in the sidebar of the Palm OS developer site, or by going directly to this URL:

```
http://www.palmos.com/dev/tech/tools
```

Developer News & Events

This page contains news about updates to the Palm OS developer Web site, development tools, and events of interest to Palm OS programmers. To get to the news page, follow the "News and Events" link in the sidebar of the developer front page, or enter this URL in your browser:

```
http://www.palmos.com/dev/news
```

Check this page regularly to see what is new in Palm OS development.

Palm Solution Provider Program

The Palm Solution Provider Program offers free technical and marketing support for Palm OS developers. You can find an overview of the program at:

```
http://www.palmos.com/dev/program
```

The single most useful part of the program, from a developer's perspective, is that the program gives you access to the Provider Pavilion, at:

```
http://www.palmos.com/dev/pavilion
```

A members-only part of the developer Web site, the Provider Pavilion offers pre-release versions of upcoming software, such as new versions of the Palm OS SDK and the Palm OS Emulator, in the Development Seeding area. In addition, a limited portion of the Palm OS source code is available for download, which can be invaluable when attempting to squash really obscure bugs.

Also, you must be a member of the Solution Provider Program to submit technical questions to the Development Support team. If you have exhausted all your options for solving a technical problem, use the link on the following page to contact Palm Computing developer support:

```
http://www.palmos.com/dev/tech/support
```

Platinum Program

Palm Computing offers a certification program to ensure that applications meet rigid standards of compatibility, quality, and usability. Run by an independent testing company, Product Quality Partners, successful Platinum certification allows you to put the Platinum Logo on your application, a sign to consumers that the program meets Palm Computing's high standards. Platinum certification is not cheap, but it also provides other benefits like marketing support from Palm Computing. An introduction to the program is available at:

```
http://www.palmos.com/dev/platinum
```

Third-party Palm OS Development Sites

The following third-party Web sites offer some helpful resources for Palm OS developers.

Developer.com

This site offers a lot of random resources. There is a great deal of information here, but much of it is hard to find.

```
http://developer.earthweb.com/directories/pages/dir.palm.html
```

Massena.com

Darrin Massena's site for free Palm OS development tools, this page contains links to a number of older tools, as well as source code and a few articles about the guts of the Palm OS.

```
http://www.massena.com/darrin/pilot/tanda.htm
```

Open Palm Group

The Open Palm Group is a group of developers promoting free software development for the Palm OS platform. Aside from being a good way to meet other Palm OS free software developers, this site has an extensive listing of links to free software projects, which can provide you with an excellent source of sample code from which to work.

```
http://www.openhandheld.org
```

Palm OS Development Resources

This site is an excellent place to look for anyone developing Palm OS software on a Unix or GNU/Linux system.

```
http://homepages.enterprise.net/jmarshall/palmos
```

Wade's Pilot Programming FAQ

Though a little dated, Wade's FAQ is still an excellent source of information.

```
http://www.wademan.com/Pilot/Program/FAQ.htm
```

Third-party Hardware Sites

Many third-party Palm OS hardware developers offer Web sites with resources for developing software for the nonstandard features included in some third-party devices.

Handspring Visor

Handspring provides a Web site for developers who wish to take advantage of the special features of Springboard modules in the Visor line of handhelds.

```
http://www.handspring.com/developers/index.asp
```

Kyocera pdQ smartphone

The pdQ smartphone, formerly owned by Qualcomm but more recently bought by Kyocera, includes a number of telephony features not present in a standard Palm OS handheld. Kyocera's developer Web site provides tools and information for developers to take advantage of the pdQ's extra features.

```
http://www.kyocera-wireless.com/devzone/index.html
```

Sony

If you are interested in developing for the unique features of the Sony handheld, including Jog Dial and Memory Stick technologies, take a look at Sony's handheld developer Web site.

```
http://www.us.sonypdadev.com
```

Symbol Palm Terminal

The Symbol Palm Terminal line adds barcode scanning and wireless LAN capabilities to the Palm Computing platform, requiring specialized developer tools and documentation.

```
http://www.symbol.com/products/mobile_computers/
mobile_palm_developers_zone.html
```

TRGPro

TRG Products offers special programming resources for developers who want to write applications that use the CompactFlash expansion slot and enhanced sound abilities of the TRGPro Handheld Computer.

```
http://www.trgpro.com/developer/developer_front.html
```

Mailing Lists

There are a number of mailing lists you can subscribe to for discussing Palm OS development with other developers.

Palm Developer Forums

Palm Computing provides a number of developer forums for discussion of Palm OS development issues. These mailing lists are one of the quickest ways to get answers to nagging technical questions. To subscribe to these forums, visit the following URL:

```
http://www.palmos.com/dev/tech/support/forums
```

prc-tools-devel

The prc-tools-devel list offers highly technical discussion about the internal parts of the PRC-Tools. For instructions about how to subscribe, take a look at:

```
http://lists.sourceforge.net/mailman/listinfo/prc-tools-devel
```

Usenet Newsgroups

Though there are no regular Usenet newsgroups for discussion of Palm OS development, there are newsgroups run by other parties.

Developer.com

Earthweb's `Developer.com` runs a few newsgroups that may be of interest to Palm OS developers:

```
earthweb.palmos.general
earthweb.palmos.conduits
earthweb.palmos.devtools
```

You can read these newsgroups from the comfort of your favorite browser by going to the following URL:

```
http://discussions.earthweb.com/handheld
```

Massena.com

There are a number of newsgroups for Palm OS discussion at `massena.com`:

```
news://news.massena.com/pilot.programmer
news://news.massena.com/pilot.programmer.gcc
news://news.massena.com/pilot.programmer.codewarrior
news://news.massena.com/pilot.programmer.pila
news://news.massena.com/pilot.programmer.jump
```

Palm

Palm pipes its developer forum mailing lists to its own news server at `news.palmos.com`. For detailed instructions regarding how to set up your news reader to read the developer forums, take a look at the following page:

```
http://www.palmos.com/dev/tech/support/forums/
```

Tools and Source Code

Sometimes, the easiest way to figure out how to program is to look at someone else's code. PalmGear H.Q., a repository for Palm OS software, has an excellent developer section, full of useful tools and source code that you can pick through. Also, PalmGear H.Q. is a fantastic place to distribute your application once you have finished writing it.

```
http://www.palmgear.com
```

✦ ✦ ✦

APPENDIX C

Developing in Other Environments

The Palm OS is a very popular development platform, and as with any popular platform, developers have created a huge number of tools to support the creation of Palm OS applications. This book concentrates only on the major C-based tools supported by Palm Computing: CodeWarrior for Palm Computing Platform and the GNU PRC-Tools. However, there are many other tools for Palm OS development that range from easy to use forms-based development systems for quickly producing database-linked applications, to Motorola 68000 assembly compilers for the really hardcore hacker. There are even a number of tools that allow you to write and compile Palm OS applications on a Palm OS handheld. This chapter serves as an introduction to the bewildering array of tools and environments available for Palm OS development.

Forms-based Development

C is a very powerful language, but it has a major drawback in that C development can be rather slow. Even with an integrated development environment like CodeWarrior, the process of writing code, compiling, testing, debugging, and recompiling can make development progress at a snail's pace, particularly for larger applications.

If you are impatient (or your boss is), note that forms-based development environments offer a much quicker way to create applications for the Palm OS. Much as in Constructor, you can visually design the forms for your application. Unlike Constructor, you can then directly associate bits of code with each of the user interface elements without switching to a different development tool. If you have ever used Visual Basic, this model of development will be very familiar to you.

One shortcoming of forms-based environments is that they do not support every feature of the Palm OS. Forms-based development is ideal for the most common applications, like data collection and display programs, particularly if you want to integrate a handheld application with a desktop database. However, if you want to make something more unusual with the Palm OS, like graphics-intensive games, you will need to stick with a more traditional development environment.

Compact Applications Solution Language (CASL)

CASL uses a Basic-like language to allow you to build applications for both the Palm OS and Windows. Applications created in CASL require that a run-time module be installed on the handheld. One of the most useful features of CASL is that it can compile the same application to run on both the Palm OS and on Windows. With the conduit support included in the CASL package, you can easily write a pair of functionally identical applications that can share data between the desktop and a handheld.

Among the features included in the CASL integrated development environment are an interactive debugger, source code editor, and drag-and-drop creation of application forms. CASL can also make calls to external C functions for routines that require either more efficient code or features that are not possible from CASL alone. A more advanced version allows output of C code based on a CASL project, which you may then compile into a standalone application that does not require that the CASL run time be installed on the handheld.

On the downside, CASL applications tend to look rather clunky next to normal Palm OS applications, because the programs that CASL generates tend to ignore Palm OS user interface guidelines. It is still possible to make a reasonably intuitive program with CASL, but be aware that its oversized buttons and other somewhat sloppy interface elements can make a program that is somewhat jarring to the average Palm OS handheld user.

CASL is shareware for the Windows operating system, and you can find it on the Web at `http://www.caslsoft.com`.

You can find the demo version of CASL on the CD-ROM that comes with this book.

Pendragon Forms

Created by Pendragon Software, Pendragon Forms allows you to rapidly create a data collection program with tight integration to a Microsoft Access database on the desktop. Using a special application that must be installed on the handheld, Pendragon Forms works very well for collecting information to later store in an Access database. In particular, Pendragon Forms is well suited to collecting survey information.

The user interface in Pendragon Forms is clean and readable, but it is rather limited. Pendragon Forms restricts you to building forms on its terms; the most customization you can provide is in choosing which fields to include in an application. Also, Pendragon Forms does not create actual applications but rather individual forms that must be run from within the Pendragon Forms application on the Palm OS. In effect, the program allows you to extend data entry forms from Access onto a handheld device. If all you need is a portable data collection device, Pendragon Forms will allow you to get a solution up and running very quickly.

The main reason Pendragon Forms works so well with Access is that Pendragon Forms actually runs within Access. The full version of Pendragon Forms includes the Access run-time module, so you can run it without owning the full version of Access, but to use the trial version, you need to have Access 97 or Access 2000 installed. You can find Pendragon Forms on the Web at `http://www.webfayre.com`.

The 14-day evaluation version of Pendragon Forms is on the CD-ROM attached to this book.

Satellite Forms

The Satellite Forms package, by PUMATECH (formerly Puma Technologies), is a versatile environment for rapidly creating Palm OS applications. Although Satellite Forms applications are not free-standing (they require that the Satellite Forms Engine be installed on the handheld), you have a lot of control over how the programs look. Most forms you can create in a normal Palm OS application are possible in Satellite Forms. It is quite possible to make a Satellite Forms application that adheres well to the Palm Computing's user interface guidelines.

Little or no knowledge of programming is necessary to create a working Satellite Forms application. After visually designing an application in the Satellite Forms App Designer, you can add actions to the user interface elements within a form with a few clicks of the mouse. If the large number of default actions available is not sufficient to your needs, you can set buttons to execute scripts, which are written in a Basic-like script language.

Behind the scenes in a Satellite Forms application are one or more tables, which store record information within the application. Each table is similar to a record database on the Palm OS, and you can connect different user interface elements directly to table fields, so that editing a value in a form automatically changes the value in the underlying table. The Satellite Forms conduit synchronizes these tables with the desktop.

If you need to build a simple database application, Satellite Forms can have a working application up and running in only a few hours. Where Satellite Forms is less useful is if you have special user interface requirements for an application. You cannot

edit menus, and regular Palm OS events are not available. Still, with an extension mechanism that allows you to make custom controls and libraries in C, Satellite Forms can serve to create a wide variety of applications.

Satellite Forms is commercial software, but a trial version is available to let you try it out for free. You can find Satellite Forms on the Web at `http://www.pumatech.com`.

The trial version of Satellite Forms is also included on this book's CD-ROM.

Developing on a Palm OS Handheld

A number of development environments allow you to write and compile applications right on a Palm OS handheld, without having to use a desktop computer at all. Most of these tools require that some sort of runtime component be installed on the handheld, but some actually allow you to compile real, standalone Palm OS applications. These development environments are perfect for creating simple applications, testing algorithms, or proving that you're geekier than your fellow developers.

HotPaw Basic

With HotPaw Basic (formerly called cBasPad Pro), you can write Basic applications from the comfort of your Palm OS handheld, and HotPaw Basic will run them straight from Memo Pad. HotPaw Basic supports more than 95 percent of the ISO/ANSI Minimal Basic language standard, along with many additions for the Palm OS, such as support for serial and IR communication, creation of To Do List and Date Book records, alarm management, and color drawing under Palm OS 3.5.

HotPaw Basic is a shareware application; you can find HotPaw Basic on the Web at `http://www.hotpaw.com/rhn/hotpaw`.

You can also find a free demo of HotPaw Basic on the CD-ROM that accompanies this book.

LispMe

LispMe is a Scheme interpreter that runs under the Palm OS. Distributed as free software under the GNU GPL, LispMe supports color drawing on Palm OS 3.5 and sports its own internal editor that breaks the 4KB limit of the Memo Pad application. Fred Bayer, the author of LispMe, also provides Parentheses Hack, a HackMaster extension that makes programming in a parentheses-laden language like Scheme much less likely to put you in a mental institution. You can find LispMe on the Web at `http://www.lispme.de/lispme/index.html`.

LispMe is also available on the CD-ROM that comes with this book.

OnBoard C

OnBoard C, by IndiVideo, is a C compiler that builds standard Palm OS applications or HackMaster hacks, right on the handheld. You can write your code in either Memo Pad or Doc format, and OnBoard C will compile it into a standalone application. OnBoard C will even generate a skeleton project in either Memo Pad or Doc format. Writing code in Doc format requires an editor that can handle Doc, such as QED, SmartDoc, or pedit.

IndiVideo also makes RsrcEdit, a Palm OS resource editor that also runs on the handheld. OnBoard C is a free application, but RsrcEdit is distributed as shareware. You can find OnBoard C on the Web at `http://www.individeo.net/OnBoardC.html`, and RsrcEdit at `http://www.individeo.net/RsrcEdit.html`.

PocketC

PocketC lets you use a variation on standard C to write applications. In order to run, a PocketC application requires the PocketC run time, which is available for free to both developers and end users. Most of the features you would expect to see in a Palm OS application are accessible to PocketC applications, which may be written in and run from Memo Pad. PocketC is quite popular, and many applications have been written using this development environment.

If you want to write longer, more complex applications, and your eyesight is starting to suffer from squinting at a tiny Palm display, note that PocketC also has a desktop version, which lets you perform all your code entry from the comfort of Windows. Both the Palm OS and desktop compilers allow you to save an applet in its own `.prc` file, though the applet still requires that the PocketC run time be installed. You can find PocketC, by OrbWorks, on the Web at `http://www.orbworks.com`. PocketC is shareware.

Demo copies of both the Palm OS and Windows versions of the PocketC compiler are available on the CD-ROM attached to this book.

Quartus Forth

Quartus Forth is easily the most full-featured onboard Palm OS compiler, turning ISO/ANSI standard Forth code into very tight and efficient free-standing Palm OS applications. Most of the important parts of the Palm OS are available to a Quartus Forth application, and the native-code compiler allows you to create right on the

handheld applications that you can later distribute to other Palm OS users. The free evaluation version cannot compile standalone applications, but it can still run Forth code using its own run time. Like OnBoard C, Quartus Forth can read source code from Memo Pad records or from Doc format. You can find Quartus Forth on the Web at `http://www.quartus.net`.

The evaluation version of Quartus Forth is included on this book's CD-ROM.

Languages Other Than C

A few development environments for the desktop allow you to write Palm OS applications in a language other than C.

Alternative Software Development Kit (ASDK)

The ASDK was the first collaborative attempt to create free development tools for building Palm OS software, and the tools contained in the ASDK form the backbone upon which later free Palm OS development tools are based. Formerly maintained by Darrin Massena, the ASDK features the following tools:

- Pila, a Motorola 68000 assembler
- PilRC, a Palm OS resource compiler that is still used to generate resources for GNU PRC-Tools projects
- Copilot, the predecessor to the Palm OS Emulator
- PilDis, which disassembles PRC code resources into 68000 assembly language instructions
- Many other useful small utilities

Although the main focus of Palm OS development in the free software community has moved away from assembly language programming with the introduction of the easier-to-use GNU PRC-Tools, the tools in the ASDK may still be useful if you like the speed (and pain) of assembly programming. These tools also include source code that can offer some insight into the more obscure inner workings of the Palm OS. The ASDK is available for both Windows and Unix platforms, and it is distributed as free software under GNU Public License.

Unfortunately, because the free software community has moved on to other tools, the ASDK can be a little difficult to obtain. As of this writing, the primary Web site for the ASDK has not been updated in three years, so you may find that a lot of the links from the site are broken. Check the official ASDK site at `http://www.massena.com/darrin/pilot/asdk/asdknews.htm`, or try the *Pilot Development Tools and Articles* page at `http://www.massena.com/darrin/pilot/tanda.htm`.

Jump

Written by Greg Hewgill, the same developer who created the Copilot application that later became the Palm OS Emulator, Jump reads Java `.class` files and generates Motorola 68000 assembly code, which you can then compile into a Palm OS application using Pila from the ASDK. It may sound like an ugly hack, but Jump actually produces very stable applications, though they tend to be a bit larger than programs compiled using CodeWarrior or the PRC-Tools. (I am still rather fond of Jump, because it is the tool I used to make my very first Palm OS application.) Jump runs on Windows, and it is free software under the GNU Public License.

You can find Jump on the Web at `http://www.hewgill.com/pilot/jump/index.html`.

NS Basic/Palm

NS Basic/Palm allows you to write standard Palm OS applications using the Basic language. Using the NS Basic/Palm integrated development environment (IDE), you can create form resources in a graphical setting. Developers have favorably compared the experience of developing applications in NS Basic/Palm to developing Windows applications using Visual Basic.

The NS Basic/Palm development environment is commercial software, which you may purchase from the NS Basic Web site at `http://www.nsbasic.com/palm`.

Pocket Smalltalk

Pocket Smalltalk allows you to create Palm OS applications using the Smalltalk language. The Pocket Smalltalk system consists of an integrated development environment for Windows, which "compiles" a `.prc` file that bundles the source code with a Smalltalk virtual machine, in effect creating a standalone program for the Palm OS. As of the version 1.5 beta, there is still a lot of "fat" in a Pocket Smalltalk program; the pre-release version adds debugging information, and much of the virtual machine code still needs to be optimized. However, future versions promise to be much smaller, requiring only 25KB for a complete Smalltalk interpreter.

Pocket Smalltalk is an open source project; for more information about Pocket Smalltalk, visit the Web site at `http://www.pocketsmalltalk.com`.

Waba

Waba, a language very similar to Java, allows you to make programs that run under the Waba Virtual Machine (WabaVM), which must be installed on a Palm OS handheld for Waba applications to run. The makers of Waba, Wabasoft, provide an entire SDK for Waba development, which can be performed using standard Java development tools.

Although the WabaVM does not support the standard Java Class Libraries, you can still run Waba code as a regular Java application or applet. This kind of cross-platform development offers some interesting possibilities, such as an application that runs with an identical interface on both the desktop and a Palm OS handheld. You can find more information about Waba on the Web at `http://www.wabasoft.com`. Waba is free software under the GNU General Public License.

You can find the Waba SDK on the CD-ROM that accompanies this book.

APPENDIX D

What's on the CD-ROM?

The CD-ROM attached to this book contains a variety of useful tools, sample code, and documentation to help you develop applications for the Palm OS. Many of the tools included are programs that I use for my own Palm OS development work, and together, they form a formidable toolkit for Palm OS programming.

Besides PRC-Tools and various utilities that go with them, I have attempted to collect as wide a variety of alternative development tools as I can find. My hope is that the CD-ROM will serve not only as a toolkit but also as an opportunity to sample different development environments until you find one that suits your needs as a Palm OS developer.

A Word About Shareware

Many of the programs included on the CD-ROM are shareware. The shareware software distribution model allows you to install a program from a shareware collection CD-ROM, or download the program from the Internet, and use the application on a "try before you buy" basis. If you find the application useful, you are obligated to pay a registration fee to the author of the program.

There is a common misconception that shareware is inferior to commercial software. On the contrary, many shareware development tools are written by individuals or small companies who are highly motivated to provide quality programs and support for the development community. Quite often, shareware applications have better support than their commercial equivalents. Also, most shareware tends to be less expensive than commercial software, because the overhead spent on of fancy packaging and marketing is not required for an application that may be downloaded from somebody's Web site.

Shareware is not free, however. Somebody put a lot of hard work into creating a shareware application, and if you choose to use that program, its author deserves compensation for the effort required to write a good piece of software. The shareware applications included on this CD-ROM are not registered; as much as I appreciate your purchasing this book, buying the book does not register these applications. Each shareware application on the CD-ROM includes clear instructions about how to go about registering that program with its author. If you find that you cannot live without one of the shareware applications on this disk, I heartily recommend that you register it to encourage its author to continue developing and supporting useful software.

A Word About Free Software

Many of the tools on the CD-ROM are distributed as free software under the GNU General Public License (GNU GPL), which is included on the disk in the file `GNU.txt`. The GNU General Public License was developed by the Free Software Foundation as a means to promote the development and distribution of free software.

In this context, "free software" does not merely refer to software that you do not have to pay any money to use. Software that is merely given away may be thought of as "free, as in beer," whereas free software under the GNU GPL should be thought of as "free, as in speech." Free software is a concept that embodies what the user of the software is permitted to do with it. Specifically, the user of a free software program should have the following rights:

- Freedom to run the program, for any purpose
- Freedom to study how the program works, and adapt it to one's own needs
- Freedom to redistribute copies of the software
- Freedom to improve the program and release those improvements to the public, so that the entire community benefits

The GNU GPL protects these rights through the concept of *copyleft*, which is a rule governing distribution of free software. Copyleft states that anyone who distributes free software, with or without changes, must pass along the freedom to copy and change the software. In this way, nobody can download a copy of the source for a free software project and convert it to proprietary software.

Because studying how software works and improving that software both require the software's source code, access to source code is a necessary condition for free software. Access to the source also means that many programmers can contribute to improving a program, adding their own improvements and releasing them to the public, comfortable in the knowledge that nobody will be able to steal their code for use in a proprietary application. As the GNU C++ compiler and GNU/Linux

operating system projects have amply demonstrated, this style of "distributed development" allows for creation of very feature-rich and stable software.

Recently, there has been an "open source software" movement, based on the very successful model of allowing many programmers access to an application's source code, so they can all offer improvements to the program. As with free software, many very good pieces of software have come out of the open source movement. Unfortunately, the licensing schemes used by a number of open source projects place restrictions on the source code that interfere with the freedoms allowed by copylefted free software. The people at the Free Software Foundation prefer the term "free software" to "open source software", because the "free" in "free software" implies the freedoms and liberties associated with software protected by the GNU GPL.

The entire GNU PRC-Tools development toolkit, as well as other very useful tools included on the CD-ROM, is distributed under the GNU GPL. These programs are the work of dozens of talented programmers, working together to produce some of the finest Palm OS development software available. In many cases, the GNU tools are easier to use and more powerful than their commercial counterparts. Best of all, these tools are free software, so you can get under the hood and see what makes them tick, or modify them to suit your own development needs.

The Contents of the CD-ROM

Here are some highlights of what you will find on the CD-ROM:

- **Electronic version of *Palm OS Programming Bible*.** The complete (and searchable) text of this book in Adobe's Portable Document Format (PDF), readable with the Adobe Acrobat Reader (also included).
- **Sample applications and source code.** Examples from the text of this book, as well as several complete sample applications and their source code.
- **SDKs.** Software development kits for third-party Palm OS devices.
- **Development Tools.** Many different tools for developing Palm OS software, including complete development systems and helpful utilities.

Running the CD-ROM

The CD-ROM contains material for two different operating systems; at the root level of the CD-ROM are two folders: `/unix` and `\windows`. Each of these folders contains all the operating system–specific material for the appropriate OS, including the sample code from this book formatted properly for each operating system.

Sample applications and source code

Within both of the `/unix` and `\windows` directories is a `samples` directory. In the `samples` directory you will find sample applications and source code from *Palm OS Programming Bible*. Samples and sources are arranged in subdirectories of `samples` by chapter; for example, the directory `\windows\samples\ch11` contains samples from Chapter 11, formatted for Windows. Librarian has its own directory, `samples\librarian`, as does Librarian's conduit application, located in `samples\libconduit`.

The CD-ROM programs

Many of the applications and tools in this section are described elsewhere in this book, but the list that follows contains brief descriptions of each of them, along with where you may find them on the CD-ROM. All of these programs are also available on the Web, and URLs for the appropriate Web sites are also listed. Because development tools tend to evolve much faster than I or the fine people at IDG Books Worldwide can cram them onto a CD-ROM, you should check the Web sites listed in this section for newer, improved versions of these tools.

Acrobat Reader (Adobe)

The electronic version of this book, as well as most of the SDK documentation from Palm and third-party Palm OS developers, is in Adobe Portable Document Format (PDF). Adobe's Acrobat Reader is a free PDF reader, available for Windows, Mac OS, and several flavors of Unix.

Acrobat Reader is in the `/unix/acrobat` or `\windows\acrobat` directories. It is also available on the Web at `http://www.adobe.com/acrobat/`.

Compact Applications Solution Language (CASL) demo

CASL is a cross-platform development system for creating Palm OS and Windows CE applications in the Windows environment. A runtime library installed on the handheld allows an application built by CASL to run the appropriate operating system. This demo version should give you a good idea of what CASL development is like.

CASL is in the `\windows\casl` directory. It is also available on the Web at `http://www.caslsoft.com`.

GNU PRC-Tools 2.0

A complete compiler chain for building Palm OS applications, the GNU PRC-Tools are free software under the GNU General Public License. The PRC-Tools are distributed as binaries for both Windows and GNU/Linux, and source is available for compilation on other Unix flavors. Best of all, the 2.0 version of the PRC-Tools is officially supported by Palm. You will also need PilRC (also included on the CD-ROM) to compile resources.

The PRC-Tools are in the /unix/prctools and \windows\prctools directories. The /unix/prctools directory also contains an src directory, where the source code is kept separate from the RPM binaries in the /unix/prctools directory. The GNU PRC-Tools are also available on the Web at http://www.palmos.com/dev/tech/tools/.

HotPaw Basic demo

Formerly cbasPad Pro, HotPaw Basic allows you to write and execute small Basic programs, directly on the handheld. HotPaw Basic is quite versatile, including features like forms creation and access to several popular handheld database formats, such as JFile Pro and HandDBase. The demo version limits you to running up to four programs. HotPaw Basic requires MathLib (also included on the CD-ROM) in order to run.

The HotPaw Basic demo is in the /unix/hotpaw and \windows\hotpaw directories. It is also available on the Web at http://www.hotpaw.com/rhn/hotpaw/.

Kyocera pdQ Software Developer's Kit

The Kyocera pdQ (formerly Qualcomm pdQ) SDK provides documentation and support for programming the special phone-related features of the pdQ smartphone. The SDK and sample applications are available in both zipped and StuffIt formats.

The Kyocera pdQ SDK is in the \windows\pdq directory. It is also available on the Web at http://www.kyocera-wireless.com/pdq/devzone.html.

LispME (Fred Bayer Informatics)

LispME is an onboard Scheme interpreter that runs entirely on the handheld. The author, Fred Bayer, provides the application and its source as free software under the GNU GPL.

LispME is in the /unix/lispme and \windows\lispme directories. It is also available on the Web at http://www.lispme.de/lispme/index.html.

Pendragon Forms 14-day Evaluation Version

Pendragon Forms allows rapid development of data collection applications for the Palm OS, which can synchronize with Microsoft Access and ODBC data sources on the desktop. This trial version gives you two weeks to evaluate Pendragon Forms on a Windows system with Microsoft Access 97 or later installed.

The Pendragon Forms trial is in \windows\pendragon. It is also available on the Web at http://www.pendragonsoftware.com/forms.html.

PocketC Compiler 3.5 (OrbWorks)

The PocketC system uses a slightly enhanced C syntax to allow you to write applets that run under the PocketC runtime. There are compilers for PocketC that run both

on the handheld (allowing development on the handheld itself) and in a Windows environment (the PocketC Desktop Edition). The runtime module is free for anyone to download and use to run PocketC applets, and the compilers are shareware. Numerous PocketC applets are already in existence, and using them is a popular alternative to regular Palm OS development.

PocketC Compiler for Palm OS is in `/unix/pocketc` and `\windows\pocketc`. The Desktop Edition is in `\windows\pocketc\desktop`. Both versions are also available on the Web at `http://www.orbworks.com`.

Quartus Forth Evaluation Version

Quartus Forth is an onboard ISO/ANSI Standard Forth optimizing native-code compiler for the Palm OS, allowing you to create freestanding applications on the handheld itself. The evaluation version cannot compile stand-alone applications, requiring a runtime library to run, but all of the other features of Quartus Forth are available to try in the evaluation. Quartus Forth has an impressive array of features, some of which are hard to find in a good compiler running on a desktop system.

The Quartus Forth Evaluation Version is in `/unix/qforth` and `\windows\qforth`. It is also available on the Web at `http://www.quartus.net`.

Satellite Forms Trial (Puma Technology)

Satellite Forms is a very easy-to-use forms-based rapid application development system for creating Palm OS programs. The program features easy "drag-and-drop" forms creation on Windows, along with a Visual Basic–like syntax for adding smarts to the forms composing an application. Satellite Forms comes in both Standard and Enterprise versions, offering a good set of features for small or large projects. The trial version allows you to compile and load more than 20 sample applications, which should give you a good idea how well Satellite Forms will work for regular development.

The Satellite Forms trial is in `\windows\satforms`. It is also available on the Web at `http://www.pumatech.com/satforms_fam.html`.

Symbol Palm Terminal SDK (Symbol Technologies)

Symbol offers a software development kit full of documentation and header files for developing applications that take advantage of the special features of the Symbol Palm Terminal 1500 and 1700 lines of handheld computers. There is lots of interesting reading here for anyone who wants to use the added features of the Symbol handhelds, such as barcode scanning and wireless LAN connectivity. Symbol supports development of only these extended features under Metrowerks' CodeWarrior for Palm Computing Platform on Windows.

The Symbol Palm Terminal SDK is in `\windows\symbol`. It is also available on the Web at `http://www.symbol.com/products/mobile_computers/mobile_palm_developers_zone.html`.

TRGPro Development Kit

The TRGPro development kit contains documentation, header files, and sample applications for making use of the extra features available on the TRGPro handheld, including its Compact Flash slot and enhanced sound capabilities. TRG recommends and supports Metrowerks' CodeWarrior for Palm Computing Platform for development of TRGPro-specific features in a Palm OS application, and the sample applications in the development kit are CodeWarrior projects, but the libraries for these added functions are distributed as Palm OS shared libraries, so it should be possible (with a little work) to develop for TRGPro extensions using the GNU PRC-Tools.

The TRGPro Development Kit is in `/unix/trgpro` and `\windows\trgpro`. It is also available on the Web at `http://www.trgpro.com/developer/developer.html`.

Waba SDK (Wabasoft)

The Waba Virtual Machine (WabaVM) is a virtual machine for the Palm OS and Windows CE. Wabasoft provides a complete SDK, including documentation and tools for programming in "Waba," a Java-like programming language. Because of the way the Waba was designed, you can use Java development tools to write Waba applications, which will also run as Java applications or applets. The Java Class Libraries will not run on the WabaVM, though. The WabaVM is an interesting idea because it allows platform-independent development of applications for different handheld operating systems. Best of all, Waba is free software under the GNU GPL, available for Windows, GNU/Linux, and Solaris.

The Waba SDK is in `/unix/waba` and `\windows\waba`. It is also available on the Web at `http://www.wabasoft.com`.

WinZip evaluation version (Nico Mak Computing)

WinZip is a popular archive creation and extraction tool for the Windows operating system, allowing easy manipulation of compressed zip archives. Many of the applications and tools on the CD-ROM are distributed as compressed zip archives. WinZip is shareware; the evaluation version is not crippled in any way, but it is such a mind-bogglingly useful tool that you will probably want to register it, anyhow.

WinZip is in `\windows\winzip`. It is also available on the Web at `http://www.winzip.com`.

✦ ✦ ✦

Glossary

active form Form that receives user input, and where all drawing occurs. There may be only one active form at a time in the Palm OS.

active window Window that receives user input. There may be only one active window at a time in the Palm OS.

alert Modal dialog that presents the user with a short piece of text information.

anchor Location within an HTML document that may be referenced by a hyperlink. For example, an anchor defined by the tag `<a name="here">` may be referenced by the hyperlink tag `<a href="#here">`. Web clipping applications often contain anchors within their pages that allow a user to jump quickly to a specific part of a page without scrolling.

API Application Programming Interface; a set of functions and data structures that give a developer access to certain features of an operating system or program.

app info string list Resource that holds the initial category names for an application.

application info block Structure at the beginning of a Palm OS database that contains category names and other information about the database as a whole, as opposed to information about the database's records.

application launcher Program in the Palm OS ROM that allows users to launch applications installed on a Palm OS handheld.

application preferences See **preferences**.

archive To mark a record as deleted but leave its data intact. An archived record may then be stored on the desktop computer by a conduit during the next HotSync operation.

automatic variable Makefile variable whose value is computed for each rule within a makefile, based on the rule's target and dependencies. For example, `$<` and `$@` are automatic variables that represent a rule's first dependency file and its target file, respectively.

auto-off timer A timer in the Palm OS that, after a certain time has gone by without any user input, causes the system to enter sleep mode to conserve power.

autoshifting A feature of text fields that automatically capitalizes the first letter entered in the field, as well as the first letter after a sentence-ending punctuation mark (., ?, or !).

backlight A common feature on monochrome Palm OS handheld screens that allows the screen to be seen in the dark.

backup conduit A default conduit installed as part of the standard Palm Desktop software that automatically makes desktop backup copies of databases that are not handled by their own conduits.

base ID Base value for generating menu item resource IDs that Rez uses to create menu resources. The first menu item in a menu has a resource ID equal to the base ID, and the resource ID of each menu item after the first increases by one.

baseline Bottom of those characters in a font that do not have descenders. For example, *a* and *z* sit on the baseline, while *p* and *y* dip below the baseline.

beaming Transfer of data or applications between two devices via infrared.

binary search Search algorithm that looks for an item in a sorted array by successively partitioning the array.

big-endian Byte order in which the most significant byte in a multi-byte data type is stored at the lowest address, or "big end first." For example, the four-character sequence `byte` would be stored as `byte` on a big-endian system. Many processor families, including the Motorola 68000 series used in Palm OS devices, use big-endian byte order. The term big-endian derives from Jonathan Swift's *Gulliver's Travels*, in which the Big Endians were a political faction that broke their eggs from the large end ("the primitive way") and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs from the small end. See also **little-endian**.

bitmap A resource type that can store an image for display on the Palm OS screen.

bitmap family A resource type containing multiple bitmap images at different color depths, designed to allow a single image to display properly on handhelds with different screens.

build-prc PRC-Tools tool that assembles code and resources into a Palm OS executable, or `.prc` file.

built-in applications Applications that come pre-installed in a Palm OS handheld's ROM. More specifically, this term usually refers to the four major applications: Date Book, Address Book, To Do List, and Memo Pad.

busy bit A flag which indicates that an application or system process is busy modifying a record. While a record's busy bit is set, other applications and processes cannot open the record.

button User interface object useful for launching frequently used commands and switching between different views in an application.

callback A function passed as an argument to another function, which the second function calls to perform some sort of task. A number of Palm OS routines take pointers to callback functions, which allows a developer to customize how the Palm OS routine works by changing the implementation of the callback.

CALLBACK_EPILOGUE Macro that restores the state of the A4 register after performing a callback function. This macro was necessary in older versions of the PRC-Tools (0.5.0 and earlier), but it is no longer required.

CALLBACK_PROLOGUE Macro that saves the state of the A4 register before performing a callback function. This macro was necessary in older versions of the PRC-Tools (0.5.0 and earlier), but it is no longer required.

card A physical memory card within a Palm OS handheld. Currently, only one card is supported by the Palm OS, but future devices may have more than one card.

catalog resource A resource available from the catalog window in Constructor.

catalog window A window in Constructor offering resources from which to choose. Resources in the catalog window are all user interface objects that may appear within a form.

category A user-defined group of records in a Palm OS application. The Palm OS data manager allows applications to sort records into fifteen categories.

CDK Conduit Development Kit, a set of tools and source code available from Palm Computing for creating conduits.

CGI Common Gateway Interface, a system used by many Web sites to communicate data from an HTML form to a script or program on a server.

character encoding A method of representing text characters as data. Most European languages may be represented using the ASCII character encoding, where only a single byte is required to denote a text character. Many other languages, including most Asian languages, contain more symbols than a byte can hold (a byte is limited to 256 different values), so such languages must use multi-byte character encoding systems.

check box User interface resource that displays a check box and, optionally, a text string. A check box may have a value of `1` if selected (checked), or `0` if unselected (empty).

chunk A contiguous memory area. Chunks may be movable, in which case they are referenced by handles, or fixed, in which case they are referenced by pointers.

Clipper Application on the Palm VII (and other wireless-enabled Palm OS devices) which displays Palm Query Applications and the Web clippings that such applications return over the Palm VII handheld's wireless connection.

clipping An HTML page sent from a server to a handheld over a wireless connection in response to a query from a Palm Query Application.

code island A small function designed to bridge the gap between functions that are more than 32KB away from each other in compiled application code. Also called a **jump island**.

code section Term used in the PRC-Tools documentation to specify a segment. See **segment**.

CodeWarrior IDE and compiler system made by Metrowerks. CodeWarrior for Palm Computing Platform is the version of CodeWarrior designed specifically for Palm OS development, though other versions of CodeWarrior exist to create applications for the Mac OS, Windows, and many other platforms. CodeWarrior for Palm Computing allows Palm OS application development in C or C++.

color depth The number of bits used to represent a pixel's color. Different versions of the Palm OS and different kinds of Palm OS hardware support different color depths, ranging from 1-bit (monochrome) to 8-bit (256 colors).

color table A count of the number of available colors, followed by an array of structures that define each individual color's red, green, and blue values.

command Part of a makefile rule that executes programs or scripts to create the rule's target.

command bar See **menu command toolbar**.

command shortcut A single character assigned to a menu item, which when entered after a Graffiti command stroke executes the associated menu item as if that item had been selected from the menu.

conduit A plug-in module called by the HotSync Manager to synchronize a specific handheld application's data with the desktop.

Conduit Configuration tool Tool that allows a developer to register and unregister conduits with the HotSync Manager.

Conduit Switch tool Command-line tool that allows a developer to save and restore the HotSync Manager's conduit settings.

Conduit Wizard Tool installed in Microsoft Visual C++ by the Conduit Development Kit that generates a skeleton conduit project, based on a developer's specifications.

Constructor Visual resource creation tool that comes with CodeWarrior for Palm Computing Platform.

control One of several user interface resources in the Palm OS. Controls consist of buttons, push buttons, check boxes, repeating buttons, sliders, feedback sliders, pop-up triggers, and selector triggers.

control group A group of mutually exclusive push button or check box controls. Only one member of a control group may be selected at a time.

Copilot Emulator program created by Greg Hewgill for testing Palm OS applications. Palm Computing used Copilot as the basis for **POSE**.

creator ID Four-character code that uniquely identifies a Palm OS application, shared library, or feature. To ensure that each code is unique, developers must register each creator ID with Palm Computing.

custom drawing routine An application-defined callback function that draws individual items within a list, or individual columns within a table.

custom load routine An application-defined callback function assigned to a table column that loads data into that column's text fields.

custom save routine An application-defined callback function assigned to a table column that saves data from that column's text fields.

database A list of memory chunks in a handheld's storage RAM, along with some header information to describe the database itself. Databases may contain either records or resources.

date picker System dialog that presents the user with a calendar, from which the user may select a day, week, or month.

debug ROM See **ROM image**.

default goal The first rule in a makefile, which make will attempt to process by default if you do not specify a target.

definition file A text file containing various properties of a PRC-Tools project, including declarations of different segments that make up an application.

dependency A file or files required by a rule for it to generate its target.

descender The part of a text character that extends below a font's baseline. The tail on a *q* or a *y* is a descender.

dialog A pop-up form that either presents information to the user or allows the user to enter some sort of data; the form may be dismissed when the user taps a button.

digitizer The hardware in a Palm OS handheld's screen that detects pressure from a stylus. The digitizer hands the coordinates where the stylus contacts the screen to the operating system, which translates a user's taps and drags into pen events.

dirty bit A flag that indicates a record was changed and should therefore have its changes saved to the desktop computer (or the handheld, if the dirty record is on the desktop computer) during the next HotSync operation.

DLL Dynamic Link Library, a library of functions that may be called from other Windows applications at run time, without requiring that the functions be statically linked into an application.

doze mode Power mode that a Palm OS handheld spends most of its time in when it is "on." In doze mode, the processor is running but not processing instructions. Doze mode requires much less power than running mode, but is quicker to start up than sleep mode.

DragonBall A family of Motorola processors used in Palm OS handhelds. The Palm OS runs on both the Motorola MC68328 DragonBall and the Motorola MC68EZ328 DragonBall EZ processors.

draw window The window where all drawing occurs. There can be only one draw window at a time in the Palm OS.

DTMF Dual-Tone Modulated Frequency, a method of sound production required to generate tones for dialing a TouchTone telephone. With the exception of the TRGPro, most Palm OS handhelds lack the necessary hardware to generate DTMF tones.

dynamic heap Memory heap where Palm OS applications store their global and temporary variables.

dynamic RAM Area of a Palm OS handheld's RAM devoted to implementing the dynamic heap.

dynamic user interface Method of creating forms and user interface elements at run time, instead of from resources compiled into an application.

edit mode Mode in which the user is editing a text field in a table.

Edit view Screen in Librarian and the built-in Address Book application that allows the user to edit the fields and properties of a single record.

EGCS Experimental GNU Compiler System, the basis for the newest versions of the gcc compiler, a version of which is used in the PRC-Tools.

elliptic-curve encryption Encryption system developed by Certicom and used to maintain a secure connection between a Palm OS handheld and a wireless network.

enter event Event generated when the user touches the stylus to the screen within the bounds of a user interface object.

event A structure used to communicate that something has happened between different parts of the system and an application. An event structure contains information about the type of event (for example, entry of a Graffiti character) and information about that event (for example, the actual character entered).

event handler A system or application function that receives events and responds to them according to their type and what data they contain. An error handler returns `true` if it completely handles an event, or it returns `false` to allow the event to "fall through" to another event handler.

event loop Central part of a Palm OS application that retrieves events from the event queue and dispatches them to the appropriate event handlers.

event queue A first in, first out (FIFO) list of events, which both system and applications may add to. An application's event loop retrieves events from the event queue.

exit event Event generated when the user touches the stylus to the screen within the bounds of a user interface object, drags the stylus outside the object, and then lifts the stylus.

FastSync Style of record synchronization used by a conduit when a handheld is being synchronized with the same desktop computer it was synced with during its previous HotSync operation. A FastSync can reliably use the dirty bit in each record to determine which records need to be processed, ignoring those records that have not been modified.

feature A 32-bit value published by the system or an application to indicate the presence of a particular software or hardware element.

feature creator The unique creator ID of the application that publishes a particular feature.

feature memory A special technique for using features to store small amounts of data that must persist between executions of an application.

feature number An application-defined 16-bit value that distinguishes features that share a creator ID from one another.

feature set A particular set of Palm OS functions and data structures, the presence or absence of which may be determined by checking for a particular feature.

feature table A list of registered features. The Palm OS maintains two features tables: one in ROM for system features, and one in RAM for application-published features.

feedback slider A slider control that returns events continuously while the user holds the stylus down on the control.

FEP Front End Processor, a text-entry method used for some languages (like Japanese) with large, complex character sets.

field A user interface element that displays text and allows the user to edit that text.

file linking An optional conduit feature that, when implemented, allows the conduit to update data on a handheld from a linked desktop file source.

file stream A block of data with no upper limit on its size that an application may read data from and write data to by using the file streaming API.

fill pattern An 8×8 pixel pattern that may be used by certain Palm OS drawing functions.

focus The user interface object that receives all key events has the focus. Text fields that have the focus display a blinking insertion point.

font A particular style for displaying characters on the screen. The Palm OS has several built-in fonts for displaying normal text, bold text, large text, and symbols, and application developers can design their own custom fonts.

form A visual and programmatic container for user interface elements. A given form usually represents a single screen or dialog in an application.

form bitmap A bitmap image that is attached to a specific form.

form layout window A Constructor window that shows a form's layout and allows developers to position and edit user interface objects contained within the form.

fragmentation A situation that occurs when very little contiguous memory is available, because of occupied memory chunks' being scattered throughout a memory heap.

frame The border around a button or window. Window frames are always drawn outside the rectangular region that makes up the window proper.

free software Software distributed with its source code, under a legal agreement ensuring that any applications made using such source code must also make their source code available to other developers. Most free software is distributed under the GNU General Public License, which was written by the Free Software Foundation.

gadget A customizable user interface object that, while providing little behavior of its own, gives a developer a framework for a new object that acts differently from the existing Palm OS user interface elements.

gcc The GNU C/C++ compiler. A version of gcc called m68k-palmos-gcc is included as part of the PRC-Tools.

gdb The GNU debugger, an interactive source-level debugger. A version of gdb called m68k-palmos-gdb is included as part of the PRC-Tools.

global find facility A Palm OS feature that allows the user to search for a string of text in every application on the handheld that supports the find facility.

GNU GNU is Not Unix, a project started by the Free Software Foundation to provide a free Unix-like operating system and developer tools. The PRC-Tools are based on work from the GNU project.

GPL General Public License or, more specifically, the GNU General Public License, a legal license that ensures that developers who base a project on free software must also make their own source code available to other developers.

Graffiti A software system that converts a special shorthand into text, used on Palm OS handhelds to allow text data entry.

Graffiti area Region across the bottom of a Palm OS handheld screen dedicated to receiving Graffiti input. The left side of the Graffiti area is for entering letters, and the right side is for entering numbers.

Graffiti shift indicator A small icon, usually located in the lower right corner of a form, that indicates the current Graffiti shift state, which may be punctuation, symbol, uppercase shift, or uppercase lock.

Gremlin A facility of the Palm OS Emulator that allows the emulator to randomly poke at an application in a reproducible manner, a testing technique that can uncover obscure bugs that might be missed by more structured testing.

Gremlin horde A facility of the Palm OS Emulator that allows a developer to queue up a whole bunch of Gremlins and assault an unsuspecting application en masse.

hack A small utility program that changes basic Palm OS behavior, such as making noise whenever a Graffiti character is entered or displaying the time in the title bar of all applications. See also **HackMaster**.

HackMaster A utility written by Edward Keyes that manages hacks, allowing some amount of safety if two hacks should happen to intercept the same Palm OS system call and making such tricky programming considerably easier for developers.

handle A pointer to a movable chunk of memory. *Handle* is sometimes also used to refer to the memory chunk itself.

hard reset A system reset that clears the contents of both dynamic and storage RAM, usually reserved for recovering from a catastrophic system crash.

hardware button One of the physical buttons on a Palm OS handheld's case, used for turning the handheld on and off, scrolling, or launching applications.

heap A contiguous area of memory that contains and manages smaller units of memory, called chunks.

hierarchy window A Constructor window that shows the hierarchy of objects within a form, useful for selection of objects that are covered by other objects in the form layout window.

HotSync log A text file maintained by the HotSync Manager that conduits use to communicate errors and other useful information to the user.

HotSync Manager Application that runs on the desktop computer and oversees the process of synchronizing a Palm OS handheld with the desktop, calling conduits to perform most of the actual synchronization tasks.

HTML HyperText Markup Language, a method of adding formatting to Web pages so they will display properly in a browser.

HTTP HyperText Transport Protocol, the primary method of transferring data between clients and servers on the World Wide Web.

i icon See **tips icon**.

I/O Input/Output, used to describe the flow of data through a connection between two devices, such as a serial port.

icon A resource holding a small graphic image, used to represent an application in the application launcher.

icon family A resource type containing multiple icons at different color depths, designed to allow a single image to display properly on handhelds with different screens.

IDE Integrated Development Environment, any of a class of programs that combine source code editors, debuggers, compilers, or other development tools into a single interface.

increment arrow A repeating button containing an arrow symbol and lacking a border, usually used to allow scrolling between records in an application.

in-place editing A method of changing text information that allows the text undergoing editing to be stored directly in storage RAM rather than a temporary buffer in dynamic RAM. In-place editing is ideal for large pieces of text, such as Memo Pad records or the Note field attached to many application's records.

insertion point The place in an editable text field where newly entered text appears, represented on-screen by a blinking cursor.

insertion sort A sorting algorithm optimized for sorting an array that is mostly sorted already, useful when only a few array members are out of order and need to be placed in their proper places.

International Feature Set A feature set present in some versions of the Palm OS that provides functions for manipulating text in a localization-friendly manner, as well as facilities for dealing with localized date, time, and number formats.

IR Infrared.

IrDA Infrared Data Association, an industry consortium that creates standards for communicating between devices using infrared beams.

jump island See **code island**.

key event An event generated when the user enters a Graffiti character or presses a hardware button.

label User interface resource that contains static text for display in a form.

large icon The larger of two icon resources an application may have, displayed in the Icon view of the Palm OS application launcher.

launch code A special code sent to an application's **PilotMain** function, requesting an application to start and display its interface, or to perform some small task and exit without showing its interface, such as setting an alarm.

Librarian Sample application that maintains a database of books, used throughout *Palm OS Programming Bible* to demonstrate Palm OS programming techniques.

link converter class Class used in conduits based on the Palm MFC Base Classes to convert records from their handheld format to a desktop format, and vice versa.

list User interface object that displays multiple rows of data in a single column, from which a user may make a selection. Lists come in two varieties: static, which take up screen space on a form, and pop-up, which are displayed only when the user taps a pop-up trigger.

list item A single row within a list.

List view Screen in Librarian and the built-in Address Book application that displays a summary of the records contained in the application's database. From the List view, the user may select a record for closer inspection in the Record view.

little-endian Byte order in which the least significant byte in a multi-byte data type is stored at the lowest address, or "little end first." For example, the four-character sequence `byte` would be stored as `ybet` on a little-endian system. Some processor families, including Intel's CPUs, use little-endian byte order. See also **big-endian**.

LocalID An offset from the beginning of a memory card to a chunk within that memory card.

lock count A count of the number of times a particular memory chunk has been locked. Only when a chunk's lock count is `0` may the system move that chunk to a new location.

logical port number A generic identifier for a kind of port on a Palm OS handheld.

m68k-palmos-gcc C/C++ compiler included in the PRC-Tools. The m68k-palmos-gcc compiler is based on GNU gcc.

m68k-palmos-gdb Source-level debugger included in the PRC-Tools. The m68k-palmos-gdb debugger is based on GNU gdb.

m68k-palmos-obj-res Tool that breaks the single binary file produced by m68k-palmos-gcc into separate code resources that may be included in a `.prc` file. This tool is included in the PRC-Tools.

make A tool that determines which parts of a program need to be compiled or recompiled and then issues commands to perform the necessary compilation. The make tool can avoid recompiling up-to-date source files, instead recompiling only code that depends on changed files and linking the newly compiled object code with object code that is already compiled.

makefile The control file that make uses to figure out how to compile and recompile a program. A makefile consists of rules that determine when a file that makes up a larger project needs to be remade.

masked record A private record that shows up in an application's list of records but whose data is obscured by a gray bar. This method of hiding a private record is available only on Palm OS 3.5 and later.

menu A user interface element that allows a user to launch a command by selecting the command from a pop-up list at the top of the screen. Menus may be opened by tapping the Menu silkscreen button, or on Palm OS 3.5 and later, tapping a form's title bar.

menu command toolbar Bar that appears across the bottom of the screen on Palm OS 3.5 or later when the user enters a command shortcut. A menu command toolbar contains buttons that may be tapped to launch the most commonly accessed menu commands.

menu item A single command listed in a menu.

menu bar Bar across the top of the screen that contains one or more menus. Each form may have exactly one menu bar.

MFC Microsoft Foundation Classes, a set of C++ classes for building Windows applications.

MIC Message Integrity Check, a system that detects tampering and transmission errors in data sent securely from a Palm OS handheld to a wireless network.

MIDI Musical Instrument Digital Interface, a system for encoding both music and musical instrument definitions.

MIME Multipurpose Internet Mail Extensions, a system of classifying and encoding file attachments for e-mail messages.

mirror image synchronization Style of synchronization that allows modification of records on both the desktop and the handheld, keeping both databases up to date with each other and resolving conflicts where a record was modified on both platforms.

modal Describes a form or window that ignores taps outside its borders. Modal dialogs must be dismissed before the user can do anything else in the current application. The Select Font dialog available in most of the built-in applications is a modal dialog.

monitor class Class used in conduits based on the Palm MFC Base Classes to control the synchronization process.

Motorola 68000 Processor architecture used in Palm OS handhelds. Specifically, Palm OS devices contain either Motorola MC68328 DragonBall or Motorola MC68EZ328 DragonBall EZ processors.

multibit icon An icon resource that can contain both monochrome and grayscale icons.

multi-byte character encoding Method of representing characters in a language that uses more than one byte to represent each character. Multi-byte character encoding is essential for many Asian languages that contain more than 256 symbols, the maximum number that may be represented using single-byte encoding.

multigen Tool that generates an assembly language file and linker script necessary for creating multi-segment applications using the PRC-Tools.

multiline field A text field that allows more than one line of text to be entered. Typically, multiline fields may be scrolled. The Edit view in the Memo Pad application is composed chiefly of a multiline field.

multi-segment application An application composed of multiple code resources, or segments. Palm OS programs larger than 64K must be multi-segment applications.

native synchronization logic Logic implemented by the Palm OS base conduit classes to perform mirror image synchronization of records.

no-notification reset A system reset that does not send a `sysAppLaunchCmdSystemReset` launch code to applications to let them know a reset was just performed. No-notification resets are useful if an application crashes upon receiving the `sysAppLaunchCmdSystemReset` launch code, thereby preventing the system from starting properly after a reset.

notification (both Palm OS and HotSync) In the Palm OS, a mechanism similar to launch codes but more efficient that allows applications registered to hear about certain events to receive notice when those events occur. In relation to HotSync operations, notification is how the HotSync Manager tells a desktop application when a HotSync operation is about to start and when it has finished, allowing the desktop application to prevent record modification while the sync is taking place.

notifier DLL A dynamic link library that the HotSync Manager calls to notify a desktop application about the start or end of a HotSync operation.

object ID Another name for **resource ID**. Some Palm documentation also uses object ID to refer to numbers that Constructor assigns internally to resources that it creates.

object index Sequential number assigned to every user interface object contained by a form.

one-directional synchronization Synchronization style where data travels only from the desktop to the handheld, or vice versa.

over-the-air icon Special symbol next to a PQA or Web clipping hyperlink, indicating to the user that following that link will make a wireless connection, thereby incurring airtime charges.

palette The range of colors available for drawing on the display.

Palm Generic Conduit Base Classes Set of C++ classes designed to build conduits that can synchronize a Palm OS application with any arbitrary data source on the desktop.

Palm Image Checker Tool for checking images to make sure they are acceptable for PQA and Web clipping use.

Palm MFC Base Classes Set of C++ classes based on the Microsoft Foundation Classes designed to build conduits that can synchronize a Palm OS application with a data source on the desktop in MFC serialized format.

Palm OS Emulator A handheld emulator that simulates most aspects of an actual Palm OS handheld's hardware and software on the desktop. Also called POSE (and occasionally Poser), the Palm OS Emulator is an invaluable debugging tool.

Palm OS Simulator A Palm OS handheld emulator that runs only on the Mac OS.

PalmRez A post-linker in CodeWarrior that combines linked object code with other resources to form a Palm OS executable, or `.prc` file.

pattern rule A makefile rule that can use wildcard values to specify more than one target or dependency file at a time.

PC ID A pseudo-random number generated by the HotSync Manager to uniquely identify a desktop computer.

PDA Personal Digital Assistant (or Personal Data Assistant), a class of handheld electronic devices designed to serve as organizers, contact managers, and portable computers. Palm OS handhelds are PDAs.

.pdb file Desktop form of a Palm OS record database.

pen event An event generated when the user taps on or drags the stylus across the screen.

Perl Practical Extraction and Report Language (also Pathologically Eclectic Rubbish Lister), a powerful scripting language commonly used for Unix system administration and World Wide Web programming.

PilotMain Entry point function for a normal Palm OS application.

PilRC Tool for creating Palm OS resources for use with the PRC-Tools.

PilrcUI Companion to PilRC that shows a visual approximation of what a PilRC source file's resources will look like once they are on an actual Palm OS handheld.

pop-up list A list resource, normally hidden, that appears in response to a tap on a pop-up trigger.

pop-up trigger A space-saving user interface element that displays the current selection from a hidden pop-up list and, when tapped, displays the list.

port ID A value passed to most New Serial Manager functions to allow them to identify an open serial port.

POSE See **Palm OS Emulator**.

PQA Palm Query Application, the client-side interface of a Web clipping application. A PQA runs under the Clipper application and is similar to a Web page.

.prc file Desktop form of a Palm OS application or resource database.

PRC-Tools Free Palm OS development tools package based on the work of the GNU project, allowing development of Palm OS applications using C or C++.

preferences A database maintained by the Palm OS that stores settings for the system and applications. System preferences hold settings that control how the operating system behaves. Application preferences may be used by any application that needs to store small pieces of data that should persist between invocations of an application but that are too small to merit being stored in a full-fledged record database.

private record A record with its secret bit set, which the system can hide or mask, requiring the user to enter a password to display it.

project The basic unit of application organization in CodeWarrior. A project contains references to the source files, resources, and settings required to build an application.

project resource A resource displayed in Constructor's project window, including but not limited to forms, alerts, icons, bitmaps, and menus.

project stationery A CodeWarrior template that provides the basic skeleton and settings required for a particular kind of application.

project window In CodeWarrior, a window that lists all the files, segments, and targets that make up a project. In Constructor, a window that lists major resources such as forms and alerts, as well as some application-wide resource settings.

protection count A count maintained for a database to prevent the database from being deleted, which allows an application to keep a record or resource in the database locked without leaving the database open. When a database's protection count reaches 0, the database is no longer protected and may be deleted.

public key cryptography A powerful system of data encryption in which a public key is used to encrypt data, which can then be decrypted only by the private key that goes with the public key. Owning the public key does not provide any way to find out the private key, so the public key may be safely distributed via insecure means.

push button User interface object similar to radio buttons on other platforms, usually used in a group to allow selection of one and only one item in the group.

Query Application Builder Tool for creating Palm Query Applications from HTML files, provided free of charge by Palm Computing.

quicksort A sorting algorithm optimized for sorting an array or other data structure that is completely unsorted. A quicksort works by successively partitioning the elements of the array.

RAM Random Access Memory, a type of memory that loses its data when it loses power. Palm OS handhelds store all of their data in RAM, including third-party applications and application data.

record A relocatable memory chunk in storage RAM that holds one piece of an application's data, such as an address book entry or a memo. Records within a database may be scattered across a single memory card, allowing the system to move them around as needed to free up storage memory for other records and resources.

record class Class used in conduits based on the Palm MFC Base Classes to access database records.

record database A database containing records, as opposed to a resource database.

Record view Screen in Librarian and the built-in Address Book that displays a detailed view of an entry's fields and properties.

rectangle Structure defining a rectangular screen region, containing the coordinates of the region's upper left corner and the region's width and height in pixels.

rectangle frame Hollow rectangular screen region that surrounds a particular rectangle. Rectangle frames are always drawn outside the rectangle structure that defines them.

repeat event Event generated repeatedly while the user holds the stylus down within the bounds of a repeating user interface object, such as a repeating button. The scroll arrows in the lower right corner of the built-in applications are repeating buttons that generate repeat events.

repeating button Button-like user interface object that sends repeat events while the user holds the stylus on it. Repeating buttons are often used to create increment arrows.

resource A memory chunk in storage RAM that has a particular type and ID number. Resources are usually used to store user interface elements like forms and buttons.

resource database A database containing resources, as opposed to a record database. A Palm OS application is a resource database containing executable code resources.

resource fork Part of a Mac OS file that contains resources, as opposed to a data fork, which contains data. On Windows, CodeWarrior mimics the data fork of a file with an empty file, and then creates the resource fork in a separate directory.

resource ID Application-defined number that identifies a particular resource within a resource database. Within a resource database, resources that share the same type must have unique resource ID numbers.

resource type Four-character code that identifies what kind of data a resource contains, such as `tBTN` for a button or `code` for executable code.

Rez CodeWarrior tool that generates resources based on text file resource definitions.

RGB Red, Green, Blue, a numeric value that defines a particular color by specifying the amounts of red, green, and blue in the color. RGB values are most often expressed as a six-digit hexadecimal number, where the first pair of digits represents red, the second pair green, and the last two blue. In this scheme, `0x000000` is black, `0xFFFFFF` is white, and `0xFF0000` is bright red, just to name a few of the more than 16 million colors possible using this notation.

ROM Read Only Memory, a type of memory that cannot be written to but that retains its contents when it loses power (as opposed to RAM). The Palm OS system software and built-in applications are stored in ROM.

ROM applications See **built-in applications**.

ROM image A file containing all the data and code packed into the ROM of an actual Palm OS handheld, in a format suitable for use in the Palm OS Emulator. ROM images may either be retrieved from an actual Palm OS handheld or downloaded from Palm Computing. Some ROM images, called debug ROMs, contain special debugging code that generates warnings when an application does something that goes against Palm Computing's coding recommendations. For example, debug ROMs check that you pass valid pointers to user interface objects.

RS-232 Electronics Industries Association standard that defines how the most common type of serial port should operate. The serial port on a Palm OS handheld is compatible with RS-232.

rule Part of a makefile that defines the commands required to create one or more target files, given one or more required dependency files.

running mode Power mode that a Palm OS handheld enters only briefly when actually processing instructions.

schema class Class used in conduits based on the Palm MFC Base Classes to serve as a template for reading records out of a table object.

scroll arrow See **increment arrow**.

scroll bar User interface object that allows the user to scroll text fields and tables, while providing visual feedback about the approximate position of the cursor within a text field or table.

scroll car A dark bar in a scroll bar that indicates the approximate position within a text field or table by its vertical position within the scroll bar and indicates how much of the field or table data is currently visible by its height. Also called a thumb, the scroll car also allows movement through a field or table when the user drags it up and down in the scroll bar.

secret record See **private record**.

segment A single code resource within a Palm OS application. Large, multi-segment applications may contain several segments.

select event Event generated when the user touches the stylus to the screen within the bounds of a user interface object, and then lifts the stylus while still within the object's bounds.

selection In a list, the highlighted list item. In a table, the table item the user most recently tapped. In a control group, the push button or check box that is currently "on."

selector trigger User interface element that contains text surrounded by a gray box, which displays a dialog allowing the user to select a new value to display within the box.

separator bar Special menu item that draws a line between other menu items to group them visually.

serial port A port for transmission of serial data. Palm OS handhelds use a serial port to connect to a desktop computer and other devices that support the RS-232 serial standard.

serial receive buffer Buffer that contains incoming serial data.

serial send buffer Buffer that contains outgoing data queued for serial transmission.

silkscreen button One of the buttons printed to the sides of the Graffiti area on a Palm OS screen. All Palm OS handhelds have Applications, Menu, Calculator, and Find buttons, and Japanese models have additional buttons to control the **FEP**. See also **FEP.**

single-line field A text field that can contain only a single line of unscrollable text.

skin Custom bitmap image that may be applied to the Palm OS Emulator to make it look like different pieces of real Palm OS hardware.

sleep mode Power mode that a Palm OS handheld is in when the unit is "off." Most systems on the handheld are shut down in sleep mode, including the display, digitizer, and processor.

slider User interface element similar to a scroll bar, but arranged horizontally.

SlowSync Style of record synchronization used by a conduit when a handheld is being synchronized with a different desktop computer from the one it was synced with during its most recent HotSync operation. A SlowSync cannot reliably use the dirty bit in each record to determine which records need to be processed, so it compares every desktop record with every handheld record to resolve conflicts between them.

small icon The smaller of two icon resources an application may have, displayed in the List view of the Palm OS application launcher.

SMF Standard MIDI File, a musical data format used by some of the Palm OS sound routines.

soft reset A system reset that clears the contents of dynamic RAM, leaving storage RAM intact.

sort info block Structure at the beginning of a Palm OS database originally intended to hold information about how a database's records should be sorted. Current and previous implementations of the HotSync Manager do not back up the sort info block, so sorting information is best stored in the application info block.

source-level debugging Debugging method that allows a developer to step through a program one line of source code at a time, which makes finding many bugs much simpler than trying to guess why an application crashes or performs incorrectly.

SSL Secure Sockets Layer, a data security layer in common use on the World Wide Web for encrypting data that needs to be transmitted securely between a browser and a server.

storage heap Memory heap containing resources and persistent data.

storage RAM Area of a Palm OS handheld's RAM devoted to storing applications and application data.

string list Resource that contains a number of strings, packed together one after another.

stylus Blunt, pen-like object that comes with Palm OS handhelds, which allows the user to interact with applications by tapping and dragging on the screen and by entering text in the Graffiti area.

switching depth Number of events a particular Gremlin will generate before the Palm OS Emulator switches to another Gremlin in a Gremlin horde.

Sync Manager API Set of functions and data types used for direct communication between a conduit and a Palm OS handheld.

system preferences See **preferences**.

system resource Application resources created by the compiler and linker, such as executable code resources.

system tick Unit of time used by a Palm OS handheld's real time clock. The actual unit is device-dependent, but on most Palm OS handhelds, ticks occur 100 times per second.

table User interface object well suited to allowing the user to edit values directly within the table's columns and rows, capable of containing many other types of user interface elements, such as text fields and check boxes.

table class Class used in conduits based on the Palm MFC Base Classes to store records in a linear format.

table item One cell in a table object.

tap Handheld equivalent of a mouse click, where the user "taps" the stylus on the screen to select a user interface element.

target In CodeWarrior, a particular group of source file and compile settings in a project, designed to produce a specific kind of output. This kind of target is often used for localization, with each country or language having its own target within the same project file. In a makefile, a target is the file or files generated by a particular rule.

TCP/IP Transmission Control Protocol/Internet Protocol, the basic communication language of the Internet.

thumb See **scroll car**.

tick See **system tick**.

time picker System dialog that allows the user to select a time of day.

tips icon Small "i" in a circle, located in the upper right corner of some modal dialogs, that when tapped displays a dialog containing help text for the original dialog.

title bar Region at the top of a form that contains the form's title.

touch To update the timestamp on a source file, usually in an effort to force it to be recompiled. Also, a common Unix tool used to touch files.

transaction-based synchronization Style of synchronization where the desktop computer must perform some sort of processing between each record synchronization. For example, a conduit might retrieve data from a Web site and update a handheld application's database from such information.

UART Universal Asynchronous Receiver and Transmitter, a microprocessor that controls a computer or handheld's interface to its serial devices.

UDP User Datagram Protocol, a simple and lightweight communication language used with Internet Protocol (IP) when a transmission does not require the separation into separate packets performed by Transmission Control Protocol (TCP). The Palm VII and other wireless-enabled Palm OS handhelds use UDP to communicate with a wireless network.

UIAS User Interface Application Shell, part of the Palm OS responsible for managing applications that display a user interface.

usable Descriptive of a form object that allows user interaction and is drawn on the screen. An unusable object is effectively not there from a user's perspective, because it does not appear on the screen and the user cannot do anything with it.

user ID A pseudo–random number generated by the HotSync Manager to uniquely identify a handheld.

virtual key event Key event that does not contain a regular text character, used to represent or trigger system events such as turning on the backlight or raising the antenna on a Palm VII.

Web clipping Process of retrieving data over a wireless connection on properly equipped Palm OS handhelds (such as the Palm VII), which differs from Web browsing in that it focuses on retrieving a simple response from a specific query instead of concentrating on hyperlinks between documents.

window A rectangular region, either on screen or off, that can receive pen events and defines a drawing region. All forms are windows, but not all windows are forms.

window list A last in, first out (LIFO) stack, containing a linked list of windows. The most recently created window is first on the stack.

Wireless Internet Feature Set A feature set present in some versions of the Palm OS that provides functions for manipulating text in a localization-friendly manner, as well as facilities for dealing with localized date, time, and number formats.

WYSIWYG What You See Is What You Get (pronounced “whizzy-wig”), descriptive of an HTML or word processing program that shows you approximately what a finished document will look like while you edit it.

GNU General Public License

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION, AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

End Of Terms And Conditions

IDG Books Worldwide, Inc.
End-User License Agreement

READ THIS. You should carefully read these terms and conditions before opening the software packet(s) included with this book ("Book"). This is a license agreement ("Agreement") between you and IDG Books Worldwide, Inc. ("IDGB"). By opening the accompanying software packet(s), you acknowledge that you have read and accept the following terms and conditions. If you do not agree and do not want to be bound by such terms and conditions, promptly return the Book and the unopened software packet(s) to the place you obtained them for a full refund.

1. **License Grant.** IDGB grants to you (either an individual or entity) a nonexclusive license to use one copy of the enclosed software program(s) (collectively, the "Software") solely for your own personal or business purposes on a single computer (whether a standard computer or a workstation component of a multiuser network). The Software is in use on a computer when it is loaded into temporary memory (RAM) or installed into permanent memory (hard disk, CD-ROM, or other storage device). IDGB reserves all rights not expressly granted herein.
2. **Ownership.** IDGB is the owner of all right, title, and interest, including copyright, in and to the compilation of the Software recorded on the disk(s) or CD-ROM ("Software Media"). Copyright to the individual programs recorded on the Software Media is owned by the author or other authorized copyright owner of each program. Ownership of the Software and all proprietary rights relating thereto remain with IDGB and its licensers.
3. **Restrictions On Use and Transfer.**
   - **(a)** You may only (i) make one copy of the Software for backup or archival purposes, or (ii) transfer the Software to a single hard disk, provided that you keep the original for backup or archival purposes. You may not (i) rent or lease the Software, (ii) copy or reproduce the Software through a LAN or other network system or through any computer subscriber system or bulletin-board system, or (iii) modify, adapt, or create derivative works based on the Software.
   - **(b)** You may not reverse engineer, decompile, or disassemble the Software. You may transfer the Software and user documentation on a permanent basis, provided that the transferee agrees to accept the terms and conditions of this Agreement and you retain no copies. If the Software is an update or has been updated, any transfer must include the most recent update and all prior versions.
4. **Restrictions on Use of Individual Programs.** You must follow the individual requirements and restrictions detailed for each individual program in Appendix D of this Book. These limitations are also contained in the individual

license agreements recorded on the Software Media. These limitations may include a requirement that after using the program for a specified period of time, the user must pay a registration fee or discontinue use. By opening the Software packet(s), you will be agreeing to abide by the licenses and restrictions for these individual programs that are detailed in Appendix D and on the Software Media. None of the material on this Software Media or listed in this Book may ever be redistributed, in original or modified form, for commercial purposes.

5. Limited Warranty.

(a) IDGB warrants that the Software and Software Media are free from defects in materials and workmanship under normal use for a period of sixty (60) days from the date of purchase of this Book. If IDGB receives notification within the warranty period of defects in materials or workmanship, IDGB will replace the defective Software Media.

(b) IDGB AND THE AUTHOR OF THE BOOK DISCLAIM ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE PROGRAMS, THE SOURCE CODE CONTAINED THEREIN, AND/OR THE TECHNIQUES DESCRIBED IN THIS BOOK. IDGB DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR FREE.

(c) This limited warranty gives you specific legal rights, and you may have other rights that vary from jurisdiction to jurisdiction.

6. Remedies.

(a) IDGB's entire liability and your exclusive remedy for defects in materials and workmanship shall be limited to replacement of the Software Media, which may be returned to IDGB with a copy of your receipt at the following address: Software Media Fulfillment Department, Attn.: *Palm OS® Programming Bible*, IDG Books Worldwide, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, or call 1-800-762-2974. Please allow three to four weeks for delivery. This Limited Warranty is void if failure of the Software Media has resulted from accident, abuse, or misapplication. Any replacement Software Media will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

(b) In no event shall IDGB or the author be liable for any damages whatsoever (including without limitation damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising from the use of or inability to use the Book or the Software, even if IDGB has been advised of the possibility of such damages.

(c) Because some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation or exclusion may not apply to you.

7. **U.S. Government Restricted Rights.** Use, duplication, or disclosure of the Software by the U.S. Government is subject to restrictions stated in paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013, and in subparagraphs (a) through (d) of the Commercial Computer—Restricted Rights clause at FAR 52.227-19, and in similar clauses in the NASA FAR supplement, when applicable.

8. **General.** This Agreement constitutes the entire understanding of the parties and revokes and supersedes all prior agreements, oral or written, between them and may not be modified or amended except in a writing signed by both parties hereto that specifically refers to this Agreement. This Agreement shall take precedence over any other documents that may be in conflict herewith. If any one or more provisions contained in this Agreement are held by any court or tribunal to be invalid, illegal, or otherwise unenforceable, each and every other provision shall remain in full force and effect.