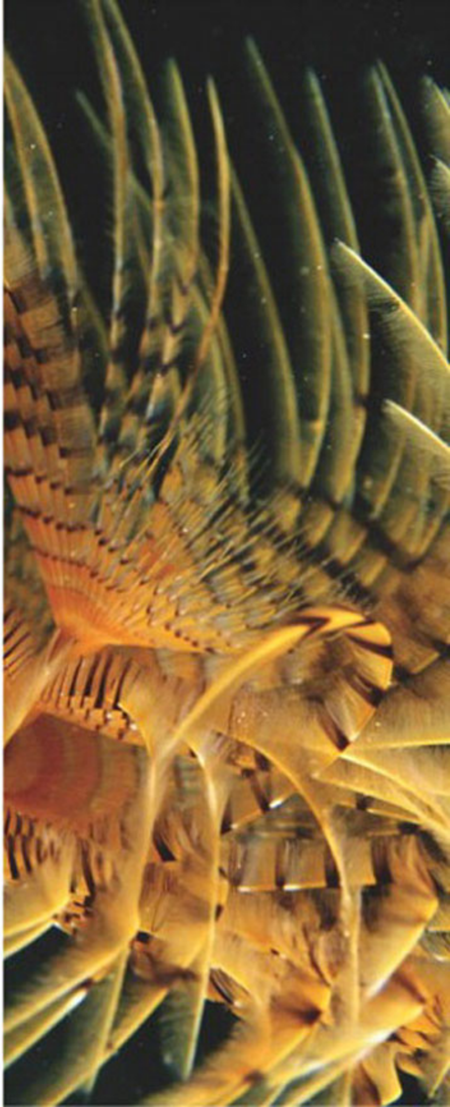
Glenn Bachmann

Palm OS Programming



SAMS

Glenn Bachmann

Palm OS Programming

SAMS

201 West 103rd Street, Indianapolis, Indiana 46290

Palm OS Programming

International Standard Book Number: 0-672-32413-X

Library of Congress Catalog Card Number: 2002100932

Printed in the United States of America

First Printing: October 2002

05 04 03 02 4 3 2 1

Trademarks

Warning and Disclaimer

Associate Publisher
Michael Stephens

Acquisitions Editor
Michelle Newcomb

Development Editor
Mark Renfrow

Managing Editor
Charlotte Clapp

Project Editor
Matthew Purcell

Copy Editor
Kezia Endsley

Indexer
Larry Sweazy

Proofreader
Abby Vanhuss
Suzanne Thomas

Technical Editor
Jeff Pajor

Team Coordinator

Lynne Williams

Interior Designer
Gary Adair

Cover Designer
Alan Clements

Page Layout
Michelle Mitchell

Graphics
Steve Adams
Tammy Graham
Oliver Jackson
Laura Robbins

Contents at a Glance

Table of Contents

About the Author

Lead Author

Glenn Bachmann is president and founder of Bachmann Software (`http://www.bachmannsoftware.com`), a leading provider of software products and professional software development services for the mobile computing and wireless communications industries. Bachmann Software's popular PrintBoy mobile printing software and FilePoint Pro file management suite are available through major retail stores and online software channels, and are in everyday use on Palm OS and Windows CE PDAs worldwide.

Contributing Authors

Frank Ableson is an expert in computer hardware and software communications, network programming and image processing, and he relishes solving problems in general. Frank is a contributing author for chapters on infrared communications, serial communications, and network programming. Franks runs his own consulting firm CFG Solutions, and his contributions to Bachmann Software over the years have earned him the title of resident communications guru. His biggest fans are his wife Nikki, and children, Julia, Tristan, and Natalie.

Greg Winton is the author of *Palm OS Network Programming* (O'Reilly & Associates, 2001) and a regular contributor to several technical magazines. He provides application development consulting for Palm OS, Linux, and a wide variety of handheld and wireless networking platforms. For more information, visit him on the Web at `www.gregsprogrammingworks.com`.

Dedication

This book is dedicated to that brown-eyed girl I fell in love with, my wife Joananne, who against all logic continues to unquestioningly support me through the roller coaster adventure of running a business.

I also dedicate this book to my beautiful children Nicholas, Julianne, and Courtney, who every day make me proud to be their father, and who provide me with first-row seats to the greatest show on earth: watching my kids grow up. I wouldn't miss it for the world.

Acknowledgments

I am grateful to Michelle Newcomb, my editor at Sams Publishing, for having the patience of 1000 saints during the development of this 2nd Edition. Despite my endless tormenting, Michelle kept the faith (and her sense of humor) without losing her sanity, and offered incredible support and understanding during a time when I needed it most. Thank you!

Thanks also to Michael Stephens, Matt Purcell, Mark Renfrow, and the rest of the team at Sams Publishing for all their support and contributions.

I have had the privilege of working closely with some very special people, whom I would like to thank:

Frank Ableson comes through time and again with solutions to just about anything I can throw at him. Frank always has time to listen and has a magical knack for instantly sizing up and grasping the most vexing problems. Thanks again to Frank for his knowledge and contributions to this book.

Pete Angiuoli is the go-to guy, the software god, king of conduits, the sultan of sync, the wizard, master of all handheld programming. Generous beyond words with his time, patience, and understanding, this book would not be possible without Pete's inspiration, for which I am truly grateful.

Thanks to Dan Reuvers, who is one of the most extraordinary people I have ever met. Dan is a true professional, a cherished friend; I am a lucky man indeed to have known him.

Greg Winton is among the most accomplished developers I have known in my career, a rare software designer with vision, and a remarkable human being.

I'd also like to take a moment to thank some friends, partners, and colleagues along the way who have provided me with help, information, feedback, inspiration, and support in this wild and crazy world of handheld computing: David Fedor and his team at Palm developer technical support, without whom I am certain the world of Palm software development would crumble and collapse; Gary Downing and the great group of people at Palm Source for being an extraordinarily generous source of information on Palm OS; Dave Diangson at Handspring; Alexander, Tamara, and Scott at Blue Nomad; Jeff, Mike, and Paul at Cutting Edge Software; Denny Atkin at Handheld Computing; Shawn Barnett at Pen Computing. Thank you for being a part of it all.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email:	`feedback@samspublishing.com`
Mail:	Michael Stephens Associate Publisher Sams Publishing 201 West 103rd Street Indianapolis, IN 46290 USA

Reader Services

For more information about this book or others from Sams Publishing, visit our Web site at `www.samspublishing.com`. Type the ISBN (excluding hyphens) or the title of the book in the Search box to find the book you're looking for.

Introduction

Why Read This Book?

Palm OS Programming is a comprehensive introduction to programming for the Palm operating system, the fastest growing computer operating system and platform ever created, and certainly the most popular mobile computing platform in existence today.

Throughout my career in software, whenever I was confronted with the challenge of quickly and efficiently getting up to speed with a new operating system, tool, development library, or other technology, I ran for the nearest bookstore (these days I head straight for `www.amazon.com`!). I would immediately scan the available titles that covered the subject matter I was interested in, and where there were several, I would naturally gravitate toward the books that presented the material in a large number of bite-sized, easy-to-understand examples that focused on various aspects of the technology.

I'd open one of those books, and right away I could usually spot four or five chapters that I could instantly make use of. I would without fail elect to buy that title over all the others, with the confidence that it would give me the immediate practical know-how I needed, but also have the depth that would allow me access to information as my curiosity drove me to investigate the technology further.

I'll leave it to the reader to decide whether I was successful in creating that kind of book. But there is no doubt in my mind that books of that sort are desperately needed for the vast numbers of software developers who are new to the Palm OS. Palm, Inc., has done an admirable job assembling all of the developer documentation and API listings for the programming community, but those are references, to be opened when you already know the functions you need to use. The problem is that developers new to Palm programming do not already know the right functions to use, nor do they have time to figure it out themselves.

If you have ever wondered how to write Palm applications, or you have already piqued your curiosity by doing some investigation into the world of Palm programming, this book is for you. In this book, you will find

- Tutorials on most aspects of Palm software development, from “Hello Palm” all the way to advanced topics such as communications, expansion memory, and synchronization
- Real, working example code that you can use right away in your own programs
- In-depth treatments of design issues specific to mobile computing, such as choosing a wireless platform, user interface design, and memory management

Who Should Read This Book?

This book is appropriate for software developers, system designers, quality assurance professionals, and technical writers who want to learn about software development for the Palm computing platform. Although an introduction to Palm programming, this book targets those who are at an intermediate level of understanding in software development, and who have some familiarity with tools such as compilers, editors, and debuggers, as well as basic programming concepts such as variables and functions.

The book assumes little or no Palm programming experience, but does assume that the reader has at least a working knowledge of a procedural or object-oriented programming language such as C, C++, or Java.

Some experience with programming on an event-driven graphical operating system such as Windows, Macintosh, or UNIX is helpful.

How to Use This Book

This book was purposely designed to provide a set of standalone, self-contained complete tutorials on various aspects of Palm programming. My goal is to provide practical and easily understandable chapters that enable the readers to quickly and easily get up to speed on any topic of interest.

For beginning Palm programmers, I recommend starting with a good straight-through reading of Chapters 1 through 6, followed by a review of the other chapters on user interface and database programming.

For readers who already have some level of Palm programming experience, a cursory review of the first six chapters is probably all that is necessary. Because virtually every chapter in the book is self-contained, from there you can navigate to any topics you find interesting.

Once beyond the basics, you can pretty much attack the rest of the book in any order you like, or refer back to sections as the need arises.

I'm a firm believer in learning by example, so I highly recommend not just reading the source code listings, but actually building, running, and debugging through the accompanying chapter examples. It's the best way to learn!

How to Contact the Author

I welcome comments and questions from readers. Due to the volume of inquiries, I cannot guarantee an immediate response, but I will do my best to answer any queries in a timely manner. Please address correspondence to `glenn@bachmannsoftware.com`.

Where to Get the Chapter Example Projects

Full source code and buildable project files are downloadable from Sams Publishing's Web site at `http://www.samspublishing.com`.

Additionally, the source examples are available on my company's Web site at `http://www.bachmannsoftware.com/palmprogramming`. Check here often for updates and corrections to the book chapters and examples, as well as new content and other useful information. You'll also be able to subscribe to a mailing list that automatically notifies you of updates and new information related to the book.

PART I

Getting Started with Palm Programming

IN THIS PART

1 Introduction to Palm Programming and CodeWarrior

IN THIS CHAPTER

- The Palm Computing Platform
- What Is a Palm Application?
- How Are Palm Applications Written?
- CodeWarrior and Development Tool Choices
- What Tool Is Best for You?
- Tools Used in This Book
- The Palm SDK

I have enjoyed the privilege of living through three major computer-platform revolutions in my career: the move to personal computers from mainframes and minicomputers, the adoption of graphical user interfaces, and the rise of the Internet. I distinctly recall feeling a sense of wonder as I witnessed the adoption of new technology and the creation of entirely new market opportunities. During each of these periods of transition, new companies began, became successful, and, in some cases, replaced old companies that either weren't paying enough attention to the importance of these new technologies, or couldn't make the transition to the new platform quickly enough.

There is no doubt in my mind that handheld and mobile computing is a fourth wave of change that is rippling through our industry. It is clearly changing the way that we work and live our lives and brings computing power to a whole new user community. Although at one time this technology was expensive, slow, and bulky and its capabilities were ridiculed in the press (including a well-known comic strip), a new breed of handheld devices has captured the imaginations (and shirt pockets!) of consumers and business people worldwide.

Handheld computing devices that run the Palm Operating System, or Palm OS, are the most successful of this new breed by almost any imaginable measurement. To be sure, there are many platform and device choices out there, and the shapes and sizes of mobile computers are proliferating rapidly. There are "pure" PDAs, PDAs with built-in wireless modems, PDAs with built-in cell phones, cell phones with built-in calendars, portable music players with built-in address books, and PDAs with keyboards and others with digital cameras. And among these many device types there are many platform choices, including Microsoft's Windows CE/PocketPC, the Symbian OS endorsed by Nokia and others, and even a portable version of Linux. Yet purely based on the evidence to date, one must conclude that Palm OS-based devices, including those from Palm, Handspring, Sony, and a growing number of other Palm OS licensees, have collectively dominated the market for PDAs.

Consequently, the Palm OS is receiving a lot of attention from software developers looking to create new software applications that provide computing power for the mobile user. Although designed to be useful right out of the box, Palm OS devices have been shown to be tremendously versatile in a dizzying number of uses and applications, with the only true limitation being the collective imagination of the customer and the developer.

The community of software developers creating aftermarket, or "third-party," software applications for Palm OS PDAs has grown from a few early enthusiasts to numbers reportedly in the hundreds of thousands. In a few short years, this aftermarket community has produced thousands of add-on software applications ranging from replacements for the built-in Address Book and Datebook, to word processors and spreadsheets, to email and wireless Internet applications, to games and entertainment, to custom applications targeted at healthcare, education, and other vertical markets. Add-on "shrink-wrapped" Palm software is now a regular sight at retail outlets around the world.

Handheld devices are among the most popular tools available for business and consumers, and not just as replacements for your old paper-based daily planner. Although for sure many PDA customers will purchase a Palm device purely for the "PIM" (Personal Information Manager) aspect, more often than not the PDA customers will purchase their devices as a starting point, and then proceed to augment and personalize them, adding functionality in the form of new add-on software and hardware until the devices are uniquely suited to their own needs.

Thus in terms of how it is viewed by the customer, a PDA is less like a toaster or refrigerator than it is like a house or apartment. Rarely does someone buy a house and live in it "as-is." Rather, they slowly but surely redecorate, add on, rearrange, and customize their house until it is a reflection of their lifestyle. Similarly, it is hard to imagine the personal computer without the existence of a rich, diverse array of add-on software and hardware solutions that address almost every imaginable business or personal need.

It is worth noting that my house analogy still only pays tribute to the portion of PDAs that are purchased for purposes of personal productivity. There is a whole additional market for PDAs that are purchased as part of internal corporate applications, for sales force automation, inspections, transportation and shipping, and other applications. PDAs provide the potential for tremendous cost savings in terms of equipment, connectivity, and just plain productivity, and the enterprise sector has certainly noticed.

Do any (or all) of these things seem like exciting opportunities for a software developer? If so, this book is for you. In many ways, the key to the success of the Palm computing platform (and by extension, mobile computing in general) is found in the creation of software applications that provide real value to individuals and businesses, by offering essential functionality and new ways of communicating, organizing, computing, and conducting business. This book is all about providing software developers with the practical know-how required in order to create those software applications.

In order to get you started with Palm programming, this chapter provides:

- A brief overview of the Palm computing platform and how it came about
- What it's like to program for the Palm in C and C++
- What tools are available for Palm software developers

After Chapter 1, we will immediately get into the meat of how to write Palm applications, so this chapter provides a good foundation to build on for the rest of the book.

The Palm Computing Platform

Although only a few years ago a Palm OS-based device literally meant a Palm-branded device, Palm, Inc. has been very active in licensing the Palm OS to other hardware manufacturers, encouraging the creation of a diverse set of devices that are suited for different markets, as well as growing the installed base beyond where a single device manufacturer might be able to take it.

It is a tribute to Palm's branding efforts that the term "Palm Pilot" has almost become a catch-all term for any PDA, whether it runs the Palm operating system or not, just as the term "Xerox" as come to mean general purpose photocopying, not just on Xerox-branded equipment. But the reality is not far from that: Today most PDA models do in fact run the Palm operating system, and the number of device manufacturers licensing the OS continues to grow. In 1999 the list of licensees was four (Palm, Symbol, IBM, and Qualcomm). Today that list has grown to include Handspring, Sony, Kyocera, Samsung, Franklin Covey, TRG/Handera, and others, with more surely to come.

Although this diverse array of devices includes PDAs that address a very large range of capability, including color, expansion card slots, high-resolution displays, bar code scanners, wireless modems, keyboards, and telephony, a strength of the Palm platform is that there is a common base set of characteristics found in all devices.

A quick survey of the devices supporting the Palm OS yields the following:

The Palm computing platform has five main pieces, including:

- A standard hardware design
- An interface for extending the Palm using add-on hardware components
- The Palm Operating System
- Data synchronization to and from host systems using HotSync
- The Palm SDK and associated tools that enable programmers to develop applications

It is important to note that all the devices listed here run one common operating system: Palm Inc.'s Palm OS. Knowing this, you can reasonably expect that your Palm application will run on a wide variety of hardware devices. Developers such as you and me appreciate that, and as a result, Palm is rapidly attracting new developers to the platform. For programmers who wish to take specific advantage of the features offered by one device or another, the opportunity exists, but there is always a core assurance that a program that uses the base operating system will run across all Palm OS-based devices.

What Is a Palm Application?

To an end user, a Palm application is a file with a .PRC extension. (In Palm parlance, executables are often referred to as .PRC files.) .PRCs are created on the desktop by writing code and creating resources using a development tool (such as Metrowerks CodeWarrior) and are then transferred to the Palm device using HotSync. On the Palm device, a .PRC is actually a special kind of database called a resource database. (Resources and databases are discussed in detail later.) A .PRC contains all the elements of your application, including the code itself and user-interface elements. The Palm operating system knows how to load and execute the code that is contained in that file, and can run it as a program on the device.

Palm applications, like applications in other environments, are "event-driven". This means that your application code interacts with and is notified by Palm OS as significant events occur, such as pushing a button or completing a pen stroke. Almost everything that happens in the system that's of interest to your application will be sent as an event.

Although internally, the Palm OS is capable of multitasking, Palm applications do not multitask: If a user is running your application and he or she decides to look up a friend's phone number, your application will actually exit when the user invokes the address book. Also, note that Palm applications do not have the ability to create multiple threads of execution.

Aside from the program that runs on the Palm device, many applications require the capability to interact with a host computer to exchange data. For this task, Palm developers must create a "conduit." A conduit is not a Palm application at all; it is a special Windows or Macintosh component, developed with standard Windows/Macintosh tools (Microsoft's Visual Studio or Metrowerks CodeWarrior), that is invoked when you perform a normal HotSync. Conduits can provide special database synchronization between data on the Palm and the desktop. They can also be used to transfer files and perform other tasks requiring interaction with the desktop. Conduit development is covered in much more depth later on in this book.

How Are Palm Applications Written?

Most Palm programs today are written in C or C++. Alternatives to C do exist, including Visual Basic and Java, although depending on your requirements, the current processing power of the device may not give your program the best possible performance.

Despite the C++ support, it is very important to realize up-front that many of the more "advanced" features of C++ are not supported by the available Palm development environments. You will find that the essence of C++ (inheritance, polymorphism, and so on) is supported, but beyond that, things get dicey. Among other things, you will not find support for C++ templates or exceptions. What this means is that if you are looking to port code from another platform to the Palm that relies on these features, you will unfortunately need to rewrite some of your code.

NOTE

Although some will debate this (indeed it is all too easy to ignite a healthy discussion on this point), my personal inclination is to "compile for C++, but code for C." By doing this I take advantage of type-checking and other benefits of compiling for C++, but I do not make extensive use of classes, inheritance, or other C++-specific constructs.

Many compilers today offer a built-in set of standard functions called the *C runtime library*. Functions such as `malloc`, `printf`, `memset`, and `strcpy` are old friends that as programmers we have come to rely on, even in new operating system environments. Unfortunately, the current Palm SDK does not natively support the standard runtime library. Instead, it offers its own set of memory and string routines. These routines are not as comprehensive nor as fully functional as their counterparts in the C runtime library, so many developers have resorted to writing their own functions as they need them. Still, in many cases the Palm routines are sufficient on their own.

NOTE

As developers, we often debate the merits of taking extra time to write portable, compiler-independent, and operating-system-independent code. Well, the Palm is an instance where you will profit handsomely if you are careful to write portable code that insulates you from dependencies on the underlying operating system interfaces. When you want to port some code to the Palm, you will also be rewarded for having shown patience and restraint in not jumping at the chance to take advantage of the latest C++ extensions or operating-system-specific frameworks. At my company, we are taking a conservative stance on new compiler and OS features for this very reason.

CodeWarrior and Development Tool Choices

Palm, Inc. has endorsed Metrowerks CodeWarrior development tool as the "tool of choice" for creating Palm OS applications. CodeWarrior is a tool for C and C++ developers, and is roughly similar to Microsoft's Visual Studio in that it is a visual Integrated Development Environment (IDE), providing developers with an all-in-one toolbox containing the compiler, linker, resource editor, project manager, editor, and debugger. CodeWarrior is available for developers who program on Windows and Macintosh computers.

Because of Palm's endorsement of and relationship with Metrowerks, you can also be assured that new Palm OS versions, SDK changes, and other enhancements will be quickly supported, usually in advance of other development tool options.

However, like Visual Studio, CodeWarrior is not free—developers must purchase a copy of CodeWarrior from Metrowerks. If you do purchase CodeWarrior, you will find that when combined with the Palm SDK it contains everything you could need to develop for the Palm platform. My recommendation is if you are a C or C++ developer and are ready to make a commitment to creating serious Palm applications, you would do best to purchase CodeWarrior.

If you are a hobbyist and programming for fun/pleasure or aren't quite ready to take a professional plunge into Palm programming, there are alternatives for you.

PRC-Tools

PRC-Tools (also referred to as the "GCC" toolset) is a suite of free tools available for Windows and Unix developers. PRC-Tools is in fairly wide usage in the development community, although it is hard to say how many commercial software applications have been written using PRC-Tools. PRC-Tools comes with a compiler, linker, debugger, and resource editor. It does not come with a visual IDE—the tools run from a command prompt, and require the use of scripts such as `make` to tie them together into a cohesive build environment.

For those of you who are used to the creature comforts provided by an IDE, I should warn you that PRC-Tools is not for the faint of heart; it does require some under-the-hood fiddling to get things working, and it does require you to provide your own editor and script your own `make` tools, both of which come as part of the Metrowerks package.

Aside from the fact that it is free, there are other rewards for those who take the plunge and use PRC-Tools. For one, the resource editor is superior and more flexible than the Constructor tool that comes with CodeWarrior. Another strong advantage is the capability to perform command-line builds using the standard `make` tools.

Falch.net

Falch.net is a nice, easy-to-use tool that layers an IDE on top of PRC-Tools, providing both a visual development environment as well as the makefile compatibility of PRC-Tools. Falch.net does a good job of tying together all the pieces required for a complete Palm development environment, and is a very attractive alternative to both CodeWarrior and PRC-Tools. It isn't free, because the IDE, editor, and other features add significant value.

AppForge

AppForge offers Visual Basic developers a tool for creating Palm applications that integrates with Microsoft's Visual Basic environment. For those who may not be familiar with C or C++, this is a good choice.

NS Basic

NS Basic is another nice choice for Visual Basic developers, providing an easy to use IDE with good operating system support.

Satellite Forms

Satellite Forms, from Puma Technologies, is a comprehensive suite of tools that is "forms-based," allowing you to visually design the screens for your applications, associate form elements with database fields, and attach script commands to form events in a Visual Basic-like manner. If your application is heavily forms and database driven, Satellite Forms represents an excellent choice. Satellite Forms also adds significant value in that conduit development, normally a bit of a black art, is much easier and simpler.

What Tool Is Best for You?

All the tools mentioned in the last section represent good options for Palm development that I am familiar with. Although I do not personally or professionally endorse any of them in particular, you might use the following simple guidelines as a rule of thumb (keeping in mind that ultimately you can create a reasonable Palm application using any of the tools):

- If you are comfortable with C and C++, and place value on using the officially support toolset from Palm, go with CodeWarrior, no question.
- If you are a C or C++ developer and interested in alternative tools, check out PRC-Tools and/or Falch.net.
- If your background is in Visual Basic, check out NS-Basic or AppForge.
- If your application is highly database and form driven, with a need to exchange data with a desktop computer or server, investigate Satellite Forms.

I'll add that by no means are these the only choices available to you, they are just in my opinion the most "mainstream" choices. Certainly there are many, many ways to create solutions for the Palm platform, including the use of databases, spreadsheets, Web, email, and other possibilities.

Note that I do not differentiate based on cost in my guidelines. Because you've purchased this book (thank you again!), I am assuming you are making a commitment of some level to becoming a Palm programmer. In my experience, unless you have no budget at all, it is short sighted to choose a development tool on the basis of cost. Spending $10, $50, $200, or even $500 will ultimately seem like small change compared to any additional hours/development time associated with a poor development tool. If you save $500 on a development tool, but then spent weeks trying to make it fit your problem, you must wonder whether you saved yourself any money in the end.

One last thing to keep in mind: The further away from pure C/C++ development and Codewarrior you go with your choice of tool, the more likely that it will take longer for your chosen tool to support the latest and greatest devices, OS enhancements, and SDK changes. You become dependent on your tool vendor to stay current. Check out how the various tool vendors are with their current support of the Palm SDK, and compare that with CodeWarrior as the "gold standard." Although your application might not be dependent on having support for the very latest enhancements, it's something to consider.

Tools Used in This Book

Beyond this chapter, the rest of the book assumes you are comfortable with C and C++, and are using CodeWarrior, PRC-Tools, or Falch.net. The sample code provided with each chapter can be compiled in all three environments, and the project and resource files compatible with each are provided through the Web site.

In instances where specific tools are relevant to the discussion (for example, Chapter 5, "Creating and Managing Resources"), I will take pains to describe how to use each environment. For the remainder of the chapters, you can assume that the concepts should apply to all three environments.

The Palm SDK

If you purchase CodeWarrior, you get the Palm OS Software Development Kit (SDK) for free on the CodeWarrior CD. For GCC or Falch.net users, or for those who want to update their CodeWarrior installations, the latest SDK is downloadable from the Palm Source Web site.

The SDK contains headers, libraries, tools, a tutorial, sample code, and documentation on how to use the Palm SDK application programming interfaces (APIs) in your Palm programming. It also contains the source code for the built-in applications such as address books. Be aware that much of the documentation is in the form of reference material and function listings. Aside from the tutorial, most of the material answers the "what," but not much material is devoted to answering the questions "how" and "why."

The Palm SDK documentation is logically divided into interface management, system management, and memory and communication management.

- "Interface Management" provides the closest thing to an overview and then provides a reference for the available user-interface resource types as well as functions used in managing your application's user interface. I refer to this manual the most.
- "Memory and Communication Management" covers a wide territory, including how to allocate and manage memory chunks, the Palm database interface, serial and infrared communications subsystems, and the networking APIs.
- "System Management" is kind of a catch-all, covering error handling, alarms, events, system and application preferences, string management, and other miscellaneous areas.

I highly recommend taking the time to walk through the tutorial. It is also a great idea to explore the source code for the built-in applications as well as the other examples. With an SDK as large as Palm's, there is nothing like looking at other people's code to gain an understanding of how to properly use the APIs.

Summary

This chapter provided an overview of the Palm device, the Palm computing platform, Palm programs, and the tools and languages used to develop Palm applications. One thing noticeably absent from this first chapter is source code. Don’t panic! This is virtually the only chapter in the book that doesn’t show you specifically how to program some aspect of the Palm. Let’s get to it: Welcome to *Palm OS Programming*!

2

Anatomy of a Palm Application

IN THIS CHAPTER

This chapter takes apart and examines a Palm version of the old "Hello World" program (I call it "Hello Palm"), the classic first program many people create when confronted with a new programming environment or development platform.

The "Hello Palm" program not only illustrates the basic structure of a Palm application, but it also serves as a project template for practically every other sample program in this book.

As you'll see when you walk through setting up a new CodeWarrior or GCC Palm project, getting all the settings just right can be tedious: What I've found to be easiest is to simply clone "Hello Palm" and change just a few settings rather than start from scratch.

PilotMain and the Smallest Palm Program Ever

I remember the first time I was confronted with a "Hello World" program written for Microsoft Windows on a 286. (Yes, I know that means I'm old....) Having seen my share of simple DOS programs that consisted of four or five lines of code in a single `main()` function, it was a shock to realize that I needed several hundred lines of code dealing with message loops, window procedures, and class registration just to get a window to display the text "Hello." It eventually became clear to me that for the most part, this code never changed; I simply copied and pasted the same boilerplate code from project to project.

Programming the Palm is similar. With the way I've structured "Hello Palm," you can learn about 300 lines of boilerplate code (including comments) in this chapter and then forget it. I would be remiss if I did not explain all that strange-looking code in every Palm program, but you should keep in mind that with only a few exceptions, it isn't necessary to revisit it.

Before you dive into the full "Hello Palm" program, take a quick look at the smallest Palm program ever. It does nothing, and aside from allowing you to try some functions in the debugger, I can't imagine what you would ever do with such an application. But it does give you a place to start:

```
/*
   PilotMain ()
   Main entry point into pilot application.
   Parms:   wCmd  - launch code.
            pInfo - struct associated w/ launch code.
            fuLaunch - extra info about launch.
*/
UInt32  PilotMain (UInt16 wCmd, MemPtr pInfo, UInt16 fuLaunch){
   return 0;
}
```

This is a very boring program because it ends as quickly as it starts—no user interface, nothing. However, it's a valid Palm program; you can build it and download it to your Palm.

Just as a standard DOS or UNIX program begins with `main()`, a Palm program begins with `PilotMain`. `PilotMain` is called when your program is launched by the operating system. Your application can be launched normally or in special modes using launch codes. This code is passed as the first parameter to `PilotMain`. (In the code segment, I've named it `wCmd`.)

Launch Codes

Launch codes can be issued by the system or by another application. Why would you receive a launch code? For example, many applications (such as the built-in ones) support the "find" interface. When the user wants to find a database record using the system find interface, the target application is launched using a special code, `sysAppLaunchCmdFind`, with the find parameters passed in the `pInfo` parameter to `PilotMain`.

An application can also receive a non-normal launch code for a system reset, a time change, or a notification that a hot-sync has completed. An application can even define its own launch codes to let other programs interact with it. For a complete list of the standard Palm launch codes, refer to the Palm SDK documentation.

The third and final parameter to `PilotMain` contains optional launch flags that are dependent on the launch code and the data passed in `pInfo`.

Now that you know something about launch codes, you can determine which launch code is passed to the little program and even do something about it:

```
   switch (wCmd)
   {
      case sysAppLaunchCmdNormalLaunch:
      {
         // Initialize application.
         break;
      }
      default:
      {
         err = 0;
         break;
      }
   }
   return err;
}
```

Here I included a simple `switch` statement that for now checks for the “normal” launch code. All other codes are ignored, so they fall into the default case.

Checking the Palm OS Version

Before you go too far in the program, make sure it is running on the required version of the Palm OS. As you saw in Chapter 1, “Introduction to Palm Programming and CodeWarrior,” there are a large number of Palm OS-based handhelds in the field, and they encompass a correspondingly large number of OS versions. With each new revision of the OS comes new functions and subsystems for the developer to take advantage of.

Although it’s wonderful to get new functions, some users still have older devices that are running versions of the Palm OS that do not support those functions. For example, infrared support became available only with the Palm III device, which contained version 3.0 of the Palm OS.

As a developer, you need to decide the minimum version of the Palm OS that you want to support in your application and check for that version in your application’s startup code so that you end your application gracefully on older devices. For an application that uses the infrared library, you would need to make sure that the OS was at least version 3.0. For an application that requires VFS support, you would need to make sure that the OS was at least version 4.0.

Listing 2.1 should be called first in any Palm application with a check for the minimum OS version the application is prepared to handle. In this case, I chose version 2.0.

LISTING 2.1 Checking the Application's Version for Compatibility

```
Err   err;

   // Check version compatibility.
   err = AppCheckVersion (0x02000000, fuLaunch);
   if (err)
   {
      // Incompatible - exit.
      return err;
   }

/*
   AppCheckVersion ():
   Check hardware compatibility.
   Parms:   dwReqVersion   - required ROM version.
            fuLaunch       - launch flags.
   Return:  0 if ROM is compatible or error code.
*/
Err
AppCheckVersion (UInt32 dwReqVersion, Int16 fuLaunch)
{
   // Assume the best.
   Err err = 0;

   // Get the ROM version.
   UInt32 dwRomVersion;

   FtrGet (sysFtrCreator, sysFtrNumROMVersion, &dwRomVersion);

   // Check if it's compatible.
   if (dwRomVersion < dwReqVersion)
   {
      // Version is too low, see what launch flags are.
      Int16  wLaunchCheck = sysAppLaunchFlagNewGlobals | sysAppLaunchFlagUIApp;

      if (wLaunchCheck & fuLaunch == wLaunchCheck)
      {
         // Tell user the problem.
```

LISTING 2.1 Continued

```
         FrmAlert (RomIncompatibleAlert);

         // Pilot 1.0 will continuously relaunch this app unless we switch
         // to another safe one.
         if (dwRomVersion < osVersion20)
         {
          AppLaunchWithCommand(sysFileCDefaultApp, sysAppLaunchCmdNormalLaunch,
        ➥NULL);
         }
      }
      err = sysErrRomIncompatible;
   }
   return err;
}
```

This code calls the Palm SDK function `FtrGet` to obtain the ROM version and compares it against the minimum required version. If the ROM version is too old, the program issues an alert and exits. (Alerts are covered in Chapter 6, "Interacting with the User: Forms." For now, it is enough to understand that it is a kind of pop-up message box.)

The Main Form

Of course, most Palm programs will display a user interface. On the Palm, you create a user interface by displaying one or more windows to the users. These windows are called *forms*. (See Chapter 6.) The main screen of a Palm application is commonly referred to as the main form.

NOTE

In this section, I present the basic code necessary to load your application's main form. You will need to create your form using CodeWarrior's Constructor tool or PilRC if you are using GCC as your toolset. If you already know how to use Constructor or PilRC to create a simple form resource, you might want to go ahead and do that now. Otherwise, I walk you through resource creation in Chapter 5, "Creating and Managing Resources."

In the last section, I included a `switch` statement to handle the `sysAppLaunchCmdNormalLaunch` launch code. Now, it's time to add some code that actually does something:

```
      case sysAppLaunchCmdNormalLaunch:
      {
         // Show the main form.
         FrmGotoForm (MainForm);
         AppEventLoop ();
         break;
      }
```

This code looks simple enough: If it is asked to launch the application, it shows the main “window” or form and then goes into what looks like an event loop. Both actions are interrelated. Let’s take a closer look at what’s happening.

`FrmGotoForm` is a Palm SDK function that, given a form’s resource ID (as defined in Constructor), will attempt to load the associated form. This will cause the Palm OS to pass a special event to the application, indicating that a form is about to be loaded.

Events

An *event* is a structure defined in the Palm SDK that can carry information to an application from the Palm OS. How do you receive events? In an event loop! Listing 2.2 shows you how.

LISTING 2.2 Using an `Event Loop` to Receive Events

```
/*
   AppEventLoop (void)
   Process events coming to application.
   Parms:   none
   Return:  none
   Note:    Application is in this function for majority
            of its execution.
*/
void
AppEventLoop (void)
{
   EventType   event;

   do
   {
      EvtGetEvent (&event, evtWaitForever);

      // Ask system to handle event.
      if (false == SysHandleEvent (&event))
      {
         // System did not handle event.
```

LISTING 2.2 Continued

```
        Int16       error;
        // Ask Menu to handle event.
        if (false == MenuHandleEvent (0, &event, &error))
        {
           // Menu did not handle event.
           // Ask App (that is, this) to handle event.
           if (false == AppEventHandler (&event))
           {
              // App did not handle event.
              // Send event to appropriate form.
              FrmDispatchEvent (&event);
           }  // end if (false == AppEventHandler (&event))
        }     // end if (false == MenuHandleEvent (0, &event, &error))
     }        // end if (false == SysHandleEvent (&event))
  }
  while (event.eType != appStopEvent);
}
```

AppEventLoop is a do-while loop that continually calls the Palm SDK function EvtGetEvent until it receives an appStopEvent, which indicates the application is closing down. For each event it receives, it is your responsibility to follow a standard protocol in determining who should handle the event. First, you determine whether the system wants to handle it by calling SysHandleEvent. Failing that, you determine whether the event belongs to the current menu (if any). If it does not, you have found an event that might be interesting to the application. Such events are passed to the AppEventHandler routine for closer scrutiny. If the AppEventHandler routine does not handle the event, it gets passed to the current form's event-handler function via the FrmDispatchEvent() call.

In AppEventHandler (outlined in Listing 2.3), I check the event type to determine whether it's something I am interested in. Because the primary job at this point is to get the main form loaded, I check whether the event is type frmLoadEvent, and if it is, I pass it to the AppLoadForm handler.

AppLoadForm performs three tasks that cause the main form to be properly loaded:

- It calls FrmInitForm to initialize the form resource.
- It calls FrmSetActiveForm to set the form to receive input from the users.
- It sets an "event handler" for the form.

In the next section, I explain form event handlers, but note that you determine which form is being loaded, rather than just assume it's the one and only main form. This step might be overkill for this application, but it reminds you that an application can have more than one "main form." For example, the Appointments application has several main "views." If you were to do the same, you would have similar load-handling code for each form. Listing 2.3 shows how to implement an application event handler.

LISTING 2.3 The Application Event Handler

```
/*
   AppEventHandler (EventPtr)
   Handle application-level events.
   Parms:   pEvent   - event to handle.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
AppEventHandler (EventPtr pEvent)
{
   Boolean bHandled = false;

   switch (pEvent->eType)
   {
      case frmLoadEvent:
      {
         // Load the form resource.
         UInt16  wFormID = pEvent->data.frmLoad.formID;

         AppLoadForm (wFormID);
         bHandled = true;
         break;
      }

      default:
      {
         break;
      }
   }
   return bHandled;
}
```

The Main Form Handler

The main form's event handler in the "Hello Palm" program is largely a placeholder: At this point, it only needs to trap the form's `frmOpenEvent` event and respond by redrawing the form. That's it! So why did you need a form event handler for "Hello Palm"? Most main forms do a little more than the one in "Hello Palm," which at the moment only stubbornly sits there, staring at the users. As you'll see in Chapter 7, "Form Elements," many events of significance, such as button taps, pass through the form handler. Given that almost all main forms eventually perform at least one or two tasks, it's a good idea to set up the infrastructure up front because it's boilerplate anyway. See Listing 2.4.

LISTING 2.4 Event Handler for the Main Form

```
/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;

   switch (eventP->eType)
   {
      case frmOpenEvent:
      {
         // Main form is opening

         FormPtr formP = FrmGetActiveForm ();

         FrmDrawForm (formP);

         handled = true;
         break;
      }

      default:
      {
         break;
      }
```

LISTING 2.4 Continued

```
   }
   return handled;
}
```

If you are familiar with event-driven programming on other computer platforms, this might look familiar, at least conceptually. An application is launched by the operating system in response to a user action. In response, the application does some initial housekeeping and setup, launches its main window, and then goes into an endless event loop until it receives an event indicating it's time to shut down. The main window, as well as each window created during the lifetime of the application, is responsible for handling its own events.

Hello Palm!

At this point, I've covered all the basic, boilerplate code elements that are a part of virtually every Palm program in existence. It's time to put all this together and create your first Palm application: "Hello Palm."

"Hello Palm" contains all the elements I've covered so far, along with a few minor additions. In anticipation of writing many more programs beyond this one, I broke out the "boilerplate" code from the code that is likely to change over time and created two separate .CPP files: helo_app.cpp and helo_mai.cpp. helo_app.cpp contains the `PilotMain` function, the event loop, and the application's event handler. This code will largely remain the same no matter what kind of application uses it. I then took the main form event handler and moved it into helo_mai.cpp. Because the main form will be different from program to program, it follows that this source file will change often as I reuse it in other projects.

The other addition is a common header file, hello.h. I use hello.h to define common values and provide prototypes for functions that will be referenced by more than one source file. Among other tasks, hello.h defines the creator ID. Creator IDs must be unique among applications installed on a Palm device. The examples in this book all use the creator ID "`TEST`," but be aware that this means that you can install only one sample program on a device at a time. Palm Computing provides a service on its Web site to let you register unique creator IDs for your company and its products. Those IDs are guaranteed by Palm to be unique so your application will not collide with any other application when a user installs it.

The Source Code

Listing 2.5 shows the source code for Hello Palm's main header file, hello.h. Listings 2.6 and 2.7 represent the application's main startup code and the main form handler, respectively.

LISTING 2.5 hello.h

```
/*
   HELLO.H
   Hello application declarations.
   Copyright (c) 1999 Bachmann Software and Services, LLC
   Author: Glenn Bachmann
*/

// Prevent multiple includes
#ifndef HELLO_H
#define HELLO_H

#define  HELLO_FILE_CREATOR   ('TEST')

#define  PALMOS_VERSION_20 (0x02000000)

#define  PALMOS_VERSION    (PALMOS_VERSION_20)

/*
   MainFormEventHandler (EventPtr)
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/

Boolean  MainFormEventHandler (EventPtr pEvent);

#endif
```

LISTING 2.6 helo_app.cpp

```
/*
   HELO_APP.CPP

   Hello application functions
```

LISTING 2.6 Continued

```
   Copyright (c) 1999 Bachmann Software and Services, LLC
   Author: Glenn Bachmann
*/

// system includes
#include <Palmos.h>
#include <SysEvtMgr.h>

// app specific includes
#include "hello.h"
#include "helo_res.h"


// Private constants.
const unsigned long  appFileCreator = HELLO_FILE_CREATOR;
const unsigned short appFileVersion = HELLO_FILE_VERSION;
const unsigned short appPrefID      = HELLO_PREF_ID;
const unsigned short appPrefVersion = HELLO_PREF_VERSION;
const unsigned long  appOSVersion   = PALMOS_VERSION;
const unsigned long  osVersion20    = PALMOS_VERSION_20;



// Private structures.

/*
   AppPreferences:
   Application preferences.
   Members:
*/
struct AppPreferences
{
   Char byReplaceMe;
};

/*
   AppInformation:
   Application information.
   Members:
*/
struct AppInformation
{
```

LISTING 2.6 Continued

```
   Char  byReplaceMe;
};

// Private functions.
static Err     AppStart ();
static void    AppStop ();
static void    AppEventLoop (void);
static Boolean AppEventHandler (EventPtr);
static void    AppLoadForm (Int16);
static void    AppReadPreferences ();
static void    AppWritePreferences ();
static Err     AppCheckVersion (UInt32, Int16);

// Private data.
static AppPreferences   s_appPref;
static AppInformation   s_appInfo;

/*
   PilotMain ()
   Main entry point into pilot application.
   Parms:   wCmd  - launch code.
            pInfo - struct associated w/ launch code.
            fuLaunch - extra info about launch.
*/
UInt32        PilotMain (UInt16 wCmd, MemPtr pInfo, UInt16 fuLaunch)
{
   Err   err;

   // Check version compatibility.
   err = AppCheckVersion (appOSVersion, fuLaunch);
   if (err)
   {
      // Incompatible - exit.
      return err;
   }

   switch (wCmd)
   {
      case sysAppLaunchCmdNormalLaunch:
      {
         // Initialize application.
         err = AppStart ();
```

LISTING 2.6 Continued

```
         if (!err)
         {
            // Show the main form.
            FrmGotoForm (MainForm);
            AppEventLoop ();
            AppStop ();
         }
         break;
      }
      default:
      {
         err = 0;
         break;
      }
   }
   return err;
}

/*
   AppStart ():
   Initialize the application.
   Parms:   none
   Return:  0 if success, or error code.
*/
Err
AppStart ()
{

   // Read preferences for the app.
   AppReadPreferences ();


   return 0;
}




/*
   AppStop ():
   Save the current state of the application.
   Parms:   none
   Return:  none
```

LISTING 2.6 Continued

```
*/
void
AppStop ()
{
   AppWritePreferences ();
}

/*
   AppEventLoop (void)
   Process events coming to application.
   Parms:   none
   Return:  none
   Note:    Application is in this function for majority
            of its execution.
*/
void
AppEventLoop (void)
{
   EventType   event;

   do
   {
      EvtGetEvent (&event, evtWaitForever);

      // Ask system to handle event.
      if (false == SysHandleEvent (&event))
      {
         // System did not handle event.

         UInt16        error;
         // Ask Menu to handle event.
         if (false == MenuHandleEvent (0, &event, &error))
         {
            // Menu did not handle event.
            // Ask App (that is, this) to handle event.
            if (false == AppEventHandler (&event))
            {
               // App did not handle event.
               // Send event to appropriate form.
               FrmDispatchEvent (&event);
            }  // end if (false == AppEventHandler (&event))
         }     // end if (false == MenuHandleEvent (0, &event, &error))
```

LISTING 2.6 Continued

```
      }        // end if (false == SysHandleEvent (&event))
   }
   while (event.eType != appStopEvent);
}

/*
   AppEventHandler (EventPtr)
   Handle application-level events.
   Parms:   pEvent   - event to handle.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
AppEventHandler (EventPtr pEvent)
{
   Boolean bHandled = false;

   switch (pEvent->eType)
   {
      case frmLoadEvent:
      {
         // Load the form resource.
         Int16  wFormID = pEvent->data.frmLoad.formID;

         AppLoadForm (wFormID);
         bHandled = true;
         break;
      }

      default:
      {
         break;
      }
   }
   return bHandled;
}

/*
   AppLoadForm (Word)
   Load a form.
   Parms:   wFormID  - form to be loaded.
   Return:  none
```

LISTING 2.6 Continued

```
*/
void
AppLoadForm (Int16 wFormID)
{
   FormPtr pForm = FrmInitForm (wFormID);

   FrmSetActiveForm (pForm);

   // Set the event handler for the form.
   //    The handler of the currently active form is called by
   //    FrmHandleEvent each time it receives an event.
   switch (wFormID)
   {
      case MainForm:
      {
         FrmSetEventHandler (pForm, MainFormEventHandler);
         break;
      }

      default:
      {
         break;
      }
   }
}

void
AppReadPreferences ()
{
   UInt16  cbPref = sizeof s_appPref;
   Err   err = PrefGetAppPreferences (appFileCreator,
                                       appPrefID,
                                       &s_appPref,
                                       &cbPref,
                                       true);
   if (err == noPreferenceFound)
   {
      // Clear the app preferences.
      MemSet (&s_appPref, sizeof (s_appPref), '\0');

      // App started okay, just no preferences.
      err = 0;
```

LISTING 2.6 Continued

```
   }
}

void
AppWritePreferences ()
{
   // Read preferences.
   PrefSetAppPreferences (appFileCreator,
                          appPrefID,
                          appPrefVersion,
                          &s_appPref,
                          sizeof (s_appPref),
                          true);
}

/*
   AppCheckVersion ():
   Check hardware compatibility.
   Parms:   dwReqVersion   - required ROM version.
            fuLaunch       - launch flags.
   Return:  0 if ROM is compatible or error code.
*/
Err
AppCheckVersion (UInt32 dwReqVersion, Int16 fuLaunch)
{
   // Assume the best.
   Err err = 0;

   // Get the ROM version.
   UInt32 dwRomVersion;

   FtrGet (sysFtrCreator, sysFtrNumROMVersion, &dwRomVersion);

   // Check if it's compatible.
   if (dwRomVersion < dwReqVersion)
   {
      // Version is too low, see what launch flags are.
      Int16  wLaunchCheck = sysAppLaunchFlagNewGlobals | sysAppLaunchFlagUIApp;

      if (wLaunchCheck & fuLaunch == wLaunchCheck)
      {
```

LISTING 2.6 Continued

```
         // Tell user the problem.
         FrmAlert (RomIncompatibleAlert);

         // Pilot 1.0 will continuously relaunch this app unless we switch
         // to another safe one.
         if (dwRomVersion < osVersion20)
         {
            AppLaunchWithCommand(sysFileCDefaultApp,
            ➥sysAppLaunchCmdNormalLaunch, NULL);
         }
      }
      err = sysErrRomIncompatible;
   }
   return err;
}
```

LISTING 2.7 helo_mai.cpp

```
/*
   HELO_MAI.CPP
   Main form handling functions.
   Copyright (c) Bachmann Software and Services, 1999
   Author: Glenn Bachmann
*/

// system headers
#include <PalmOS.h>
#include <SysEvtMgr.h>

// application-specific headers
#include "hello.h"
#include "helo_res.h"

static void    MainFormInit (FormPtr formP);

/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
```

LISTING 2.7 Continued

```
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;

   switch (eventP->eType)
   {
      case frmOpenEvent:
      {
         // main form is opening

         FormPtr formP = FrmGetActiveForm ();

         MainFormInit (formP);

         FrmDrawForm (formP);

         handled = true;
         break;
      }

      default:
      {
         break;
      }
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   formP - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr formP)
{
}
```

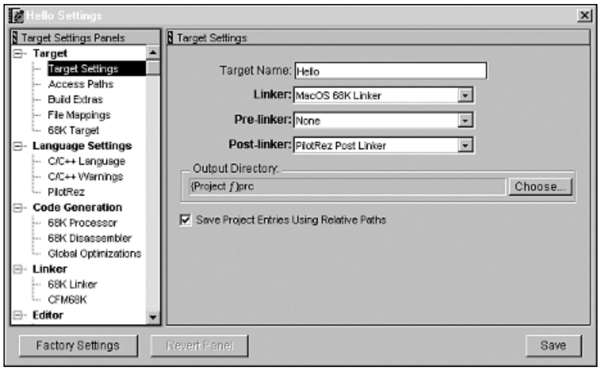
Creating a CodeWarrior Project for “Hello Palm”

All this code is nice, but it would be nicer if you could build it, wouldn’t it? This section shows you how to create a CodeWarrior project to properly build a Palm application.

Each project in this book is accompanied by the source code, including a CodeWarrior project file and GCC makefile, for each example. You are encouraged to use the project files for this chapter’s “Hello Palm” example as the basis for your own projects. (Like the old story about makefiles, nobody ever writes a makefile from scratch; they just copy one that works from somebody else.) I take a moment here to explain how “Hello Palm’s” project was created.

The way I see it, if you are using CodeWarrior, you essentially have two choices in creating a new Palm programming project: Create one from scratch from an empty project, or use one of the predefined Palm OS project types, which are called stationery by the folks at Metrowerks. The latter sounds like a sure winner, but in my experience, I’ve spent some unpleasant time fixing the undesirable side effects created by choosing these predefined projects as a template, so I now follow these steps to create a new project:

1. Start CodeWarrior and choose New Project.
2. Choose Empty Project in the Select Stationery dialog, and check the Create Folder option.
3. Name your project, which will autocreate a folder and a .MCP file of the same name.
4. Now, modify the project settings by selecting Edit, Hello Settings from the main menu.
5. Under Target, Target Settings, set the Target Name to Hello, the Linker to Mac OS 68K Linker, and the Post Linker to the PilotRez Post Linker, as in Figure 2.1.
6. Specify the directory where you want your .PRC file to be placed as the Output Directory.
7. To enable your project to be moved to another directory on your computer, check the option to save using relative paths.
8. Under Target, Access Paths, you need to set the paths where CodeWarrior will look for your source and output components. Check the Always Check User Paths option, and set the project path to be a \SRC directory, and add another path to be your \PRC directory (or the directory you specified as the output directory).



Target Name: Hello
Linker: MacOS 68K Linker
Pre-linker: None
Post-linker: PilotRez Post Linker
Output Directory: {Project ƒ}prc
[✓] Save Project Entries Using Relative Paths

FIGURE 2.1 Configuring project settings for "Hello Palm."

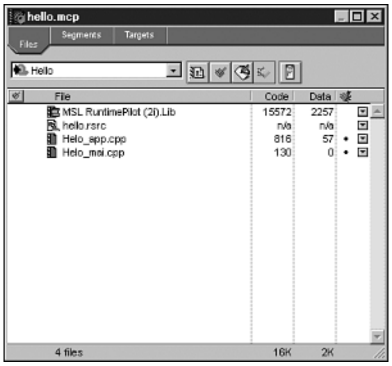
9. Under System Paths, ensure that it points to the location of your Palm support files (usually "Palm *x.x* Support" where *x.x* is the version of the OS).
10. Under Target, 68K Target, set the project type to be a Palm OS application, and name the .TMP file.
11. In Language Settings, PilotRez, set Mac Resource Files to be the name of the .TMP file specified in the last step. Also, check Disable UI Resource.
12. Under Language Settings, Output options, set the name of your .PRC file (for example, hello.prc). Also, set your unique creator ID here. (Use "`TEST`.")

That's all the configuration necessary. Now, you add files to the project as follows:

1. From the main menu, choose Project, Add Files.
2. Add the appropriate runtime library by selecting MSL Runtime 2i.lib, located in the Palm OS x.x Support path.
3. Add your source files (helo_mai.cpp and helo_app.cpp).
4. Add your resource file (hello.rsrc).

When you've added your files, the "Hello Palm" project should look like Figure 2.2.

If this sounds unpleasant, I admit it; it is unpleasant. However, I think it's a good idea to know how to create a project without the stationery option, and this way, I have full control over how my project is set up. Unfortunately, using the pre-existing stationery creates a project called "Starter," including template source files. It is (in my opinion) just as much work, and given that I have already identified the areas of the application that will be boilerplate and used from project to project, it is also unnecessary. You might want to experiment with both approaches and choose the one that works best for you.



File	Code	Data	
MSL RuntimePilot (2i).Lib	15572	2257	
hello.rsrc	n/a	n/a	
Helo_app.cpp	816	57	•
Helo_mai.cpp	130	0	•
4 files	16K	2K	

FIGURE 2.2 The main project view for “Hello Palm.”

Creating a GCC Project for “Hello Palm”

Before you proceed, a quick note: This book assumes that if you are using GCC as your development environment to build your Palm project, you have already set up PRC-Tools, and demonstrated that you can build correctly. If you have not performed this step, it is quite complex, somewhat error prone, and confusing to boot. Thus it is important that you follow the instructions provided at `http://www.palmos.com/dev/tools/gcc`. From Palm’s site you will need to visit the Cygwin site, and carefully follow each step until things are working well.

Hopefully, this is not the first time you are ever seeing a makefile. GCC projects are built using standard makefiles, which are a kind of script that automatically determines the dependencies in your project and can build them for you. What I present in the following listing is about as stripped down a makefile as you can get for the project in this chapter.

```
CC = m68k-palmos-gcc
CFLAGS = -palmos4.0 -g -DDEBUG_BUILD
APPNAME=Hello

SRCDIR=Src/
OUTPUTDIR=
RCPFILE=Hello.rcp

OBJS=$(OUTPUTDIR)Helo_app.cpp $(OUTPUTDIR)Helo_mai.opp

$(OUTPUTDIR)$(APPNAME).prc: $(OUTPUTDIR)$(APPNAME) $(OUTPUTDIR)bin.stamp
             $(APPNAME).def
```

```
   build-prc $(APPNAME).def $(OUTPUTDIR)$(APPNAME) -o
             $(OUTPUTDIR)$(APPNAME).prc  $(OUTPUTDIR)*.bin

$(OUTPUTDIR)$(APPNAME): $(OBJS)
   $(CC) $(CFLAGS) -o $@ $(OBJS)

$(OUTPUTDIR)%.o: $(SRCDIR)%.c
   $(CC) $(CFLAGS) -c $< -o $@

$(OBJS): $(SRCDIR)ResourceDefines.h $(SRCDIR)Utils.h

$(OUTPUTDIR)Main.o $(OUTPUTDIR)MainForm.o: $(SRCDIR)MainForm.h

$(OUTPUTDIR)Main.o $(OUTPUTDIR)SecondForm.o: $(SRCDIR)SecondForm.h

$(OUTPUTDIR)bin.stamp: $(SRCDIR)$(RCPFILE) $(SRCDIR)ResourceDefines.h
   ( cd $(OUTPUTDIR); rm *.bin; pilrc -I
             ../$(SRCDIR) ../$(SRCDIR)$(RCPFILE))
   touch $(OUTPUTDIR)/bin.stamp


clean:
   rm -f $(OBJS) $(OUTPUTDIR)$(APPNAME) $(OUTPUTDIR)$(APPNAME).prc
         $(OUTPUTDIR)*.bin $(OUTPUTDIR)bin.stamp
```

Building and Running “Hello Palm”

To build “Hello Palm” in CodeWarrior, access the Project menu in CodeWarrior and choose Make (F7). Once you’ve addressed any compilation problems, you should see two new files in your output directory \PRC, hello.prc and hello.psym. hello.psym contains symbolic debugging information that will be used when tracing through your code with CodeWarrior’s Debugger. hello.prc is the actual program.

To build “Hello Palm” in GCC, simply open up a DOS command prompt, navigate to the root directory of the Hello project (where the makefile is), and type **`make`**.

Your program is transferred to the Palm device in the same manner as any program you might have purchased or downloaded off the Web: Use the Palm Install tool to locate and select the .PRC file, and use hot-sync to transfer the file. After the hot-sync is complete, you should see “Hello Palm” listed in your Palm’s main application list.

Figure 2.3 shows what “Hello Palm” looks like when it’s running.

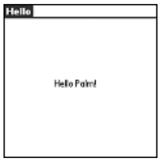


FIGURE 2.3 Hello Palm!

Summary

In this chapter you got your first look at actual Palm application code. Many of the topics covered will become standard elements in the sample applications used in the rest of *Palm OS Programming*, including `PilotMain`, launch codes, version checking, event loops, and the main form. You also created your first project using CodeWarrior and GCC, which you will also reuse throughout this book.

The next two chapters will round out the introductory section of the book, and complete the coverage of the basics of Palm programming.

3 Rapid Development Using POSE, the Palm OS Emulator

IN THIS CHAPTER

If you've programmed for other computing platforms before, you might be starting to realize that developing for a handheld device such as Palm may be a bit more cumbersome than you are used to. After all, the well-known cycle of write a few lines of code, compile, debug is something most programmers are comfortable with. But in Palm development, your program runs on an external device, not your PC, right? So does that mean every time you make a change in your code you will have to install a new copy of your program to your handheld device just to see if it runs properly? And what about debugging? How does that happen when your program isn't even running on your PC? Thankfully, there is a solution, and it is part of the Palm OS development toolset.

The Palm OS Emulator, commonly referred to as POSE, "POSEr," or just "The Emulator," is a wonderful piece of software that provides an emulation of the Palm OS on Windows (a version also exists for Macintosh platforms) so that you can run, test, and debug your application without having to physically download it to a Palm device. POSE runs as a native Windows application, except that instead of the normal look and feel of a Windows app, POSE presents a screen and user interface that is a replica of the Palm OS. When you run POSE, you can do just about everything you can with a physical device, including working with the built-in applications (Datebook, Address Book, To Do, and so on) and running the calculator; even the hardware buttons are emulated properly. You can even perform a HotSync!

What this means for Palm developers is that for much of the normal software development cycle, most Palm OS applications can be tested and debugged on your PC using POSE, rather than having to download the program to the handheld. Note that no emulation is perfect, and there is no substitute for running your application on an actual device. But for most types of applications, having POSE eliminates the need to download to the device for a good portion of the software development process.

In this chapter, I describe how to use POSE during your Palm OS development. I also go beyond the basics of running your application in PC by describing some advanced techniques that will really help you get the most out of POSE. The next chapter builds on the concepts learned here by showing you how to debug a live application with the help of POSE.

Where to Get POSE

As of this writing, the latest Palm OS Emulator (POSE) version is 3.4, and new versions of POSE seem to appear every few months or so. This chapter describes the current capabilities of POSE, but please note that because new versions are being released all the time, you should frequently check the Palm OS Core Tools Update Web page (`http://www.palmos.com/dev/tools/core.html`) for the latest version of POSE (as well as for other development tools). At any given time there is a current "official" version available for download, as well as a "seed" version that is available for testing.

Although a version is installed as part of the Metrowerks CodeWarrior development toolset, POSE is also available as a standalone, free download. In fact, it is safe to assume that the version of POSE that came with your copy of CodeWarrior is already out of date. The best way to ensure you have the best available version of POSE is to frequently check the Palm OS Web site.

ROM Images

If you download POSE, install it, and run it, the first thing you will see is a prompt to create a new session, and to supply something called a "ROM." You can think of the ROM as a snapshot of the software environment that runs inside a Palm OS device, including the operating system itself, the applications shell and preferences, system components, and the built-in applications.

Just as your PC cannot run without an operating system like Windows, POSE cannot run without a ROM. POSE itself represents an empty shell of a house—supplying POSE with a ROM is like moving in, putting up wallpaper, adding furniture, and stocking the refrigerator. The ROM makes POSE a living, breathing, environment for running Palm OS-based applications.

As you shall see, there is in fact no real difference between ROMs that are used in POSE and the ROMs that are found inside the actual devices. In fact, Palm OS licensees such as Handspring and Sony get tools from Palm that allow them to create their own unique ROMs that are distributed inside of their devices.

Where to Get ROMs

So where does one get a ROM? This is one of the more frequently asked questions by new Palm developers. Many new developers will ask for someone to simply email them a ROM. This is a pointless question, so don't even bother to ask. A ROM is in fact a full copy of the Palm operating system, so it is actually a violation of the license agreement to distribute unauthorized copies of the ROM without permission from Palm.

A ROM is legally obtained by transferring it from an actual Palm OS device or preferably by joining the developer program at Palm. Although it is possible to get by with transferring a single ROM from your Palm OS device, doing so is a cumbersome process, and unless you are keen on purchasing one unit of every Palm OS device ever shipped, you will not get to use POSE to emulate the wide array of devices that your application may encounter in the real world. Thus I highly recommend that you join the Palm developer program, which gives you not only access to a full set of ROMs to test with, but also to a wealth of tools and support from Palm Inc. You can join the developer program at `http://www.palmos.com`.

After logging into the developer area at Palm OS, you will find you have access to ROMs representing every operating system version that shipped with any modern Palm OS-based device. This instantly gives you the capability to test your applications on every device that is out there, which is a phenomenal benefit of obtaining ROMs through the developer program.

Note that at present, the developer area does not include ROMs from Palm OS licensees; only ROMs from Palm-branded devices are available. For Handspring, Sony, Symbol, or other licensee ROMs, you will need to join the licensee-specific developer programs as well before gaining access to them. Depending on the likelihood that your customers will run your application on the devices, it is probably a good idea to obtain those ROMs for testing as well.

Because Palm is breaking out its OS group from its device group, and is portraying the device group as just another Palm OS licensee, it is unclear at this point whether the OS group will able to make available ROMs from all licensees, or whether the OS group might need to direct developers over to the Palm device group for ROMs in the same manner as other licensees.

The Different Kinds of ROMs

Aside from the ROM variations due to operating system versions and devices, there are a couple of other types of ROMs that are interesting to investigate. The first is what is known as a "debug" ROM. Most of the available Palm ROMs come in "non-debug" and "debug" flavors.

The non-debug ROMs are the same as the ROMs found on devices. The debug ROMs, however, are special versions of the device ROMs that are designed to subject applications to a "stricter" operating environment than that of the regular device. A debug ROM offers extra error checking and debugging features above and beyond what a normal ROM has. Running your application under this environment can help you catch and repair poor programming practices and buggy code that would otherwise be hard to recognize when your application is running on an actual device.

It is highly recommended that you use the debug ROMs regularly in testing your application under POSE. So why wouldn't you just use the debug ROMs all the time? Although strict error checking can expose the most problems with your code, at times in the development of your product it may be inconvenient and/or distracting to have every little error pop up. Also, POSE can be used to demonstrate applications, either internally among development teams, or even to clients and customers. In those situations, you would want to suppress debugging alerts and messages.

Beginning with Palm OS 3.0, ROMs became available in non-English languages. ROMs are now available in English, French, Italian, German, Spanish, and Japanese. Starting with OS 3.5.1, the non-Japanese target language may be selected within a single ROM by the POSE user. Previous versions of the OS require an entire ROM to be distributed for each target language.

Because the Internet generally provides easy access to your applications from all over the world, it is worthwhile testing your application under a variety of languages for compatibility. These ROMs also can be helpful when you create localized versions of your programs.

Skins

In addition to the ROMs and POSE, the last thing you might want to obtain is a "skin." For the most part, a skin simply provides POSE with the ability to take on the appearance of a specific target device, such as a Palm m505 or a Sony Clie 615. POSE does come with a generic skin, so technically you do not need to obtain extra skins.

Although the use of skins can make for a more eye-catching POSE experience and will almost certainly impress all your friends, generally speaking a skin has no positive or negative effect on the actual operation of POSE. The one exception that I will point out is in the instance of a device with a radically different form factor. For example, the Handspring Treo communicator, which is a combination cell phone, PDA, and wireless device, comes with an integrated keyboard as well as a fairly drastically remapped set of buttons on the device. It can be somewhat disorienting to emulate such a device using the generic skin, and some functions may only be accessible by using the skin of the target device. But at present this is the exception rather than the rule, and personally I tend to use the generic skin more often than not.

NOTE

One final note on skins, ROMs, and POSE. Several Palm OS licensees ship a custom version of POSE that is specially enabled with options and extensions that are unique to their device. If a device manufacturer offers it, it is generally a good idea to use the device-specific version of POSE with the appropriate ROMs.

Creating, Loading, and Saving POSE Sessions

Now that you have POSE and have at least one ROM to use with it, you are ready to begin running POSE.

The first time you run POSE, it will prompt you to create a new "session." Sessions are saved snapshots of your POSE activities, and are bound to a specific ROM, device type, and memory configuration. Sessions make it easier to create and maintain a wide variety of test environments to run your applications in.

To set up a new session, choose a ROM file to test with, a target device, a skin (optional), and a "RAM size." The RAM size will help you emulate a specific device with a specific amount of memory (for example, a Palm Vx with 8MB of memory), or can be used to help you stress-test your application under low memory conditions. Note that the device options are partly dictated by the ROM you have chosen (not every device supports every ROM). Also note that skins will show up only if they are installed under the Skins subdirectory of your emulator installation location.

After a brief flash, you will see the familiar "boot" sequence of a normal Palm device, followed by the appearance of the Preferences screen. Congratulations, you are now running the Palm OS on your PC! All the user actions you would normally perform on the device are supported—you can tap on the Datebook hardware button to launch Datebook; you can tap on the silk-screened Home button to run the default Palm Launcher; you can even (awkwardly) scribble characters using Graffiti in the Graffiti area.

In general, a left mouse click is equivalent to a stylus tap on an actual device. Thus you can operate your POSE session just as you would your real device. An added bonus is that in text input fields (for example, the Memo application), you can use your PC keyboard to perform data entry, an option that's preferable over attempting Graffiti with a mouse!

Right-clicking on any area of the POSE screen invokes the POSE menu system. The POSE menu system provides you with access to a variety of POSE features that control the behavior, appearance, and functionality of POSE in your current session. Note that right-clicking does not have any effect on the application currently running in POSE. The application is put into a kind of suspended animation while the POSE menu system appears, and resumes running when you are done with the menu.

As you can see, there are quite a number of menu options that appear when you right-click. We will get to most of them later; for now let's focus on the first three panels, which contain the options you will be using most often.

The Exit menu option will cause POSE to close down and exit altogether. If you have made any changes to the ROM during your current session, POSE will ask you if you want to save your session before closing. A change is anything that affects the memory storage of your ROM, including new Datebook entries, installation of new applications or databases, changed POSE settings, or a new POSE session. In all of these cases, POSE will offer to save your session under a name of your choosing, so that you can return to the session later if you want.

You can also invoke the save function explicitly by tapping Save or Save As on the menu's third panel. In some cases it may take a significant amount of time to set up a perfectly configured session with the right options and installed applications and data, so it is worthwhile to save your session often, just as you would frequently choose to save a long document you were typing into a word processor.

POSE sessions are saved as files with a .PSF extension, and you can name them anything you want, although it often helps to use a descriptive name that gives you a clue as to the operating environment contained in the session. A complete snapshot of your session, including all databases, applications, program options, and even the last running application, is saved in this file.

To load a previously saved session, load POSE, and either choose a .PSF file at the first screen, or right-click on the POSE screen and choose the Open menu. Upon successful loading of the .PSF, POSE will return you to the state as you last left it in the saved session.

You can create as many new sessions as you want by right-clicking and choosing New from the menu. Each new session is created as a blank slate using the ROM and memory configuration you choose. If you want to make multiple copies of saved sessions, you can use the Save As function to save a current session under a different name. This can be helpful if you are about to perform an action that may cause a program error or a fatal reset. You can then quickly reload the saved copy to get back to the spot prior to the error.

Installing Applications and Databases

The real benefit to using POSE is in running your own programs under the emulator, thus enabling you to observe your program under a wide variety of OS versions, devices, and configurations. So how does one get an application onto POSE?

To install applications and other Palm files onto your PDA, you normally would use the Palm Install tool, which can be launched standalone or from the Palm Desktop's launch toolbar. The Palm Install tool queues files for synchronization to the PDA on the next HotSync. (Although some applications provide their own installation programs that bypass the Install tool and deposit files directly in the queue.)

POSE, on the other hand, does not interface with the Install tool (although this might be a nice feature to see in a future version to help test a program's installation procedure). Rather, you can use one of two methods to directly copy one or more Palm files (.PRCs, .PDBs, or .PQAs) to the current POSE session.

The first method is by right-clicking on the POSE screen, thus bringing up the POSE system menu. Choose the menu option Install Applications and Databases, and you will be presented with a standard File Open dialog box that allows you to browse your hard drive for Palm applications (.PRCs) and databases (.PDBs). Select one or more files, and POSE will dutifully install them to your POSE session. Even better, when you save your session (thereby saving the state of your POSE ROM), POSE will save the installed files as well. This allows you to build up an emulation of a Palm device that contains any combination of applications and databases.

An alternative to using the Install menu is to use the drag-and-drop method, which involves selecting one or more Palm files from your PC's desktop and dragging them with your mouse on top of the POSE screen.

One anomaly you should be aware of is that if you have the standard Palm Applications screen displayed when you add new files to POSE, the new files will not automatically show up on your screen until you switch to another application (such as Calculator) and switch back, thereby forcing the Applications screen to rescan the files on the system.

Taking Screenshots

Another nice feature that POSE offers, in addition to helping you test and debug your program, is the ability to take a screen capture, or screenshot, of any POSE screen. This can be very helpful in creating program documentation, or in giving visitors to your Web site a sense of what your application looks like.

To take a screenshot, simply run your application in a POSE session (or run any application whose screen you want to capture). Then right-click on POSE to bring up the POSE system menu, and choose Save Screen. After giving a name to the saved screen image, POSE will then store a Windows Bitmap (BMP) containing a perfect 160×160 image of your screen.

Note that the captured screen does not include the silk-screen graffiti area, nor does it include the POSE skin. If you want to capture those areas, you can use the tried and true “print screen” technique to capture the screen to a bitmap, and perform editing/cropping from that point.

Creating a Bound Copy of POSE

Another convenient feature of POSE is the ability to create what is referred to as a “bound emulator.” What this means is that you can take any POSE session and instruct POSE to build a self-contained executable program that consists of POSE itself, along with your selected ROM and configured settings, and any programs you installed. Once created, the bound emulator, in the form of an executable (.exe) file, can be distributed to others and run as a standalone version of POSE, without the recipient needing to obtain a full copy of POSE or a Palm OS ROM.

This is a handy way to give others a demonstration version of your application; I’ve used it several times as a convenience for clients and customers who might otherwise have needed to install my programs on their Palm devices in order to understand how they work.

As with other POSE features, this one is just a right-mouse click away, by choosing Save Bound Emulator. Simply choose a name for the bound emulator session, and you will have a distributable version of POSE complete with your program running in it.

Advanced POSE Techniques

Thus far I have given you an overview of the basics of POSE, including how to create, load, and save sessions, as well as how to run your program in a POSE session. These features alone make POSE an indispensable tool for observing and demonstrating your program running under a wide variety of Palm OS devices and configurations.

The remainder of this chapter delves deeper into some advanced POSE features that are instrumental in testing and debugging your application. We also take a look at some advanced configuration options that can be used for specific types of Palm applications.

POSE Warnings and Error Messages

POSE can produce a significantly greater amount of warnings and error messages than you would ever see running your program on an actual device, especially if you choose to run POSE with a debug ROM (again, highly recommended).

Although initially annoying, these messages should not be dismissed. The messages are POSE's way of telling you that there is either a bug in your program, or that you have used a programming technique that might not be supported in future versions of the Palm operating system. Pay attention to POSE, it is trying to help you!

To get an understanding of the types of messages POSE produces, right-click on POSE and choose Debug Options. A screen will appear that lists the various types of program checking that can be enabled or disabled. You can disable some or all of the options to suppress most of the errors that POSE is displaying. Occasionally this is called for in order to have a cleaner debugging session, or when perhaps you are demonstrating a pre-release version of your program to a customer.

Under normal POSE operation, however, it is highly recommended that you leave the debug options checked, and that you set some time aside to investigate what part of your code is causing the error conditions to trigger. A quick look at the *POSE User Guide* (installed under the Docs directory in your POSE folder) yields a detailed description of each debug option, and most of the options are designed to catch true error conditions in your program. Certain specialized applications (such as games or imaging programs) that need to write to screen memory may require turning off the Screen Access option, but most of the others will indicate that you have a legitimate problem in your code that needs attention.

Logging

POSE offers a built-in logging facility that you can use to see into the inner workings of your program without resorting to a source-level debugger. Although the current version of POSE does not yet support the ability to automatically log your own program's internal function calls, POSE's logging facility can log system calls, serial port activity, and a number of other types of events that might otherwise be hard to track.

A good example of a program that can benefit from logging is a TCP/IP-based application that performs NetLib calls. Such programs are not always easy to debug using the normal CodeWarrior tools. Oftentimes all you need is to be able to see a log of the types of networking calls occurring and in what order they do so.

In short, you should have an idea of what you are looking for before turning on all of the logging options. A tremendous number of events can be generated in a very short period of time, which can obscure your ability to find what you are looking for (to say nothing of the performance impact you will have on POSE as it attempts to log all of the events).

Profiling

Profiling is a technique used by some software developers to try to determine what parts of their program code are using the most processing time. Although profiling is not a tool that you will need on a daily basis, it can often be just the trick for peering into the inner workings of your code.

POSE offers a special profiling version of POSE (called "Emulator-Profile.exe "), which you can use to report on the execution of your code. As with logging, it is best to know what you are looking for, rather than simply profiling your entire POSE session.

POSE does offer some control over when to start and when to stop profiling, thus helping you zero in on a certain user action or part of your program's execution.

The results of your profile session are saved in your POSE directory in a text file, along with a measurement of how long POSE spent in each function.

One important note is that you must be compiling your program with debugging enabled (see your compiler manual for how to enable this) in order for POSE to produce readable output with real function names.

TCP/IP and Serial Communications Support

If you are developing an application that uses the Palm OS NetLib TCP/IP library, you will be pleased to know that POSE can help you test your networking code as well. POSE can actually be set up to redirect any TCP/IP calls you make to the TCP/IP stack that is provided on your PC.

Similarly, if your application performs serial communications using the Palm OS Serial Manager, POSE can be configured to map the Palm OS serial port to one of the available serial ports on your host computer.

Gremlins

POSE supports a special automated testing facility called *Gremlins*. Just like the gremlins of legend that sought to perform evil mischief with airplanes, POSE Gremlins are used to put your application through what might be considered the "worst-case-user" scenario.

For the reader not familiar with the concept of automated testing, these tools present the opportunity to randomly throw events such as button presses, stylus taps, text entry, and application switching at your program. A log of how your program stood up under this test is recorded, allowing you to review and debug any problems encountered.

For software developers, the thought of an end user exercising every possible menu, button, or user interface element in their application over and over thousands of times is terrifying—and that is just what the POSE Gremlins feature does!

Before you run away in fright, abandoning the thought of ever running Gremlins on your application, consider the following:

- Gremlins is probably the most efficient way to flush out hard-to-find instances of bad memory handling and questionable use of Palm OS function calls.
- Would you rather have POSE tell you about problems in your code now, or would you rather hear it from your customers?
- Passing a certain level of Gremlins testing is a required step in order for a software product to bear the Palm certification logo.

With these thoughts in mind, the next chapter will focus on debugging tools and techniques for Palm OS applications. As part of the chapter, the Gremlins facility is discussed in more depth, and an actual Gremlins session is performed as an example.

The OS 5 Simulator

As this book is being written, Palm is readying the introduction of Palm OS 5.0. OS 5 will be the first version of Palm OS designed to run on the more powerful ARM processor (OS 4 and previous versions ran on the Motorola Dragonball processor).

Palm OS 5 will continue to support testing and debugging Palm applications on a host PC, but due to the significant changes in the lower levels of the operating system, POSE will not be capable of running the new OS. Palm is instead providing a new tool called the “Palm Simulator” (or PalmSim for short).

Rather than consisting of a single monolithic “emulator” program, PalmSim is made up instead of many smaller components built as native Windows DLLs. Despite some differences in appearance and operation, PalmSim still provides essentially the same type of functionality, enabling Palm OS applications, even those built for previous versions of the Palm OS, to run and be tested and debugged under the OS 5.0 environment.

We will cover some of the differences between the new Palm OS 5.0 tools and the existing 4.x toolset throughout this book, culminating in an entire chapter that presents an overview of the new OS and its impact on developers.

Summary

POSE has taken a rightful place alongside editor, compiler, and debugger as among the most valuable assets in a Palm developer's toolbox. The ability to test and debug Palm applications in a near perfect emulation of the Palm OS, as well as the advantages of being able to run tests on virtually any OS and configuration without owning the actual devices, combine to make POSE one of the most frequently loaded programs on my computer.

This chapter provided an overview of POSE as a standalone tool, exploring some of its many useful features. The next chapter makes use of POSE in tandem with a debugger, a very powerful combination that presents the fastest, most effective way to identify and fix coding problems for Palm developers.

4

Debugging Techniques for Palm Applications

IN THIS CHAPTER

- Why Palm Debugging Is Different
- Defensive Programming
- Effective Debugging Techniques for the Palm
- The CodeWarrior Debugger
- Other Debugging Tools
- Using Gremlins to Find Problems

Although debugging is often considered something you do after a program is written, the best time to make plans for how you'll debug an application is before you write the first line of code. If you don't understand how to effectively use the tools at your disposal now, you will be in deep trouble when you have a 100,000-line (or longer) program that does not work properly, and you have no idea how to tell what's wrong.

Because of the unique challenges in debugging a Palm application, unless you get a handle on how to efficiently locate and resolve some of the most common programming mistakes on the Palm, you might find yourself faced with a horribly long and frustrating compile-edit-debug cycle. Thankfully, there are tools and techniques available to shorten this cycle significantly.

It might seem premature to be diving into debugging techniques at this stage of your tour through Palm programming, given the fact that you have barely written any code yet. However, by learning a few valuable techniques now, with very simple projects, you will be able to establish a foundation for quickly finding and fixing bugs in far more complicated projects.

In this chapter, we explore some of the best ways to achieve faster development times by making good use of debugging tools and techniques available to the Palm programmer. Specifically, we will cover:

- Which factors make Palm debugging different for developers
- Available debugging tools and techniques for Palm programmers

- An actual Palm debugging session
- Effective use of POSE, the Palm OS Emulator
- Use of the Gremlins feature of POSE to automate detection of errors

Why Palm Debugging Is Different

Programming for the Palm environment presents some unique problems to the developer in terms of program testing and debugging. Probably the single most glaring of these problems is that on the Palm, your code is written, compiled, and linked on a different host system from the target OS.

If you think about it, you typically develop your Palm application using the CodeWarrior tools, which are hosted on Windows, Macintosh, or whatever your chosen desktop environment is. If you wanted to run your program on an actual target device, your application's .PRC file must be compiled, linked, and installed via the Palm Install tool and then subsequently hot-synced to the device. Then you could run your application on the device, and observe any problems it has. If you found a problem, you'd go back to your project, and begin sifting through the code looking for potential causes of the problem.

For those of us who have become used to integrated development environments and fast source-level debugging, this is starting to sound ugly. Clearly after a few cycles of this, you'd start looking for a better way to develop and test your application or you will be in for a very tedious development cycle.

As we hinted at in the previous chapter, the availability of POSE (the Palm OS Emulator) greatly improves the compile-edit-debug cycle. Because your program can run on the same computer that it is being written on, no hot-sync transfers are necessary. POSE has come a long way since it first appeared on the scene, and has much to offer as a means of effectively debugging a Palm application, both with and without the use of a source-level debugger.

Defensive Programming

Many programmers are tempted to seek the shelter of their debugger at the slightest hint of anything wrong. In fact, debugging your source code line-by-line should not be your first step in diagnosing coding problems, no matter what your target environment (Palm or otherwise). This is especially true when you consider that starting a debugging session for a Palm application is much more time-consuming than for a PC application. If you can get your program to provide enough hints as to the nature of the problem without having to resort to an actual debugging session, you will save many hours of tedious debugging.

One of the best ways to debug your application is to use development practices that minimize the time you have to spend in front of a debugger. It sounds odd, but the most efficient debugging you can do is performed without a debugger. Even with the best, state-of-the-art debugging tools, staring at your program as it executes line-by-line is slow and tedious at best. When you consider that the Palm platform has not been around long enough to have attracted the state-of-the-art tools found elsewhere, you can start to see that the debugger should be used as a last resort.

As with any other environment, for the Palm programmer the best offensive strategy is a good defense. By defense, I mean you should write your code with the attitude of expecting the worst-case scenario. With every line of code you write, you should be thinking ahead, wondering, “What could go wrong here, and how would my program react?”

Seasoned software developers use several techniques that help expose problems with their programs early and isolate the problem to a single module, function, or even line of code. Some call this collection of techniques “debug scaffolding” because as you develop the “real” code, you simultaneously build up a growing framework of code designed to catch a wide variety of errors and give you visibility into the inner workings of your program.

Defensive programming starts with good error handling. The C programming language provides you with the ability to write functions that return error codes, and to check those codes. You should diligently check all functions called, whether the functions are yours or part of the Palm OS, and provide some sort of safe handling should the function return an error. Sad to say, many programmers write what I refer to as “sunny day” code: code that expects only the best-case scenario when it runs, blissfully ignoring the possibility of failure. In fact, this is one of the most common causes for program crashes. Suffice it to say that not performing basic error checking turns a deaf ear to what your program and the operating system is trying to tell you about what is happening in your program.

Effective Debugging Techniques for the Palm

This section walks through some of the defensive programming techniques that work well across all computing platforms, and that work especially well in Palm programming. Specifically, we will cover the use of the Palm Error Manager, Error Logging, and Alerts.

The Palm Error Manager

The Palm Error Manager supports several useful macros that can help your program report unexpected conditions. These macros work on the same general principle as the well-known `assert` macro available on many other platforms. You give the macro a condition to test and a message to display if the condition triggers an error message. The error displayed includes not only the specified text but also the name of the source code module and the relevant line number.

Three macros are supported, including

- `ErrDisplay`—Always displays an error message.
- `ErrFatalDisplayIf`—Displays a fatal error message if the first argument is evaluated to be `TRUE`.
- `ErrNonFatalDisplayIf`—Displays a nonfatal error message if the first argument is evaluated to be `TRUE`.

You can control the level of error checking by setting a special macro value `ERROR_CHECK_LEVEL` to `ERROR_CHECK_FULL` (which enables all three error calls), `ERROR_CHECK_PARTIAL` (which enables `ErrDisplay` and `ErrFatalDisplayIf` calls), or `ERROR_CHECK_NONE` (which disables all `Err` calls).

It is important to understand that the level of alerts supported is determined at compile time, but the error condition check is performed at runtime.

Although similar to the `assert` macro in function, in actual usage `ErrFatalDisplayIf` and `ErrNonFatalDisplayIf` act like the reverse of `assert` in that the alert is triggered if the condition is `TRUE` rather than `FALSE`. For example, the following code determines whether a pointer is `NULL`. If the expression `p == NULL` evaluates to `TRUE`, the message `Null Pointer` is displayed:

```
ErrFatalDisplayIf ( p == NULL, "Null Pointer!");
```

The question of how much error checking should be left in your code when it is released to your users is the subject of a long-running debate that is not specific to the Palm platform. Different developers follow different schools of thought; some think that these checks are developer tools only, whereas others firmly believe that they are invaluable tools for determining system problems in the field. At minimum it would be wise to include assert/error manager calls in a debugging build of your program, if not in the actual released code.

FrmCustomAlert

Although using the Error Manager macros can help signal a variety of conditions in your program, they are limited in terms of how much information they can display. If you want to display more information about what is going in your code, you can make use of a message box-like facility called an “alert.” An alert is a specialized kind of form, or dialog box, that can display a message to the user in a pop-up window.

An alert appears to the end user as a message box with a title, some text, and one or more buttons that dismiss the message. The topic of how to add different kinds of alerts to your program is covered in more depth in Chapter 6, "Interacting with the User: Forms," but it is worth mentioning them here because alerts are a useful tool for displaying data to the users in the case of errors or other program conditions.

Alerts can be created in a couple of different ways, but the most useful method in terms of displaying debugging information is the `FrmCustomAlert` function. `FrmCustomAlert` allows you to specify an "alert resource ID" (alerts and other resources are explained further in Chapter 5, "Creating and Managing Resources"), as well as up to three text buffers.

Typically to display a debugging message to the users, you would enter the following code:

```
Char *pMessage = "This is an error message";
err = FrmCustomAlert (ErrorAlertID, pMessage, "", "" );
```

What you put into `pMessage` is up to you. You can use the `StrPrintF` function to combine program variable values, error codes, or other helpful information.

Aside from display of debugging information, as part of good defensive programming, error checking should normally be combined with generous use of `FrmCustomAlert` to let your users know what is going on when things go awry. Writing good error checking code means not only comparing function return values, but also alerting the user that there is a problem. Most would agree that it is better to alert the user to an error condition than to allow an error condition to eventually cause the device to crash or reset.

Logging

In many situations, message boxes and Error Manager macros are too obtrusive or awkward to employ. For example, if your Palm is receiving data through the serial or infrared port from some other device or host, it is not acceptable to display a modal message requiring user intervention for every incoming message or packet of data.

What is really needed is an unobtrusive "error logger" window that can display a stream of events as they happen. This is actually pretty easy to code and involves simply using the `WinDrawChars` function to draw directly on the screen.

One way to use on-screen logging is by creating a test program that exercises the functionality in question, and then creating a main form that devotes a section of its display area to your logging messages. This arrangement lets you see each event as it happens, but with a little work, you can actually create a scrolling list of events.

Going further, you can even log your program's activity to a file using the file-streaming APIs provided by Palm, and you'll be able to perform a post-mortem examination of your program's session.

The CodeWarrior Debugger

Despite the usefulness of the techniques described thus far, there comes a time when there is no substitute for being able to debug your program line-by-line.

When you finally need to sit down and step through your code line-by-line, you need a source-level debugger. CodeWarrior provides a source-level debugger that can let you control and observe your application's execution either on the Palm device or under POSE, the Palm OS Emulator.

If you have your project loaded in CodeWarrior, you can begin a debugging session at any time by choosing the menu Project, Debug. But before you start debugging, you will need to configure CodeWarrior's debugging settings by choosing the Edit, Preferences menu. In the dialog box that appears, you will need to go into the last category of settings from the list on the left (Debugger), and click on Palm Connection Settings. Here you can configure whether you want to debug your program when it is running in POSE or on an actual device. Unless you have a specific need to debug on the device, for the most part you will be debugging using POSE, so choose Palm OS Emulator as your target. The other settings should be left as the default. Note that although there is a check box for Always Launch Emulator, which launches POSE whenever you launch CodeWarrior, this can be wasteful and in practice it is best to start POSE manually before starting a debugging session.

A Debugging Session

You are now ready to run your first debugging session. For the purposes of this chapter we will use the "Hello" program introduced in Chapter 2.

With your project loaded in CodeWarrior and POSE running on your PC, choose the Project, Debug menu option. After a few seconds, a new window will appear in CodeWarrior that contains three panels.

NOTE

Note that you can also launch a debugging session by double-clicking the .PSYM file generated by the most recent build of your program. This is located in the same folder as your program's output .PRC file.

The Stack panel displays your program's stack. At the start of a session this will be positioned at the beginning of the PilotMain function. As you progress through running your program, this stack listing will change to reflect the nested function calls that are made in your program. When you are at a break in your debugging session, you can click on a function that appears in the stack and view the state of your program when it was last in that function.

The Variables panel displays a list of program variables that are visible in the scope of the stack point you are looking at. This includes local variables, static variables, and globals that are visible. If you want to watch one of the variables more closely, you can double-click on the variable name to view it in a separate window. You can also change the value of a variable by double-clicking on the existing value. This can be useful for experimenting with values of certain variables during live program execution.

The Source panel contains the source file for the section of code being run at the current stack point. At program start, this is PilotMain, but as you move through the execution of your program, this file can be whatever function and source file being run at a given time. A special arrow pointer shows you which line of code is being executed.

The next few sections take you through common debugging tasks such as setting breakpoints, stepping through source code, and debugging to the physical Palm device.

Setting Breakpoints

If you started stepping through the execution of your program line-by-line starting with PilotMain, you would be in for a long debugging session. Think of your program in the debugger as a book that holds the answers to a set of questions you have about how the program operates. Rather than read the book cover to cover, it would be much more efficient to use the table of contents and the index to jump directly to the areas that have the most promise of holding the answers to your questions.

Similarly, you should have a plan of attack before you start debugging. Think about the problem you are having and construct a set of logical theories as to why it might be happening. Although some errors are harder to diagnose than others, with experience, you will learn to be able to make good guesses as to the cause of the error and determine the most likely areas of your program that are at fault.

With a set of likely problem spots in mind, you can set a couple of lines of code where you want your program to stop and allow you to examine its state. To do this, you use CodeWarrior's *breakpoints*. Breakpoints can be set by clicking on the short horizontal line to the left of any line of code while in a debugging session (these lines are not available when you are not debugging). Note that you can set a breakpoint in any source code file, not just the one being viewed in the debugger window. When a breakpoint is set, the line to the left of the code changes to a red circle.

With breakpoints set, you can tell CodeWarrior to execute your program. When your program hits a line of code that contains a breakpoint, your program will "break" into the debugger at exactly that line of code, giving you the opportunity to view program variables and the stack, as well as to step line-by-line through the problem area.

When you no longer want to keep a breakpoint, you can click again on the red circle and the breakpoint will be cleared.

Stepping Through Code

Once you have your program running to the potentially troublesome part of the code, you can use a series of commands to navigate through the code and observe changes in variables that give you clues as to what is happening. These commands are available from the Debug menu as well as in a toolbar at the top of the debugger window:

- The Run command (the green arrow in the toolbar) tells CodeWarrior to run your program until it hits a breakpoint or you kill the program.
- The Kill command (the red x in the toolbar) immediately stops execution of your program and resets the emulator session.
- The Step Over command advances program execution by one line of source code in the current function.
- The Step Into command drills down inside of a function call that is at the current line of code.
- The Step Out command runs the program until the end of the current function scope.

Debugging to a Palm Device

Although it is far more convenient to debug into POSE, some problems can be diagnosed only by debugging your program when it's running on the device. If you find you need to debug on the device, you will need to change your Palm Connection settings in CodeWarrior to reflect the Device option, and also set the proper communications method and speed.

Starting a debugging session on the device requires you to put the Palm device in the cradle, choose Project, Debug, and then put your device into a special Console mode. Console mode allows your device to be used in a debugging session, and is entered by using a special Graffiti input sequence.

You will need to tap on the Find button and perform the “Shortcut dot-dot 2” sequence. This means you enter the special gesture for shortcut (which looks something like a script lowercase L), followed by a period gesture (two consecutive dot taps), and finally the gesture for the number 2. If you did it correctly, your Palm screen will go blank except for a blinking cursor. The sequence can be tricky to master, so keep practicing if you don’t get it right the first time.

Once you’ve successfully entered Console mode, the rest of your debugging session continues similarly to the section on debugging to POSE. The most noticeable difference will be the speed of the debugging session’s communication with the device—it is much slower and more awkward than debugging to POSE.

Other Debugging Tools

Aside from the CodeWarrior debugger, alternative debugging tools are available as part of other development environments, or as supplements to the code Palm OS development SDK. The primary alternative C/C++ development environments GCC and Falch.net come with different debugging tools. The GCC toolset (often obtained as part of PRC-Tools), comes with the GDB debugger, which will do the job although it is not as user friendly as the CodeWarrior debugger. The Falch.net IDE comes with its own source-level debugger, which continues to mature along with the rest of the Falch.net IDE, and as of this writing is quite nice in its own right.

Although the tools are somewhat different, the general approach to source-level debugging holds true for Palm developers. For more details on how to operate these debugging options, see the documentation that comes with each environment.

Using Gremlins to Find Problems

Gremlins are a special feature in POSE that allow you to stress-test your application in the emulator by throwing a multitude of simulated events called a “Gremlin Horde” at your application. In this way, your application goes through a kind of torture test until either it fails (by crashing or other error) or the horde has completed running.

To see how Gremlins work, take a very simple application (again the Hello example will work just fine) and load it into the Emulator. From POSE, bring up the menu and choose New Gremlin. You will be greeted by a dialog box that asks you to select the application(s) that should be the target of the simulated events. These will be the only applications that receive simulated stylus and key events.

In this case, if you chose the Hello application, Gremlins would besiege Hello with a horde of stylus and key events, even including taps on the silk-screen area (Find, Menu, and other windows that can be invoked through shortcuts such as the

Keyboard window). It will not however run the Launcher, Calculator, or other applications (unless you chose them in the New Gremlin Horde dialog box).

In addition to the target application(s), you can choose numeric ranges that will control the extent of the stress-testing your application will undergo. By default, a horde consists of 100 individual "gremlins" run in succession, although you can change or limit those if you want. Each gremlin, when it runs, will execute a certain sequence of events, randomly generated.

You can also specify in the Switch After field how many events POSE will generate for a given Gremlin before moving on to the next gremlin on the horde. The Stop After field specifies how many total events for each Gremlin will be run before the horde stops running. So if you specified a range of 100 Gremlins, and a Stop After value of 1000, you would have generated a grand total of 100,000 events before stopping. If at any time in the running of a Gremlin your program experiences an error, that Gremlin is flagged and is not rerun in the sequence.

After initiating a Gremlins session, the progress of the session is displayed in a Gremlin Control window. Progress can be manually stopped at any point by clicking on the Stop button that is displayed in the progress window. The session can at that point be abandoned (by clicking on the exit box) or resumed.

Gremlins can be set to save an emulator session periodically, which will give you a chance to reload a POSE session that represents your (stressed out) application, potentially yielding a more productive debugging session. You can also specify which types of calls should be logged to the Gremlins log file, under Logging Options. Although tedious, an iterative process of running Gremlins to higher and higher thresholds—paired with close examination of log files to determine when your program failed—can be quite useful in cleaning up your program code.

As with most debugging techniques, Gremlin results are easier to understand when there is less code, as well as (theoretically) fewer possible problem areas. Thus it is recommended that you incorporate the running of Gremlins against your code starting very early in the development process, so that errors can be caught and attributed to code as soon as possible after the code is added to the program. Deciding to run Gremlins after you've written 100,000 lines of code can still yield useful information, but it might be very difficult to disentangle which parts of your code are causing the fatal errors.

Summary

I've introduced a wealth of debugging options in this chapter. Combined with the last chapter on POSE, you are now armed with an excellent set of tools that can be used to ensure your program is as error-free as possible, and that any errors that do creep in are easily diagnosed and repaired.

If you are reading this book chapter by chapter, you might be getting antsy to start working with some real code. By placing this chapter near the beginning of this book, it was our hope that we could emphasize the importance of debugging. The compile-edit-debug cycle in Palm programming is so expensive that it is foolhardy not to take advantage of the help that is available.

5 Creating and Managing Resources

IN THIS CHAPTER

In Chapter 2, "Anatomy of a Palm Application," you created your first Palm program, "Hello Palm." You walked through all the pieces of code that were involved in creating the program. However, you skipped one essential component in developing "Hello Palm": creating the main form's visual appearance.

This chapter builds on the last one by explaining Palm OS resources. This chapter also explains how to use CodeWarrior's Constructor tool and the PRC-Tools component PilRC. Each of these tools offers the developer a way to design and layout the visual elements of a Palm program.

This chapter covers the following topics:

- What resources are and why they are used
- The CodeWarrior Constructor tool
- The PRC-Tools PilRC Resource tool
- How to create and manage resources in both a CodeWarrior project and a PRC-Tools/GCC project

This chapter is not an exhaustive tutorial or reference on using Constructor or PilRC (that's why they make user manuals!). Please refer to the CodeWarrior documentation for more in-depth descriptions of how to use the Constructor tool. Rather, this chapter focuses on describing what resources are, as well as giving practical information above and beyond what is found in the Constructor and PilRC documentation.

What Is a Palm Resource?

You can consider Palm resources special data structures, stored separately from your program code, which are included in your program's .PRC file by the CodeWarrior or GCC linker in your project's build. Although most often, we think of resources as relating to user-interface elements, Palm OS resources can be menus, forms, user-interface controls, character strings, string lists, application info, alerts, icons, or bitmaps.

Without resources, you would either have to hard-code these definitions in static or global program values, in which case you would need to recompile your application every time they changed, or you would store them in an external database, which creates programming and performance overhead. Resources allow you to change many characteristics of your program without having to recompile all of your code. They also encourage good separation of code and data using a standard format. Finally, because they are loaded at runtime by Palm OS, resources help to reduce the footprint and memory requirements of your application.

You will find that this book covers almost all the various resource types in one chapter or another as you encounter them.

Why Use Constructor?

In CodeWarrior, you define all user-interface elements using the visual Constructor resource-management tool. Right out of the box, Constructor is the only way to create and modify your resources so they are in the correct format expected by the CodeWarrior linker; the resource file generated by Constructor is in a proprietary binary format, so all changes must happen through Constructor.

As an alternative, a number of third-party utilities can help you manage your resources via text files. My personal preference (for what it's worth) is to keep the best of both worlds: have a visual tool for designing and prototyping, yet be able to browse and modify the actual resource tags in a text file as needed. If nothing else, the text-file version is more suitable for tracking changes in your program with version-control and source-code-management tools. This is where PRC-Tools and PilRC come in.

Why Use PilRC?

PilRC is a kind of script compiler for Palm resources. With PilRC, you create a text-based script with an .RCP extension. The source script lists a description of the various resource elements in your program. This script is then fed into PilRC, and the resulting output is created in binary form as an .RO file.

Although it would be natural to assume that you will be most productive by using the Constructor tool because it comes bundled with CodeWarrior, the advantages of using PilRC instead of Constructor are numerous.

As mentioned before, some developers are simply more comfortable with having a precise level of control over their code. With Constructor, you don't really enjoy that control. Although you can visually lie out your user interface, the final representation of your resources is totally controlled by Constructor.

Secondly, if you use version-control software (and if you don't I heartily recommend you start!), it is impossible to track changes over time to the Constructor-based binary representation of your user interface. With PilRC, your interface description is right there in text format, and you can easily perform different comparisons to determine exactly when and where changes took place.

Creating a Resource Project File

The next sections describe how to use both Constructor and PilRC to create a resource project file that describes the user interface for your Palm application.

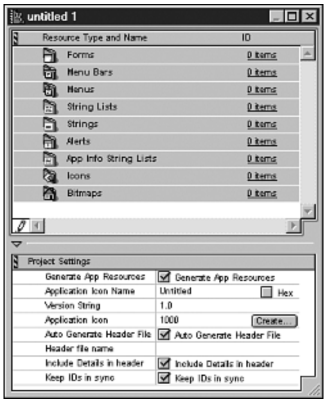
You'll learn how to lay out a simple form using each tool. This section also covers some nuances and considerations for dealing with Constructor.

Creating Resources in Constructor

Constructor creates a single resource file that contains all the resources you've defined in your application. As with much of the code created in "Hello Palm," I've found that some parts of a resource file are pretty much boilerplate and do not change from project to project. Just as it is unnecessary to create a new CodeWarrior project file from scratch for each new Palm project, it is also unnecessary to create a new resource file for each new project.

I walk you through the creation of a resource file, and as you go along, I note which settings you need to modify specifically for each project.

To create a resource file, launch Constructor, and from the main menu, choose File, New Project File. You should see a window similar to the one in Figure 5.1, which represents your project. At the bottom of the window are the global settings for the resource file, and at the top are sections representing each possible type of resource. This interface might seem unusual to those used to navigating project hierarchies in other tools, but over time, you will become accustomed to the somewhat nonstandard look and feel of Constructor.


untitled 1

Resource Type and Name	ID
Forms	0 items
Menu Bars	0 items
Menus	0 items
String Lists	0 items
Strings	0 items
Alerts	0 items
App Info String Lists	0 items
Icons	0 items
Bitmaps	0 items

Project Settings	
Generate App Resources	✓ Generate App Resources
Application Icon Name	Untitled ☐ Hex
Version String	1.0
Application Icon	1000 Create...
Auto Generate Header File	✓ Auto Generate Header File
Header file name	
Include Details in header	✓ Include Details in header
Keep IDs in sync	✓ Keep IDs in sync

FIGURE 5.1 Constructor's main project window.

A few global settings change from project to project. Specifically, you should

- Set your application's icon name from Untitled to the one you want the users to see in the Palm application list.
- Create an application icon by clicking the Create button, which invokes the icon editor.
- Make sure that Autogenerate Header File is checked.
- Specify a header filename into which Constructor should place the definitions for the IDs associated with your resources.

After these project settings are complete, you can save your resource file. You should save it to the same location as your other source files.

What's a Resource Fork?

Saving your resource file causes a couple of interesting things to happen. First, a header file is created with the name you chose in the Header Filename setting. Given that you haven't yet created any individual resources, this header will be mostly empty, but eventually, all your resource IDs will be defined here. Be warned that you should not attempt to modify your resource definitions by directly editing this file! Changing the header definitions does not cause any changes to the corresponding binary resources in the resource file, so things will be out of sync if you make edits here. Also, any changes you make in the header will be overwritten the next time you save your resource in Constructor. If you need to make changes to your resources, relaunch Constructor and make the changes there.

The second thing that occurs is that your resource file is saved with a .RSRC extension. This is not terribly exciting until you notice (if you are working under Windows) that a new subdirectory called resource.frk is also created and it appears to contain a copy of your resource file.

To me, a long-time Windows developer, this was hopelessly confusing. Which file contained my resources? Which one should I add to my project file? Which one should I double-click to edit my resources? As it turns out, this is an attempt by CodeWarrior to mimic the storage behavior of Constructor under the Macintosh OS and to allow the transparent transfer of resources between Mac and Windows development environments. What CodeWarrior does is create a two-part file: a data fork and a resource fork. The resource fork, which actually contains your resources, will always be stored in the resource.frk subdirectory. Even though it shows up as a 0-byte length file, you can include the data fork file in your project and even re-edit your resources in Constructor by double-clicking the data fork file. The resource fork file will automatically be loaded.

Version-Control Considerations in Constructor

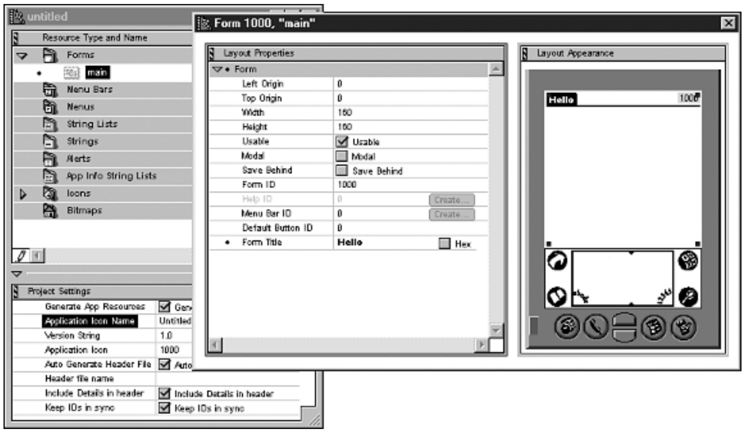
My company uses a version-control system, and I've noticed that CodeWarrior in some respects does not coexist peacefully with it:

- Most version-control systems will set the read-only attribute of a file after you check a file in to the depot. Unfortunately, Constructor and the IDE expect both resource files and project (.MCP) files to be read-write. In all fairness, this problem is common to other software development environments. A workaround is to always check out the project and resource files for editing prior to building or running Constructor.
- The dual nature of resources causes headaches in that you must remember to check out and check in both files to the version-control system.

A Constructor Example: "Hello Palm" Revisited

Reopen the resource file you created earlier, and design the main form for "Hello Palm." (For an exhaustive treatment of how to define various types of forms in Constructor, see Chapter 6, "Interacting with the User: Forms.") You can open an existing resource file either via the File, Open command in Constructor or by double-clicking the data fork resource file (.RSRC).

To add a form, highlight the Forms section in the top half of the Constructor main project window, and choose Edit, New Form Resource from the Constructor main menu. Click once on the Untitled label, and give your form a name of Main. Press Enter when you are done. Now, you can invoke the Form Layout window by double-clicking the form. The Form Layout window is pictured in Figure 5.2.



Project Settings

Generate App Resources	✓ Gen
Application Icon Name	Untitled
Version String	1.0
Application Icon	1000
Auto Generate Header File	✓ Auto
Header file name	
Include Details in header	✓ Include Details in header
Keep IDs in sync	✓ Keep IDs in sync

Form

Left Origin	0
Top Origin	0
Width	160
Height	160
Usable	✓ Usable
Modal	☐ Modal
Save Behind	☐ Save Behind
Form ID	1000
Help ID	0
Menu Bar ID	0
Default Button ID	0
Form Title	Hello ☐ Hex

FIGURE 5.2 The (initially empty) main Form Layout window.

On the left are all the properties you can set for the form. On the right is a WYSIWYG representation of what your form will look like on the Palm. For main forms, about the only property you'll need to change from the default is the Form Title, which you can set to "Hello."

You add user-interface elements, such as buttons and labels, to a form by dragging them from the Catalog window. The Catalog window is not initially visible; you have to load it by choosing Window, Catalog from the Constructor main menu.

Once the Catalog window is loaded, you can drag any supported user-interface resource type onto the form. For this simple form, you only need a label. Click the Label resource type, and drag it to the center of the form. A new form object of type `Label` is created, and the current properties pane changes from that of the form to that of the new label object. Because of the modest needs of this program, simply change the label `Text` property to read "Hello Palm." You might need to resize the label so that all the text shows.

At this point, you can save your work. (I've found it a good idea to save my work often during long Constructor sessions; there are few things more heartbreaking than experiencing a system crash after getting 25 teeny-weeny user-interface elements positioned just right on a form!) Return to the main CodeWarrior IDE to build your .PRC file, which will contain your new resource definitions.

A PilRC Example: "Hello Palm" Revisited

As I described earlier, PilRC is a text-based utility that converts text representations of your Palm resources into binary files that can be linked into your Palm application using PRC-Tools.

Let's take the same "Hello Palm" example, and create the same resources using PilRC-compatible script commands.

The PilRC script specification supports a rich set of user interface elements and other resource types. For this simple example, you will need to define just two elements: a main form for your application, and a label that says "Hello Palm."

The following PilRC script creates a set of resources equivalent to those created in the previous section with Constructor. (This script is stored in a file called hello.rcp.)

```
#include resource.h
FORM ID MainForm AT (0 0 160 160) NOSAVEBEHIND
BEGIN
  TITLE "Hello"
  LABEL "Hello Palm" AUTOID AT (53 77)
END
```

Starting at the top of a PilRC .RCP file is an `include` statement that references a resource header file. When using PilRC, you should not directly embed resource IDs in your .RCP file. The reason for this is that your program's source files need to reference your resources by name, and it is easier to define associations between resource names and IDs in a separate resource.h file that can be included in both your .RCP and .CPP files.

This example uses an ID for the form called `MainForm`. In the resource.h file, you will then add a statement as follows:

```
#define MainForm 1000
```

This assigns the form a name of `MainForm` and an ID of `1000`, which is the same as what Constructor did.

For the label, you use an attribute of `AUTOID`. `AUTOID` allows PilRC to automatically create a unique ID on its own, which is handy when you are dealing with form elements that do not require an assigned ID, such as the label on the "Hello" form.

If you wanted to add other user interface elements to the form, or other types of resources to the program, you would simply look up the syntax in the PilRC documentation, and add them to both the .RCP and resource.h files.

Summary

This concludes the introductory section of Palm Programming. The first five chapters have

- Described the Palm Computing platform and the tools involved in creating a Palm program
- Created the structure and code components required for most standard Palm applications
- Discussed the debugging environment and strategies for determining the cause of coding errors
- Examined the CodeWarrior toolset, including the IDE and Constructor
- Described the most popular “free” alternative PRC-Tools, including PilRC
- Created the first Palm program, “Hello Palm”

At this point, you should be ready to dive into more specific subject matter. Where should you go from here? I recommend covering at least the first few chapters of the user-interface section, especially Chapters 7, 8, and 9. These chapters will give you some experience in creating more complex user interfaces, experience you can then apply as you learn about Palm databases, communications, and more advanced topics.

For the remainder of the chapters that have projects containing resources, I describe Constructor as the tool being used (to the extent that it is necessary). The example projects include PilRC resources as well as Constructor resources, as part of both GCC and CodeWarrior implementations.

PART II
Programming the Palm User Interface

IN THIS PART

6 Interacting with the User: Forms

IN THIS CHAPTER

You can implement the most efficient algorithms, design the most elegant program architecture, or design the most flexible data storage system in the world. But if your users cannot quickly and easily learn how to effectively exploit those features, it is likely that they will discard your application in favor of one that is easier to use.

Creating an attractive, easy-to-use application that strikes just the right balance between providing all the features that customers want and maintaining an application's focus remains a difficult challenge. This is an especially daunting problem on handheld platforms such as the Palm, where the immediate temptation is to replicate the breadth of functionality and complex user interfaces found in desktop applications.

This chapter provides an introduction to programming the Palm user interface by presenting the most widely used building block in a Palm application's user interface: the form. You'll learn about the unique problems and constraints associated with Palm user-interface design and implementation. More specifically, you'll become familiar with how you can use the Palm form APIs to shape your application's interface. By the end of the chapter, you will have used the Palm form APIs along with the CodeWarrior Constructor tool to code several examples that use forms: a message box, an "about" box, and a rudimentary shopping-list form.

In this chapter, you'll learn

- The unique characteristics of the Palm user-interface model
- The various types of forms and how they are applied in the Palm user interface
- How to use Palm development tools and the Palm APIs to program forms

The Palm User Interface

The Palm user interface is different from the one you might be used to working with on other computer platforms. Perhaps the most obvious difference is its size: The standard display dimensions today are 160 pixels wide by 160 pixels high. Several newer device models now sport higher resolution screens, but even those screen dimensions are smaller than the most basic VGA monitors running at 640x480. As a developer, you must assume and even embrace this small form factor when designing your applications.

Aside from screen size, there are some other differences, including

- Only one application is visible at a time.
- Application forms are not resizable or movable.
- Most devices today do not support color.

Application developers must learn to live within these restrictions and design their user interfaces to work well on the Palm device.

It is also important for developers to understand that users expect the Palm to provide information almost instantaneously. Keep in mind the following tenets:

- Place the most-often-used information on the main screen.
- Relegate lesser-used information to secondary screens and pop-up screens.
- Provide fast and easy access to the most commonly needed information, with the minimum pen strokes required.
- Minimize the number of screens the users must navigate in your application.
- Avoid placing the users in difficult-to-escape modes, such as nested dialogs.
- Do not make the users scroll through more than one or two screens to locate information.
- Consider moving complex or infrequently accessed data to the desktop and using conduits to download the information to the Palm.

For further guidelines, I encourage you to read the relevant sections in the Palm SDK documentation (available at `http://www.palmos.com/dev/support/docs/ui/UIGuidelinesTOC.html`); there are some very good suggestions.

What Is a Form?

A form can be defined as a visual container for user-interface elements. Some of the user-interface elements commonly found on forms are labels, buttons, lists, and check boxes. For those familiar with user interfaces on other computer platforms such as Microsoft Windows, the most obvious parallel to a form is a window or a dialog box.

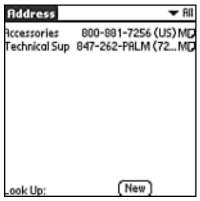
Palm applications generally have one or more forms, which together compose the user interface for the application. Forms are either full-screen or pop-up. Full-screen forms are sometimes called *views* because they usually provide different logical views of the data associated with the application. A view is similar to the concept of a window on other platforms, except that it cannot be moved or sized and can only appear one at a time. Views are said to be *modeless* because they do not force the users to complete a specific action in order to perform other application functions. Every application has at least one full-screen form: its main form.

Pop-up forms are closely related to the concept of *modal* dialogs. They occupy a portion of the screen and "pop up" over the currently active form. When an application displays a pop-up form, the users can resume using the rest of the application only after dismissing the pop-up form. Pop-up forms are often referred to as *dialogs* in the Palm SDK.

The Palm SDK has much to say about the standards and guidelines that application developers should follow in the process of designing their forms. Pop-up forms in particular should always be designed to fill the entire width of the screen and are justified to the bottom of the screen so that the top portion of the current form is still visible to the users.

Examples of Forms in the Built-in Palm Applications

Figures 6.1, 6.2, and 6.3 show real-life examples of forms in the built-in Palm Address Book application.



Accessories 800-881-7256 (US) MD
Technical Sup 847-262-PALM (72... MD

FIGURE 6.1 The main form of the Address Book application.

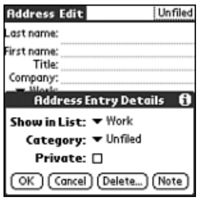
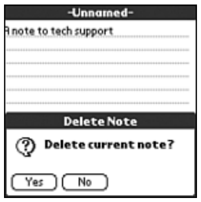


FIGURE 6.2 The Address Entry Details pop-up form.



A note to tech support

Delete current note?

FIGURE 6.3 An alert confirming the deletion of a note entry.

Programming with Forms

Forms come in several flavors, ranging from simple message box types to full-screen user interfaces for your application. The Palm SDK supports the following types of forms:

- Alerts
- Simple pop-up forms
- Complex pop-up forms
- Full-screen forms

The following sections explore these types of forms in depth.

Alerts

The simplest form of all is called an *alert*. Alerts are the closest thing Palm has to message boxes in that their general appearance and functionality is already handled for you. On the downside, for each type of message you want to display, you must still use Constructor to create an alert resource. Unfortunately, there is no "standard" message box API that predefines a common resource (although you'll create one later in this chapter).

There are two kinds of alerts. The simplest is created entirely in Constructor and has fixed-text content that is hard-coded into the resource: If you want to change the text, you have to change the resource in Constructor and rebuild your application.

In the next section, you'll use the simplest type of alert to display a pop-up message on the Palm. As with many Palm programming tasks, this one has two components: using Constructor to create the resource and adding code to interact with the resource in your application.

Using Constructor to Create a Simple Alert Resource

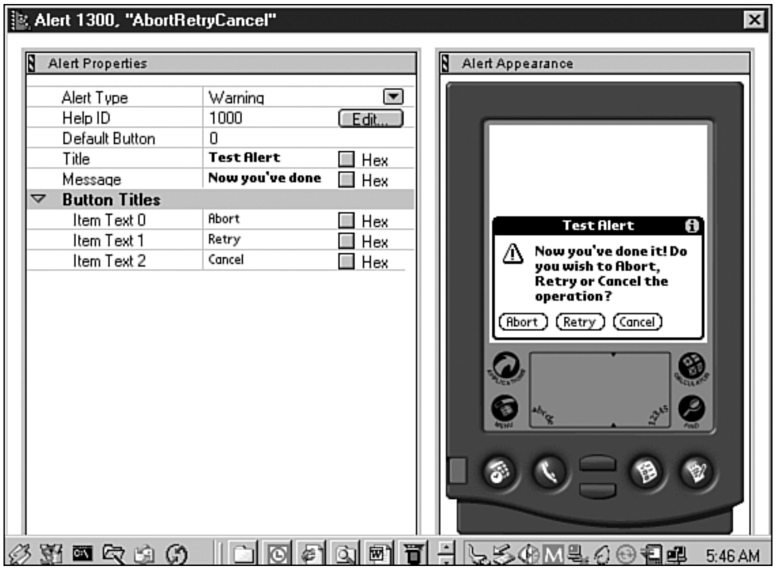
As you saw in Chapter 5, "Creating and Managing Resources," CodeWarrior comes with a visual resource editor called Constructor. You use Constructor for creating any kind of form resource; it provides a quick way to manage and navigate all of your project's resources.

To create an alert in Constructor, follow these steps:

1. Open the main project window for your resources.
2. Select the Alerts resource type from the project window.
3. Add a new Alert resource by choosing Edit, New Alert Resource or pressing Ctrl+K.

   A new, untitled alert resource appears under the Alerts section. Note the ID of the Alert: You'll be using it soon.
4. Rename the new Alert resource from the default `<untitled>` to some name that is meaningful.
5. Double-click the new resource to open the Alert Properties window. (See Figure 6.4.)

   The Alert Properties Window appears with two panes: a property sheet on the left and a visual representation of how your new alert will appear on the Palm display. Note that the alert comes with some default properties, such as a title, message text, and OK button already filled in with placeholder values.


Alert 1300, "AbortRetryCancel"

Alert Properties

Alert Type	Warning	
Help ID	1000	Edit...
Default Button	0	
Title	Test Alert	Hex
Message	Now you've done	Hex
Button Titles		
Item Text 0	Abort	Hex
Item Text 1	Retry	Hex
Item Text 2	Cancel	Hex

Alert Appearance

Test Alert

Now you've done it! Do you wish to Abort, Retry or Cancel the operation?

FIGURE 6.4 Creating an alert resource in Constructor.

6. Enter each of the Alert Properties with the values described in Table 6.1.

TABLE 6.1 Alert Resource Properties

Property	Meaning
Alert Type	Choose from a set of available alert types: Information, Confirmation, Warning, and Error.
Help ID	The ID of an associate string resource containing help text. Palm recommends that each alert or form contain help text that is displayed when the users tap the "I" on the form. To define help text, create a new string resource from the main project window and type the text that should be associated with the alert or form. The ID of the string resource is the value that you should then set in the Help ID property.
Default Button	If there is only one button on the alert, this property is grayed out. If there are two or more buttons defined, set this property to the item ID of the button that should be the default.
Title	The text that should appear in the title bar of the alert.
Message	The text that should appear in the client area of the alert.
Button Titles	For each button defined, enter the label shown on the surface of the button. Note that if you want to remove a button from an alert, highlight the button in the properties pane and choose Edit, Clear Button Text from the main Constructor menu.

When you are happy with the appearance of your alert resource, close the Alert Properties Window and choose File, Save from the main menu.

Adding Alert Handling to Your Application

Now that you've specified the appearance of your alert in Constructor, it is time to see it in action. Just as you would hope, it's easy: To display your alert in your application, all you need to do is add the following code:

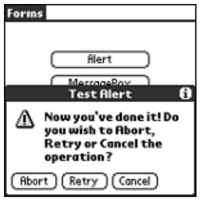
```
UInt16 uiButton = FrmAlert (MyAlertID);
```

MyAlertID is the ID of the alert. The return code is the "item number" of the button the user tapped to dismiss the alert. Don't confuse the item number with an ID of a button. The item number starts at zero for the first button, and if there are other buttons on the alert they are numbered sequentially after that. Checking item numbers is really only important if there are multiple buttons on the alert (for example, a Yes/No choice).

Let's look at an example of handling an alert with three buttons—Abort, Retry, and Cancel:

```
UInt16 uiButton;
uiButton = FrmAlert (AbortRetryCancelAlert);
switch (uiButton)
{
   case 0: // Abort
   case 1: // Retry
   case 2: // Cancel
}
```

Figure 6.5 shows what this simple alert looks like when called from an application.



Now you've done it! Do you wish to Abort, Retry or Cancel the operation?

FIGURE 6.5 The Abort, Retry, Cancel alert in action.

Custom Alerts

If you want to be able to customize the text of an alert during your program's runtime (for error messages, debugging purposes, and so on), you need to use a custom alert. You create custom alerts in the same way as regular alerts, but the item text can contain up to three special embedded replaceable characters, `^1`, `^2`, and `^3`, which correspond to three parameters in the `FrmCustomAlert` API.

When your program runs, you pass the values you want to display in place of these characters, as in the following example:

```
err = FrmCustomAlert ( MyAlertID, "Error 21", "No item
➥  selected", "");
```

Note that the third parameter is an empty string; use an empty string if your alert resource does not define all three replaceable characters or if you simply do not need to supply text. The replaceable character will be changed to an empty character string in the alert display.

This might sound like an unreliable way to provide a generic message box capability in your application—I agree—you're going to remedy that situation in the next section.

`PalmMessageBox`: A Reusable Message Box API

The basic flaw in using `FrmCustomAlert` to implement message box functionality in your application is that when you call the `FrmCustomAlert` function, you are trusting that the alert resource matches your call in terms of the number and meaning of the replaceable parameters. Imagine an alert resource defined with alert message text as follows:

```
"Warning: the database named ^1 already has ^2 records in it with
➥ that field value"
```

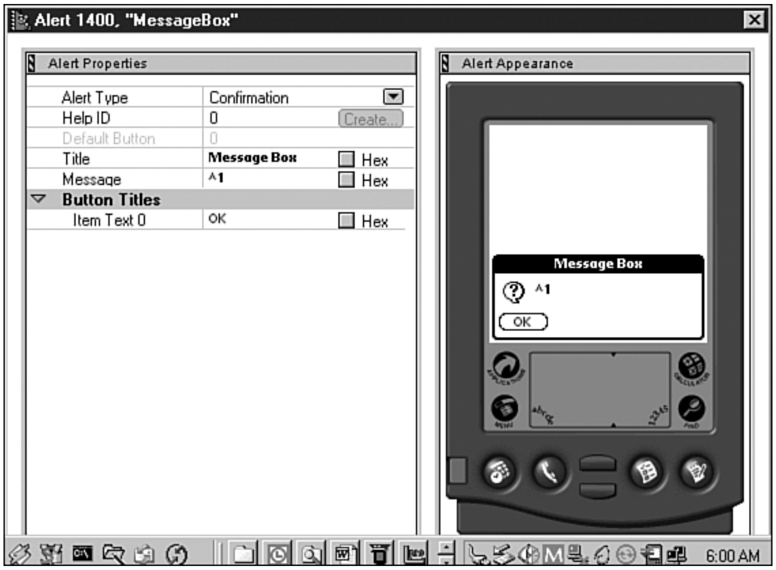
Unless you are very familiar with this resource, you could easily misuse the alert by making the following call:

```
FrmCustomAlert (MyAlertID, "Error", "Too many records", "Abort");
```

As you can see, the text passed in the first two parameters is inappropriate in the context of the alert resource. What's more, the third parameter value will never be seen by the users: The designer of the alert resource did not define a replaceable parameter with `^3`. Unfortunately, this is not safe and is unnecessarily error-prone, especially in an application with a significant number of alerts.

What is needed here is a simple, reliable general-purpose message box API that lets you define its message text at runtime according to program conditions. Let's create one:

1. Define a new alert in Constructor with a title of `"Information"` and a message containing a single replaceable parameter `^1`. See Figure 6.6 to see how the alert appears in Constructor.


Alert 1400, "MessageBox"

Alert Properties

Alert Type	Confirmation	
Help ID	0	Create...
Default Button	0	
Title	Message Box	☐ Hex
Message	^1	☐ Hex
Button Titles		
Item Text 0	OK	☐ Hex

Alert Appearance

Message Box

^1

OK

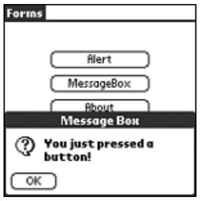
FIGURE 6.6 Creating a custom alert with replaceable parameters in Constructor.

2. Create a function as follows:

```
Err PalmMessageBox (Word wAlertID, CharPtr pMessage)
{
   Err err;
   err = FrmCustomAlert (wAlertID, pMessage, "", "" );
   return err;
}
```

That's all there is to it: You now have a general-purpose message box API that you can use in a variety of program situations and to which you can always be assured is passing the right parameters. With a little extra work, you could predefine a series of these message box APIs, one for each type of alert. You might even remove the need to pass the alert ID, instead simply asking the callers to select what type of alert they want from a number of predetermined types.

Look at Figure 6.7 to see the message box function in action.



You just pressed a button!

FIGURE 6.7 The general-purpose message box alert.

Custom Forms

Obviously, alerts are somewhat limited in functionality and appearance: They may only contain a single text string and one or more buttons. For more complex user interactions, you can instead use Palm's Form Resource.

Although forms are more flexible and support a rich set of programming interfaces, you add forms to an application in a similar fashion as adding alerts. You will follow the two-step sequence established in the previous section, first defining a form resource in Constructor and then adding code to load and interact with the form in the application.

You'll start by adding a simple pop-up About box to the program. You'll then move on to a more complex example that begins to explore the power of the form API.

Using Constructor to Create a Form Resource

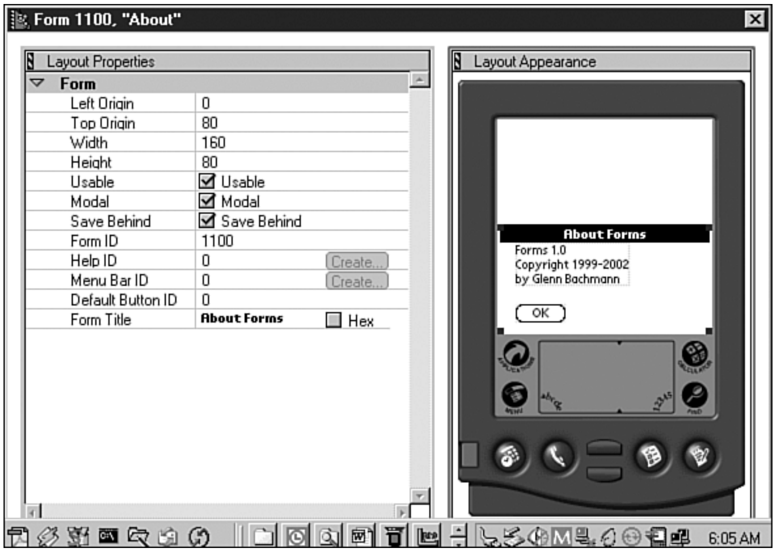
Use the following steps to create a form in Constructor:

1. Open the main project window for your resources.
2. Select the Forms resource type from the project window.
3. Add a new form resource by choosing Edit, New Form Resource or pressing Ctrl+K.

   A new, untitled form resource appears under the Forms section. Once again, note the ID of the Form: You'll be using it in your sample code later in this section.
4. Rename the new Form resource from the default `<untitled>` to "About."
5. Double-click the new resource to open the Form Properties window. (See Figure 6.8.)

   The Form Properties Window appears with two panes: a property sheet on the left and a visual representation of how your new form will appear on the Palm display. You will immediately notice that you have many options available in defining the appearance and behavior of your form.

6. Enter each of the form properties with the values described in Table 6.2.



Form

Left Origin	0
Top Origin	80
Width	160
Height	80
Usable	☑ Usable
Modal	☑ Modal
Save Behind	☑ Save Behind
Form ID	1100
Help ID	0 Create...
Menu Bar ID	0 Create...
Default Button ID	0
Form Title	**About Forms** ☐ Hex

About Forms
Forms 1.0
Copyright 1999-2002
by Glenn Bachmann
OK

FIGURE 6.8 The About box form in Constructor.

TABLE 6.2 Form Resource Properties

Property	Meaning
Left Origin	Offset of the form, in pixels, from the left edge of the display.
Top Origin	Offset of the form, in pixels, from the top edge of the display.
Width	Width of the form, in pixels.
Height	Height of the form, in pixels.
Usable	The 3.0 Palm documentation states that this property is not currently supported.
Modal	Check this if you are defining a modal dialog. If it is checked, the operating system will refuse to process pen events that occur outside the form, effectively forcing the users to complete their interaction with the form before moving on to other program functionality.
Save Behind	Also checked generally for modal dialogs. If checked, the system will save the value of the pixels that are covered by the form and automatically restore them when the form is deleted.
Form ID	The unique ID of the form. This property is automatically set by Constructor but you can change it if you want.
Help ID	Refer to Help ID in Table 6.1.

TABLE 6.2 Continued

Property	Meaning
Menu Bar ID	You can enter the ID of a menu bar resource here to associate a menu with the form.
Default Button ID	The ID of the button that you should assume is selected if the users switch to another application while the form is active.
Form Title	The title text to be displayed on the form.

Because this will be a modal pop-up form, you will need to make some adjustments to these properties:

1. Resize the form to take up the bottom one half of the screen. You do this by changing Left Origin, Top Origin, Width, and Height to 0, 80, 160, and 80. You can also use your mouse to directly drag the upper boundary of the form down by 80 pixels. In many cases, I find it to be more accurate to directly modify the numeric coordinates in the properties pane.
2. Check the Modal property.
3. Check the Save Behind property.

Note that you should check both the Modal and Save Behind properties for all modal dialogs.

Adding Elements to the Form

For the new form to look like the one in Figure 6.8, you need to add some form elements. Specifically, you'll need an OK button and a label.

Constructor has a special window called the Catalog, which presents a palette from which you can add any supported element type to your form using drag-and-drop.

To use Constructor to add the missing elements to your form, follow these steps:

1. If the Catalog window is not showing, choose Window, Catalog from the menu or press Ctrl+Y.
2. Click the Button type and drag it onto your form. The main Constructor project window changes to display the properties of the new button. For now, leave the properties at their defaults. (I discuss button resources and their properties in more detail in Chapter 8, "Button Controls.")
3. Drag a label type onto the form. Position it in the upper part of the form and widen it to almost fill the width of the form. In the Properties pane, set the text to reflect your application's name and copyright information.

4. Save your form by closing the form properties window and choosing File, Save from the main menu. Your form resource is ready for integration into your application.

The Resource Header File

Before you integrate the new form into the program, let's take a short detour, peek under the covers, and see what is happening as you add resources to the project.

Two important files are created and updated during a Constructor session:

- A header file containing the IDs of each resource defined in the resource file. This file has an .H extension and is located in your project's SRC directory.
- The resource file itself. This file has a .RSRC extension.

CodeWarrior has some odd legacy behavior that results in two .RSRC files being created: one in the SRC directory of your project and one in a directory called RESOURCE.FRK off your SRC directory. You will notice that the copy in the SRC directory is size 0 bytes. The real resources are stored in the RESOURCE.FRK directory, but you should not mess with the size 0 bytes file. This seemingly bizarre behavior has its roots in CodeWarrior's Macintosh beginnings, and Palm, in cooperation with Metrowerks, has stated its intention to remove this behavior in a future version of CodeWarrior.

CAUTION

Do not directly edit the resource header file. It is regenerated by Constructor after every resource-editing session, so you will lose any changes you make. Use Constructor to change resource IDs or define new resources.

Adding Form Handling to Your Application

If you look through the Palm SDK documentation, you will see a fairly large and sometimes confusing set of APIs for handling forms. There are more than 50 APIs dealing with various aspects of form handling. Most of these are related to either interacting with form elements or supporting behaviors associated with modeless, full-screen forms.

Modal, pop-up dialogs, involve three main functions:

- `FrmInitForm`
- `FrmDoDialog`
- `FrmDeleteForm`

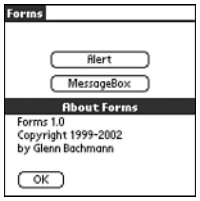
Here's how you use them to display the new form resource:

```
/*
   AboutFormShow ():
   Show the about dialog box.
   Parms:   none
   Return:  none
*/
void AboutFormShow ()
{
   FormPtr  pForm = FrmInitForm (AboutForm);
   FrmDoDialog (pForm);
   FrmDeleteForm (pForm);
}
```

Outside of using an alert, this is the simplest way to invoke a form. You pass the ID of the form resource to `FrmInitForm`, which returns a form pointer. If you pass the form pointer to `FrmDoDialog`, the API does not return until the user clicks a button in the form. The ID of the button is returned to the caller as a result code. You are then responsible for deleting the form.

Note that if necessary, you should perform any initialization of form elements before calling `FrmDoDialog`. Likewise, any querying of the state of form elements must occur prior to calling `FrmDeleteForm`.

Figure 6.9 illustrates the About box form.



Forms 1.0
Copyright 1999-2002
by Glenn Bachmann

FIGURE 6.9 The About box form in action.

Using Forms to Capture User Input

Now that you've seen how to initialize a form resource, display it as a modal dialog, and properly destroy it, let's look at how you capture user-entered values from form elements after the form is dismissed.

As an example, let's create a "shopping-list" screen that asks the users to check off which items they will need at the grocery store. Note that although you will use check box form elements in this example, I cover check boxes only to the extent necessary for this example. Chapter 8, "Button Controls," covers check boxes in depth.

Use Constructor to create another form called "Create Shopping List," and configure it so it resembles Figure 6.5. This time, carefully pay attention to giving your form elements IDs that reflect the elements' purposes. For example, change the ID of the OK button to "OK." Add a second button at the bottom of the form, and give it a label of "Cancel." Finally, add two check box field resources with labels "Milk" and "Cookies." After you are satisfied with the layout of your form, save it.

Remember when you took a peek at the resource header file that Constructor created for the resources? Here's where it becomes important: Constructor takes some liberties in supplying IDs for your form elements. Basically, it takes your supplied ID (such as "OK") and appends the resource type. The ID for the OK button winds up `OKButton`, the ID for the Milk check box field winds up `MilkCheckbox`, and so on. These are the IDs that you will need to use to obtain element values from your form.

TIP

If you are consistent in your naming system for your form elements, it will become easy to guess the resulting ID as Constructor defines it. Constructor's insistence on messing with the name is just another odd behavior that I expect will be eliminated as the Palm tool set matures.

Now, let's add some code to capture the user's shopping-list selections:

```
/*
   CreateShoppingList ():
   Displays the Create Shopping List dialog, allowing the user to select
   items to add to the shopping list.
   Parms:   none
   Return:  none
*/
void CreateShoppingList ()
{
   Boolean bGetMilk;
   Boolean bGetCookies;
   // Initialize the form
   FormPtr  pForm = FrmInitForm (CreateShoppingListForm);
   // Preset the check boxes for all items to be unchecked
   // Get the index of the check box form object
   Int16 wIDMilk = FrmGetObjectIndex (pForm,MilkCheckbox );
   // Given the object index, get a pointer to the control object
   // for the check box
   ControlPtr pCtlMilk = (ControlPtr) FrmGetObjectPtr (pForm, wIDMilk);
   // Use the object pointer, set the check box value to 0 (False)
   CtlSetValue ( pCtlMilk, 0 );
   ...
   // Display the dialog
   FrmDoDialog (pForm);
   // Get the values from the check box
   bGetMilk = CtlGetValue ( pCtlMilk );
// Destroy the form
   FrmDeleteForm (pForm);
}
```

This code uses some new APIs to initialize and query the form elements. I explore in depth the APIs that provide interaction with the various types of form elements as you dive into the details of each resource type. For now, it is sufficient to understand that given a form pointer, you can obtain a pointer to any given form element by supplying its resource ID. Once you have a pointer to a form element, you can use Palm SDK APIs to set and query the element's value.

More Complex Forms: Event Handlers

The methods used so far in form handling hide the complexity of what happens inside the form. The caller only knows what button was pressed and the state of the various controls. This capability satisfies the needs of a great many forms.

Some forms require some custom handling of events while the form is still visible on the screen. Perhaps the form supports a pop-up calendar date-picker. Maybe the form's display needs to change based on some user action such as a radio button selection. Each of these situations requires handling events that occur inside the form and taking action on the event.

To hook into the flow of events that are happening inside a form, a programmer must create an event handler for the form. If you have programmed for Windows, you can consider the form event handler a kind of dialog box or window procedure. It is a callback function that gets notified upon any significant event in the life of a form.

Continuing with the shopping-list example, let's add the ability for users to select a date when the shopping list will be used. The standard way to handle date selection in a Palm application's user interface is to use a push button as a launching point into the Palm SDK's `SelectDay` function. Upon return, the push button's label is set to the selected date. Unless you somehow trap the push button event, you have no way of knowing when to show the calendar date-picker dialog. With the addition of an event handler, the task becomes straightforward.

The following is a minimal form event-handler procedure:

```
Boolean MyEventHandler (EventPtr pEvent)
{
   Boolean  bHandled = false;
   return bHandled;
}
```

As I've said, an event handler receives notification of any significant event in the life of a form. It receives this notification by way of the `EventPtr` parameter. For most forms, you are only interested in one kind of event: `ctlSelectEvent`. This event indicates that a control on the form has been tapped or selected or has somehow changed state. A form's event handler receives this notification and has an opportunity to take action on the event. If the application chooses to process the event, it should return `TRUE` in the event handler.

Let's modify the event-handler callback to handle the case in which a new push button `DateButton` is tapped:

```
Boolean CreateShoppingList_EventHandler (EventPtr pEvent)
{
   Boolean  bHandled = false;
   switch (pEvent->eType)
   {
      case ctlSelectEvent:
      {
         Int16 wCmd = pEvent->date.ctlSelect.controlID;
         // A control button was pressed.
         switch (wCmd)
         {
             case DateButton:
            {
              Int16 mth, day, year;
              if ( SelectDay ( selectDayByDay, &mth, &day, &year,
              ➥ "Select a Shopping Day") )
             {
                 FormPtr pForm = FrmGetActiveForm ();
                 Int16 wIDDate = FrmGetObjectIndex (pForm,DateButton );
                // Given the object index, get a pointer to the
                // control object
                ControlPtr pDate = (ControlPtr) FrmGetObjectPtr
                ➥ (pForm, wIDDate);
               char szDate[20];
               CStrPrintf ( szDate, "%d/%d/%d", mth, day, year );
               CtlSetLabel ( pDate, szDate );
         }

         bHandled = true;
         break;
      }
    return bHandled;
}
```

If you examine this code, you see that when you receive notification that `CreateShoppingListDateButton` was tapped, you call the standard `SelectDay` function, which pops up a familiar "date-picker" form. Upon return, you take the month, day, and year of the selected date, format it, and set it as the new label of your push button.

The final step in adding an event handler to your form is to tell the Palm OS that it should associate your event handler with your form. It's a single function call: `FrmSetEventHandler()`.

Now, let's hook the event handler into the `CreateShoppingList` function:

```
// As usual, we initialize the form
FormPtr  pForm = FrmInitForm (CreateShoppingListForm);

// We set an event handler for the form
FrmSetEventHandler (pForm, CreateShoppingList_EventHandler );

// Launch the dialog
FrmDoDialog (pForm);

// Clean up
FrmDeleteForm (pForm);
```

After you have entered the code, the form should look like Figure 6.10.

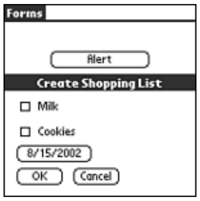


FIGURE 6.10 The nutritionally challenged shopping-list form.

Summary

You've seen that the Palm display imposes some significant and unique challenges to a user-interface designer. It is especially important to become familiar with the various types and proper usage of forms, the most common user-interface element on the Palm device. Forms provide a natural framework for your application and guide the users through the application's functionality.

You should now be comfortable with creating basic forms using CodeWarrior's Constructor tool and integrating them into your application. You should also be thinking about how to structure your application so that it uses forms to present functionality in a logical fashion, providing maximum ease of use and utility to the users.

The next few chapters build on the concepts presented here, expanding the use of forms and exploring the other user-interface elements available to the Palm developer.

7 Form Elements

IN THIS CHAPTER

- What Kinds of Form Elements Are Available?
- Custom Gadgets

In Chapter 6, "Interacting with the User: Forms," I introduced the concept of forms, one of the most basic elements of a Palm application. Forms enable your application to communicate information to the users. Forms also allow the users to navigate through your application and input or change your application's data. Forms can, in many ways, be considered the backbone of your application—a framework of screens that gives your application its personality. I cannot over-emphasize this point: How you design and implement forms in a Palm program can be the difference between a successful application and a failure.

Although a form can be considered analogous to the painter's canvas, it would indeed be a dull, blank canvas were it not for the availability of *form elements*. Form elements (sometimes referred to as *controls* or *widgets* on other platforms) are small user-interface objects such as buttons, lists, and editable fields that perform a specific job when placed on a form. When one or more form elements are placed on a form, and combined with some program code to handle the user's interaction with the form, something magical happens. Your program can interact with the Palm users!

In order to successfully design your program's user interface, you will need to achieve an in-depth understanding of the form elements that are available to you as a Palm developer. This means not only understanding how to write code that incorporates them into your program, but also gaining a feel for what types of form elements are most appropriately used when faced with different user-interface requirements.

Although each chapter in this book is designed to be self-contained, in order to gain a complete understanding of forms and form elements it is a good idea to read and understand the material in Chapters 7 through 12 before starting to design a user interface for your program. This chapter provides an introduction to the various form elements available to you as a Palm developer. Chapters 9 through 12 then explore each type of form element in more depth, and provide example programs that illustrate their use.

What Kinds of Form Elements Are Available?

The primary form elements that are available to the Palm developer are as follows:

- Buttons
- Labels
- Input fields
- Lists
- Triggers
- Tables

The next few sections provide an overview on each of these elements, and discuss appropriate uses for them.

Buttons

The term "button" is somewhat overloaded in the Palm world. In the Palm SDK, buttons don't just refer to the standard push button that we see on just about every program's user interface. The button element also encompasses user-interface controls that, on other platforms, are referred to as check boxes and radio buttons.

One easy way to think of buttons on Palm devices is that a button is a visible region on a form that has a known state. The state of a button can be changed programmatically, or by the users tapping the button.

Figure 7.1 presents a good example of a form that uses buttons: the Datebook main screen. On this screen, the New, Details, and Go To buttons are examples of push buttons. The days of the week at the top of the screen are good examples of using multiple buttons for a fixed number of choices on a form.

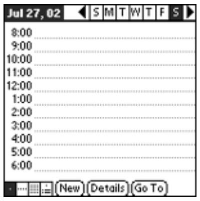


FIGURE 7.1 Buttons in use on the Palm Datebook main screen.

Input Fields

Input fields (known as *edit controls* in Windows) are used as a way to enter alphanumeric text via graffiti or an attached keyboard.

Input fields can be used for both single-line or multiple-line data entry.

Figure 7.2 illustrates the use of a multiple-line input field in the Memo application.

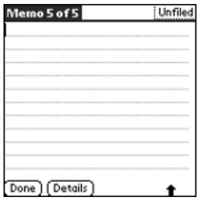


FIGURE 7.2 A multiple-line input field is used as the basis for the Memo application's New Memo form.

Labels

Labels are typically employed to display a piece of text describing another adjacent control ("Name" would be a label to the left of an input field where users enter a name value).

Labels can also be used to display either single- or multi-line text on their own.

Figure 7.3 illustrates the use of a label in the Address Book application.

Lists

Lists are used for displaying a set of choices to the users. Although sometimes multiple radio buttons can be used for this purpose, once you get beyond three or four options, or if your choices are variable in number depending on program circumstances, it makes more sense to present the users with choices in a list.

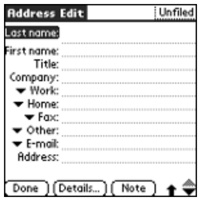


FIGURE 7.3 On the New Address form, each data-entry field has a text label on the left side of the screen.

Aside from supporting dynamic numbers of items, lists have other advantages over using multiple buttons. One is that they are more suited for displaying readable text. Another is that they can be associated with *triggers* (another type of element discussed in the next section) to create a complex user-interface element that behaves like a pop-up list.

Figure 7.4 illustrates the use of a list in the Address Book data entry form.

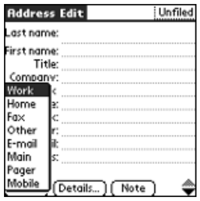


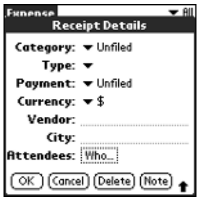
FIGURE 7.4 A list of possible phone number types for an address.

Triggers

Triggers are almost exclusively used in conjunction with other types of elements, as a way to automatically “trigger” the appearance of the other user-interface element. Triggers can be thought of as a type of button that can be hooked to a list or another user-interface element.

The combination of a trigger and a list is very powerful, and can save a tremendous amount of form real estate by displaying list choices only when they are needed and called for by the users.

Figure 7.5 illustrates the use of a trigger in the Expense application’s Receipt form.



Category: ▼ Unfiled
Type: ▼
Payment: ▼ Unfiled
Currency: ▼ $
Vendor:
City:
Attendees: Who...

FIGURE 7.5 The Attendees field at the bottom of the screen is a trigger that calls up the Attendees form.

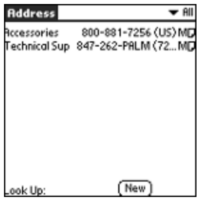
Tables

The table element is the most complex user-interface element available in the palette of form elements. Although, conceptually, a table is a list that can have multiple columns, in practice, a table is used as a scrollable, virtual viewport into rows of formatted data.

As opposed to a list box or list view in Windows, which can be filled with as many items as you want, a table only has a fixed number of visible rows, which are mapped to rows and columns of data, a relationship that the developer must manage. For this reason, code associated with tables is typically among the most complex in an application.

One could argue that table programming is overly complex, but unless you want to write your own custom *grid control*, it will at one point or another become necessary to learn how to use tables in your programs.

Figure 7.6 illustrates the use of a table in the Address Book's main form.



Accessories	800-881-7256 (US)	M
Technical Sup	847-262-PALM (72...	M

FIGURE 7.6 The Address Book main list is a prime example of a table used for displaying a scrolling view of a database of address records.

Custom Gadgets

The previously described list of form element types is sufficient to handle the needs of most applications that employ forms. In general, even if one of the pre-defined elements does not seem to exactly match your needs, it's worth looking at either modifying your user interface or adapting the appearance/behavior of one of the predefined elements.

However, there may be a situation where you truly believe the extra work in developing your own custom user-interface element is warranted. For this purpose, Palm provides something called a *gadget*. Think of a gadget as a placeholder form element that can be used as the basis for your own custom element. Successful gadget programming requires fairly advanced understanding of windows, pen actions, and drawing APIs, but once these techniques are mastered, gadgets can be a very effective way of creating a unique and effective user interface for your program.

Summary

This chapter introduced the various form elements and some of their uses in Palm applications. The next set of chapters drills down and provides a more in-depth examination of each type of form element. An actual code example is provided with each chapter to illustrate the element's purpose in a real program.

8 Button Controls

IN THIS CHAPTER

In our modern world, we are surrounded by knobs, dials, buttons, and switches. From ATMs to voting machines, automobiles to elevators, light switches to televisions, we are confronted with a myriad of ways in which we make selections, adjust our environment, and control the way we live and work.

As in the real world around us, computer programs also provide special visual elements that help us make selections, adjust behavior, or control the software's execution. The overall presentation of these elements on a computer program is called its user interface. The individual elements that help us navigate the user interface are commonly referred to as *user-interface controls* or simply *controls*.

As on most other computer platforms that support graphical displays, the Palm platform supports a rich array of controls for a developer to use in designing and constructing an application's user interface. This chapter begins a tour of the various types of controls, explores their uses, presents guidelines for implementation, and walks you through several examples of how to incorporate them into your application.

What Is a Control Object?

Control objects allow for user interaction with your program. As we've seen, forms are visual containers that often present groups of controls. Strictly speaking, Palm control objects fall into a larger family of objects called *user-interface objects*. User-interface objects encompass a wide range of types, covering buttons, menus, forms, list boxes, scrollbars, windows, and more. Control objects in the Palm SDK represent a subset of user-interface objects and as a group are generally variations on the push button concept.

This definition notwithstanding, I've found it easier to adopt a broader definition of a control as any reusable user-interface element that has a well-defined appearance and behavior and is supported as a user-interface element on a form.

This chapter explores most of the available types of buttons in depth, and subsequent chapters cover other types of user-interface elements that, also fall under the broader definition of a user-interface control.

Types of Control Objects

The Palm SDK provides support for a rich array of control objects. Each object has a name, a resource type that identifies it as a Palm resource, and a well-defined appearance and behavior. Table 8.1 describes each control object.

TABLE 8.1 Control Objects

Control Object	Resource	Appearance	Behavior
Button	`tBTN`	A rectangle containing a text label. Corners are round by default but can be set to have a rectangular frame.	When the user taps with the pen, the button highlights (inverts) the button's display region.
Push button	`tPBN`	Same as a button but always has square corners.	Same as a button, but the push button remains inverted after the pen is lifted.
Repeat button	`tREP`	Same as a button.	When the user taps with the pen, the object is repeatedly selected while the pen remains down. The repeating action stops when the pen is lifted.
Check box	`tCBX`	A square, either empty or containing a check mark, depending on the check box's setting. May have a text label associated with it, which appears to the right of the square.	When the user taps with the pen, the check box's setting is toggled on or off, either checking or unchecking the square on the display.
Pop-up trigger	`tPUT`	A text label to the right of a down-arrow graphic.	When tapped, the control initiates an associated pop-up list box.
Selector trigger	`tSLT`	A text label surrounded by a gray frame.	When tapped, the control initiates a user-interface event.

What Is a Button?

As you can see, although only one control object is actually called "button," all of the control objects have some button-like appearance or behavior. Buttons are one of the most prevalent types of user-interface elements in Palm applications, or indeed non-Palm applications.

Buttons are unique in that they provide simple, one-tap manipulation of the user interface. No other action is required by the user to complete a command or provide information: no graffiti or multi-step sequence of taps. Thus, buttons provide unprecedented power and simplicity to any user interface. Assuming you have made your buttons large enough (as the user-interface guidelines recommend), your entire application could theoretically be run by a person using his or her finger to manipulate the user interface.

About the only downside in using buttons instead of some other form of user-interface element is that they take up valuable real estate on the display. It is important to design your application so that the primary commands and actions are supported through onscreen buttons and secondary or optional commands are invoked using some other method (such as through menus).

Types of Buttons

Let's explore the various button types in depth so that you understand why you use each type in different situations. To illustrate each type, I present a figure of a built-in Palm application form that makes appropriate use of the button in question.

Using Buttons on Forms

Buttons are probably the most common user-interface element. In Palm parlance, a plain old "button" is usually employed to execute a command. On other platforms such as Windows, this type of button is called a push button. Push buttons are actually another type of button on the Palm. See Figure 8.1.

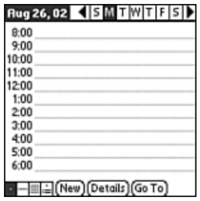
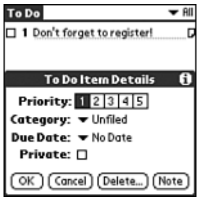


FIGURE 8.1 A button on a form.

Using Push Buttons on Forms

A push button on the Palm platform is what we are accustomed to calling a radio button. Due to screen real-estate considerations, these buttons are commonly grouped and placed adjacent to one another, but their behavior is essentially the same: You tap a single button to select a value from a set of possible values. Figure 8.2 shows push buttons used to provide the users with the ability to choose one of five possible priorities.


To Do ▾ All
☐ 1 Don't forget to register!

To Do Item Details
Priority: [✓ 1] 2 3 4 5
Category: ▾ Unfiled
Due Date: ▾ No Date
Private: ☐

FIGURE 8.2 Push buttons providing a multiple-choice selection.

Using Repeating Buttons on Forms

A repeating button is most commonly used to create something called an increment arrow. (In Windows applications, this is referred to as a spin button.) An increment arrow is usually placed next to a label or field and is used to move forward or backward through a series of values. Getting a repeat button to look like an increment arrow takes some work, and I show you how to do this later in the chapter. Figure 8.3 shows repeat buttons being used as left and right arrows to control the week being viewed.

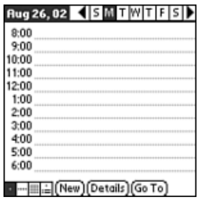
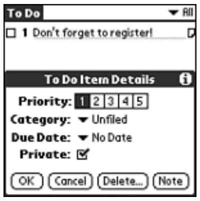


FIGURE 8.3 An increment arrow on the Appointment Day View Form.

Using Check Boxes on Forms

Surprise! A check box is exactly what its name implies: A visual toggle for managing a value that can have only one of two states. You use a check box for yes/no or true/false values. In Figure 8.4, a check box indicates a record as private or public.



☐ 1 Don't forget to register!

Priority: 1 2 3 4 5
Category: Unfiled
Due Date: No Date
Private: ☑

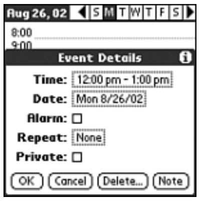
FIGURE 8.4 A check box.

Using Pop-Up Triggers on Forms

A pop-up trigger is most commonly seen in conjunction with a list box resource. Together, they give the functionality of a drop-down combo box. I discuss pop-up triggers further in Chapter 10, "Giving the User a Choice: Lists and Pop-up Triggers."

Using Selector Triggers on Forms

A selector trigger is best described as a "live" label; it displays a (noneditable) piece of text and also triggers an event when tapped. You can trap the event with a form handler and trigger some program options—such as a pop-up form. Upon returning from the event, the text can change to reflect the user's action. The selector trigger saves precious screen real estate by combining a text label with the functionality of a push button. One of the most common uses of a selector trigger is to handle date selection with the help of the `SelectDay` API. In Figure 8.5, both the `Date` and `Time` values are in fact selector triggers.



Time: 12:00 pm - 1:00 pm
Date: Mon 8/26/02
Alarm: ☐
Repeat: None
Private: ☐

FIGURE 8.5 A selector trigger used to select and display a date.

Guidelines for the Use of Buttons

The Palm SDK provides guidelines for the use and display of buttons in your application. These guidelines are excellent, and attempt to provide for a more consistent user interface across applications.

General Guidelines on Buttons

You should use buttons for the most important commands; relegate secondary and optional commands to pop-up dialogs and menus. This rule stays in line with the recommendation that you keep high-frequency actions accessible with one tap.

Make command buttons large enough to support finger navigation. When placing command buttons on a form, leave at least three pixels between the edge of the dialog box and the buttons. Locate command buttons at the bottom of the screen, and align them evenly. When sizing command buttons, leave one pixel above and below the top and bottom of the text area, using the maximum height of the selected font as a measurement guideline. When sizing push buttons (radio buttons), leave two pixels to the left and right of the text label.

Using Buttons in Your Application

You will interact with the Palm SDK in four ways when putting buttons (and controls in general) in your application:

- Creating the visual representation of the buttons in your application resource with Constructor
- Calling the appropriate SDK functions to control the button's behavior and appearance within your program
- Trapping events triggered by pen interaction with buttons
- Reading values in control structures passed to you through control object events

Note that the last two items are necessary only if you need to use a form event handler to provide special event handling in your form. If this is not a requirement, you can easily place button controls and other user-interface elements by creating the resources in Constructor, setting the control values and states at form-initialization time, and retrieving their values upon termination of the form.

A Simple Survey Application

To illustrate the use of each type of button, we'll create an application that presents a simple survey form. The form will ask the users to specify their marital status, age, date of birth, and whether they want to be contacted. The form supports a reset button to set the form values back to defaults and a submit button, which for our basic application simply displays the user's current choices from the form in a message box.

Figure 8.6 shows the survey form.

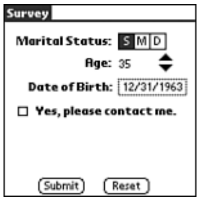


FIGURE 8.6 Our survey application.

As you can see, we use command buttons, push buttons (radio buttons), a repeat button, a selector trigger, and a check box all in one form.

The first thing you'll need to do is create the main form resource and place each control on the form. I assume that by now you are familiar enough with Constructor, so I skip the detailed steps. However, I'd like to cover just a few subtle nuances in the creation of the form resource. The following sections walk you through how to incorporate various button controls on your Palm forms.

Grouped Push Buttons

Constructor allows you to assign a group ID to which a push button control belongs. All push buttons on the form that belong to the same group ID will behave, to the user at least, the way that radio buttons are expected to behave. Tapping on one button will automatically select that button and deselect the other buttons in the group. Leaving the push buttons assigned to group ID 0 does not give you this behavior. You need to identify a group ID that is nonzero and assign the same ID to each button that belongs to the group.

If you happen to put multiple groups of push buttons on a single form (for example, if you add a set of buttons to let the users choose a favorite color), assigning each group a different group ID works just fine; each group operates independently.

Repeating Buttons and the Arrow Increment

Constructor does not offer built-in support for arrow-shaped repeating buttons. You have to manually tell Constructor what to use as the repeat button's label. To achieve an up-arrow and down-arrow look and functionality, you need to create two repeat buttons, one for up and one for down. The next part is not obvious: You need to set the label to a character code from a font that contains an arrow symbol. For up-down arrows, choose the Symbol 7 font in Layout Properties, and for the label, click the Hex check box (indicating the value you are entering should not be interpreted as a character string). You then enter `01` for the up arrow and `02` for the down arrow.

If you are like me and have not committed the character codes for the Symbol 7 font to memory, you might want to download a handy Palm utility called *ASCIIChart* by John Valdes, which displays all of the character codes and their values for each Palm font. (ASCII Chart is available from various online Palm software sources, such as `www.palmgear.com`.)

Listing 8.1 contains the code that implements the initialization and event handling for the main survey form.

LISTING 8.1 Source Code for the Main Survey Form

```
/*
   BTN_MAI.CPP
   Main form handling functions.
*/

// system headers
#include <PalmOS.h>
#include <SysEvtMgr.h>

// application-specific headers
#include "buttons.h"
#include "btn_res.h"

static void    MainFormInit (FormPtr formP);
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);

void ResetForm (FormPtr formP);
Int16 PalmMessageBox ( Int16 wAlertID, char * pMessage );

static Int16 s_Age = 35;

/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;
```

LISTING 8.1 Continued

```
switch (eventP->eType)
{
   case menuEvent:
   {
      FormPtr formP = FrmGetActiveForm ();
      handled = MainFormMenuHandler (formP, eventP);
      break;
   }

   case ctlRepeatEvent:
   {
      FormPtr formP = FrmGetActiveForm ();
      handled = MainFormButtonHandler (formP, eventP);
      break;
   }
   case ctlSelectEvent:
   {
      // A control button was tapped and released
      FormPtr formP = FrmGetActiveForm ();
      handled = MainFormButtonHandler (formP, eventP);
      break;
   }

   case frmOpenEvent:
   {
      // main form is opening

      FormPtr formP = FrmGetActiveForm ();

      MainFormInit (formP);

      FrmDrawForm (formP);

      handled = true;
      break;
   }

   default:
   {
      break;
   }
```

LISTING 8.1 Continued

```
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   formP - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr formP)
{
   // set the controls to an initial state
   ResetForm ( formP );
}


void
ResetForm (FormPtr formP)
{
   Int16 wIDContactMe = FrmGetObjectIndex (formP, MainContactMeCheckbox );
   ControlPtr pCtlContactMe =
      (ControlPtr) FrmGetObjectPtr (formP, wIDContactMe);

   Int16 wIDAge = FrmGetObjectIndex (formP, MainAgeLabel );
   ControlPtr pCtlAge =
       (ControlPtr) FrmGetObjectPtr (formP, wIDAge);

   Int16 wIDSingle = FrmGetObjectIndex (formP, MainSinglePushButton );
   ControlPtr pCtlSingle =
       (ControlPtr) FrmGetObjectPtr (formP, wIDSingle);

   Int16 wIDMarried = FrmGetObjectIndex (formP, MainMarriedPushButton );
   ControlPtr pCtlMarried =
       (ControlPtr) FrmGetObjectPtr (formP, wIDMarried);

   Int16 wIDDivorced = FrmGetObjectIndex (formP, MainDivorcedPushButton );
   ControlPtr pCtlDivorced =
       (ControlPtr) FrmGetObjectPtr (formP, wIDDivorced);

   Int16 wIDDateofBirth =
```

LISTING 8.1 Continued

```
        FrmGetObjectIndex (formP, MainDateofBirthSelTrigger);
    ControlPtr pCtlDateofBirth =
        (ControlPtr) FrmGetObjectPtr (formP, wIDDateofBirth);

    char szAge[5];
    char szDate[12];

    // set the marital status radio buttons to Single
    // Note that because this is grouped, user taps will automatically
    // turn off the untapped selections, but programmatically we have to
    // do it ourselves.
    CtlSetValue ( pCtlSingle, 1 );
    CtlSetValue ( pCtlMarried, 0 );
    CtlSetValue ( pCtlDivorced, 0 );

    // set the label for age to 35
    s_Age = 35;
    StrPrintF ( szAge, "%d", s_Age );

    FrmHideObject (formP, wIDAge);
    FrmCopyLabel (formP, MainAgeLabel, szAge);
    FrmShowObject (formP, wIDAge);


    // set the date of birth label on the selector button to be 12/31/63
    Int16 mth, day, year;
    mth = 12;
    day = 31;
    year = 1963;
    StrPrintF ( szDate, "%d/%d/%d", mth, day, year );

        char *pLabel = (char *)CtlGetLabel (pCtlDateofBirth);
        StrCopy (pLabel, szDate);
        CtlSetLabel (pCtlDateofBirth, pLabel);


    // set the check box for "contact me" to no
    CtlSetValue ( pCtlContactMe, 0 );


}
```

LISTING 8.1 Continued

```
/*
   MainFormMenuHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormMenuHandler (FormPtr /*formP*/, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.menu.itemID)
   {
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormButtonHandler (FormPtr formP, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      case MainAgeUpRepeating:
      {
         // decrement the age
```

LISTING 8.1 Continued

```
        if ( s_Age > 22 )
        {
           Int16 wIDAge = FrmGetObjectIndex (formP, MainAgeLabel );
           ControlPtr pCtlAge =
                  (ControlPtr) FrmGetObjectPtr (formP, wIDAge);
           char szAge[5];

           s_Age--;
           StrPrintF ( szAge, "%d", s_Age );
           FrmHideObject (formP, wIDAge);
           FrmCopyLabel (formP, MainAgeLabel, szAge);
           FrmShowObject (formP, wIDAge);
        }
        break;
     }
     case MainAgeDownRepeating:
     {
        // increment the age
        if ( s_Age < 65 )
        {
           Int16 wIDAge = FrmGetObjectIndex (formP, MainAgeLabel );
           ControlPtr pCtlAge =
               (ControlPtr) FrmGetObjectPtr (formP, wIDAge);
           char szAge[5];

           s_Age++;
           StrPrintF ( szAge, "%d", s_Age );

           FrmHideObject (formP, wIDAge);
           FrmCopyLabel (formP, MainAgeLabel, szAge);
           FrmShowObject (formP, wIDAge);
        }
        break;
     }
     case MainDateofBirthSelTrigger:
     {
        // tapped the selector trigger button control
        // show the selectday dialog, and
        // update the date of birth label upon
        // return
        Int16 mth, day, year;
        mth = 12;
```

LISTING 8.1 Continued

```
         day = 31;
         year = 1963;

         if ( SelectDay ( selectDayByDay, &mth,
                          &day, &year, "Date of Birth") )
         {
            ControlPtr pCtlDateofBirth =
                (ControlPtr)eventP->data.ctlEnter.pControl;
            char szDate[10];

            StrPrintF ( szDate, "%d/%d/%d", mth, day, year );
            CtlSetLabel ( pCtlDateofBirth, szDate );
         }
         handled = true;
         break;
      }

      case MainSubmitButton:
      {
         char szMessage[100];

         // get the values from the survey form
         Int16 wIDContactMe =
                FrmGetObjectIndex (formP, MainContactMeCheckbox );
         ControlPtr pCtlContactMe =
                (ControlPtr) FrmGetObjectPtr (formP, wIDContactMe);

         Int16 wIDAge =
                FrmGetObjectIndex (formP, MainAgeLabel );
         ControlPtr pCtlAge =
                (ControlPtr) FrmGetObjectPtr (formP, wIDAge);

         Int16 wIDSingle =
                FrmGetObjectIndex (formP, MainSinglePushButton );
         ControlPtr pCtlSingle =
                (ControlPtr) FrmGetObjectPtr (formP, wIDSingle);

         Int16 wIDMarried =
                FrmGetObjectIndex (formP, MainMarriedPushButton );
         ControlPtr pCtlMarried =
                (ControlPtr) FrmGetObjectPtr (formP, wIDMarried);
```

LISTING 8.1 Continued

```
Int16 wIDDivorced =
       FrmGetObjectIndex (formP, MainDivorcedPushButton );
ControlPtr pCtlDivorced =
       (ControlPtr) FrmGetObjectPtr (formP, wIDDivorced);

Int16 wIDDateofBirth =
       FrmGetObjectIndex (formP, MainDateofBirthSelTrigger);
ControlPtr pCtlDateofBirth =
       (ControlPtr) FrmGetObjectPtr (formP, wIDDateofBirth);

const char * pAge = CtlGetLabel ( pCtlAge );
const char * pDateofBirth = CtlGetLabel ( pCtlDateofBirth );

char szContactMe[20];
char szMarital[50];
char szAge[20];
char szDateofBirth[50];

StrPrintF ( szAge, "Age = %s", pAge );
StrPrintF ( szDateofBirth, "DOB = %s", pDateofBirth );

if ( CtlGetValue ( pCtlContactMe ) )
{
   StrCopy ( szContactMe, "Contact me" );
}
else
{
   StrCopy ( szContactMe, "Do not contact me" );
}
if ( CtlGetValue ( pCtlSingle ) )
{
   StrCopy ( szMarital, "Marital Status = Single" );
}
else if ( CtlGetValue ( pCtlMarried ) )
{
   StrCopy ( szMarital, "Marital Status = Married" );
}
else
{
   StrCopy ( szMarital, "Marital Status = Divorced" );
}
```

LISTING 8.1 Continued

```
        StrPrintF ( szMessage, "%s, %s, %s, %s",
                    szMarital, szAge, szDateofBirth, szContactMe );

        PalmMessageBox (MessageBoxAlert, szMessage);

        handled = true;
        break;
     }

     case MainResetButton:
     {
        ResetForm (formP);
        handled = true;
        break;
     }

  }
  return handled;
}
```

The main form's event handler (amazingly enough, called `MainFormEventHandler`) is where all the action begins.

Initializing the Form

As you saw in earlier chapters, if you want to be able to trap events that happen while a form is active, you have to supply a form event handler. All main application forms have an event-handler function, and this event handler also processes form initialization. (Contrast this with a modal pop-up dialog, in which form initialization happens before the form event handler receives notification.) The event that signals that the form is coming to life is `frmOpenEvent`. We trap this event and pass control to `MainFormInit`.

`MainFormInit`'s job is to set the initial display state of the various controls on the form. There are two times in the life of the survey form when we want to do this: during form initialization and when the reset button is tapped. We can save ourselves some work and create a single function `ResetForm` to handle both instances.

`ResetForm` is where, for each control, we obtain a `ControlPtr`. This is then used in various SDK control functions to set the controls' values. For push buttons and check boxes, we use `CtlSetValue` to set each control's state to `1` or `0`. You'll note that we have to explicitly set each push button (radio button) to on or off, even though this behavior is automatic for the users. The programmer has no such luxury. You must set each button manually so that only one button in a group is selected; otherwise, it is possible to select multiple buttons at the same time.

For controls whose values are reflected by their labels, we use `CtlSetLabel`. The two kinds of user-interface elements in use here are a label (technically not a control at all) and a selector trigger.

It is perfectly legal to set a selector trigger's label using `CtlSetLabel`, but the one catch is that you should make sure the label specified in the form resource for the trigger selector is at least as long as the longest value you will copy into the label. In this case, we will always be using a formatted date string (mm/dd/yyyy), so you can use 10 blank spaces for the label in Constructor.

For label objects, you actually cannot use `CtlSetLabel`. You should use form functions such as `FrmCopyLabel` and `FrmGetLabel` to manipulate label objects. Even then, there is some weirdness when trying to change label objects at runtime. I cover fields and labels in depth in Chapter 9, "Labels and Fields."

Trapping Control Events

In `MainFormEventHandler`, you'll see that we are interested in two events generated by controls: `ctlRepeatEvent` and `ctlSelectEvent`. Both events are passed to a common routine `MainFormButtonHandler` for processing.

`ctlRepeatEvent` is generated by repeat buttons (and scrollbars) for every half second that the pen remains within the bounds of the control. Because we have two repeat buttons (one each for incrementing or decrementing the age count), we check which one is being tapped by examining the `controlID` member of the event structure. Then, we simply increment or decrement the age counter, format it as a string, and reset the age label.

`CtlSelectEvent` is generated when the pen is lifted after having been tapped inside the bounds of a button. We have three buttons for which we care to trap this message: the date-of-birth selector trigger, the submit button, and the reset button. (The marital status radio buttons and the check box also generate these events, but the automatic handling of the events by Palm is sufficient for our purposes.) The selector trigger simply launches into the standard `SelectDay` API and, upon return, resets the label to the new date. The reset button, as mentioned earlier, maps to the common `ResetForm` function.

The submit button requires us to obtain the value for each of the controls on the form and display them to the users in a message box. This processing is the reverse of what happened at initialization time. Given a `ControlPtr` to each control, we call either `CtlGetValue` or `CtlGetLabel` to obtain the controls' values.

One final note before I close this treatment of buttons: If you're like me, you'll quickly tire of writing the same two lines of code over and over again to obtain a `ControlPtr`. You might want to add the following function to your (ever-growing) toolbox of utility functions:

```
ControlPtr PalmGetControlPtr (FormPtr formP, Word objID)
{
    ControlPtr p;
    Word wID = FrmGetObjectIndex (formP, objID);
    P = (ControlPtr)FrmGetObjectPtr (formP, wID);
    Return p;
}
```

Summary

This chapter provided a focused look at how the various types of buttons are incorporated in a form. I covered some of the areas in which you must take some not-so-obvious steps to achieve some common user-interface goals. The next chapter moves on to the subject of labels and fields.

9 Labels and Fields

IN THIS CHAPTER

Despite the presence of the graffiti handwriting-recognition capability and the growing proliferation of portable keyboards made specifically for use with PDAs, handheld computing devices remain awkward to use when it comes to performing data entry. As a result of this, much of the data found on PDAs was entered on a desktop computer and then used in a "read-only" fashion on the PDA. That said, in many applications, entering data on a mobile device is either desirable or even essential to the successful use of the application.

Although it is popular to poke fun at the awkwardness and inaccuracies associated with handwriting recognition, the problem is really largely due to ergonomics. When you use a PDA in real life, you are generally not in a situation optimized for lengthy data entry. More typically, you are mobile, on an airplane, in a car (passenger seat of course!), standing in the hallway, walking with a colleague, dashing off from a meeting, or sitting in someone else's office. In most of these scenarios, there is not a stable flat surface to use as a support for pen- or keyboard-based data entry. Even if there is, the normal posture and arm rest positioning is not present. Thus our job as application developers is to keep in mind the target users, and design our user interfaces such that they offer the users the most efficient way possible to enter the essential data required by our applications.

The most well designed Palm applications tend to minimize data entry as a required activity for the users, relying instead on single-tap access and other assisted navigation through the application's data. When the users must create data on the Palm, providing predefined choices through pop-up lists, buttons, and tables goes a long way toward avoiding unnecessary and cumbersome data entry. It is also worth considering moving some of the more onerous data entry portions of your application to the desktop and then providing a conduit that can exchange the data with the Palm.

That said, some situations still call for the capability to record data in the field. Quickly jotting down a memo, scribbling a to-do task before it slips your mind, or adding some notes regarding a client appointment are all compelling needs for the mobile computer user and provide a legitimate reason to include some simple data entry in an application.

This chapter focuses on the primary means for directly accepting textual or numeric user input in a Palm application: the field user-interface object. I also briefly cover the use of label resources as a complementary companion to field objects in forms.

Form Labels

I'll tackle the easiest topic first. A *label* displays noneditable text on a form. Labels are used in a variety of ways, but most labels provide a visual cue about the user-interface element (field, list, and so on) directly to the right or below the label. For example, on the Expense application's edit form, the Vendor and City fields have labels to their left. There are other uses for labels: a short bit of instruction at the top of a form or a way to get text to the left of a check box (although it will not respond to a tap).

The Palm style guidelines dictate that labels should be right justified and displayed using a bold font. It is also customary to use a colon at the end of the label.

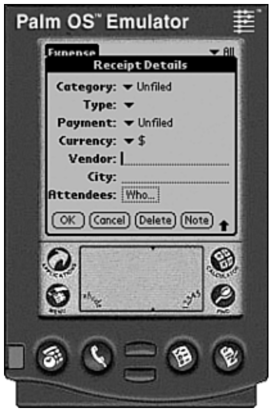
Input Fields

A field object, also referred to as an input field, is akin to an edit control on Windows. It displays one or more lines of text and provides editing capability either through graffiti strokes in the special graffiti area or by direct pen manipulation within the field. Fields are typically distinguished visually from labels by underlines, although it is possible to create a field that is not underlined.

Figures 9.1 and 9.2 show a single-line field and a multiline field, respectively, in action within a Palm application.

Fields support some fairly advanced features, such as proportional fonts, left and right justification, single-line or multiline editing, cutting, copying, and pasting, and scrolling. (Note that only vertical scrolling for multiline fields is supported; there's no horizontal scrolling within a single line of text.)

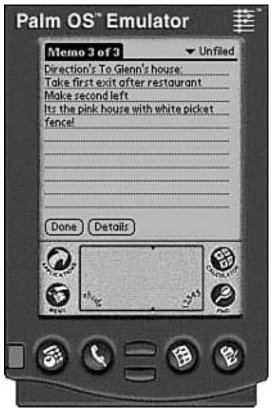
Like a control object, a field object has an associated resource type and a set of SDK functions that let you interact with it in your application. Field objects also handle pen and other system events and pass those events to your code as field events.



Receipt Details

Category: Unfiled
Type:
Payment: Unfiled
Currency: $
Vendor:
City:
Attendees: Who...

FIGURE 9.1 The Expense edit form. The Vendor and City fields show labels and single-line fields.



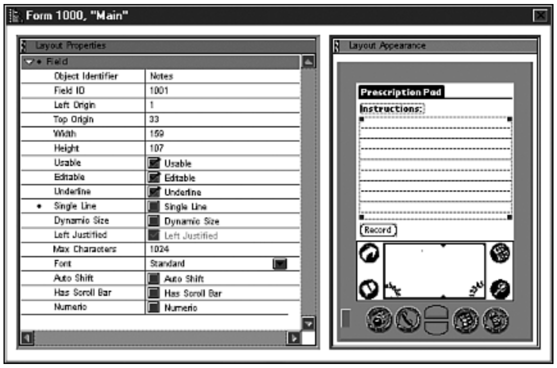
Direction's To Glenn's house:
Take first exit after restaurant
Make second left
Its the pink house with white picket
fence!

FIGURE 9.2 The Memo application's New Memo form. Most of the form is devoted to one large multiple-line field.

Using Constructor to Create Labels and Fields

Labels and fields are created in Constructor and are fairly straightforward to define. Labels in particular can't be configured much other than setting the position, size, ID, and text.

Fields are slightly more complicated. Figure 9.3 displays a Constructor session for the Prescription application that we create later in this chapter.



Field

Object Identifier	Notes
Field ID	1001
Left Origin	1
Top Origin	33
Width	159
Height	107
Usable	✓ Usable
Editable	✓ Editable
Underline	✓ Underline
• Single Line	▢ Single Line
Dynamic Size	▢ Dynamic Size
Left Justified	✓ Left Justified
Max Characters	1024
Font	Standard
Auto Shift	▢ Auto Shift
Has Scroll Bar	▢ Has Scroll Bar
Numeric	▢ Numeric

FIGURE 9.3 Creating a field in Constructor.

Let's quickly scan the unique attributes for a field resource in Table 9.1.

TABLE 9.1 Field Resource Attributes

Attribute	**Meaning**
Editable	If checked, sets the field to be editable. (Makes sense, right?) Although you usually will want to set this attribute, leaving it unchecked can be a handy way to create a programmatically changeable label that is less cumbersome to program than a regular label.
Underline	Also usually checked. If checked, each displayed line of text has a gray underline.
Single Line	Fields can be single line or multiline. Single-line fields do not allow the users to enter text beyond the visible extent of the field. They also do not accept tab or return characters. Multiline fields expand visually as the user enters more text.
Dynamic Size	If this is checked, the number of lines in a multiline field is increased or decreased as characters are added or removed. This setting is only applicable to multiline fields.
Left Justified	It is recommended for consistency that single-line fields be set as left justified. This setting has no effect on multiline fields.
Max Characters	This is the maximum number of characters that a field will accept. Multiline fields will continue to expand in size until they reach this limit. Single-line fields will beep if the limit is reached before the visually displayed field is filled with characters. The maximum possible number of characters in a field is 32,767.
Auto Shift	If this is checked, the field will automatically change the graffiti you enter so that an uppercase letter is substituted after an empty line in a multiline field, after a period or other sentence-terminating punctuation, and after two spaces are recorded.

Using Labels and Fields in Your Application

In many respects, incorporating fields into your application forms is just like the process for any other user-interface object: Create the field as a resource using Constructor, and use the SDK functions to initialize and interact with the field object within your form logic. Although it's not a typical thing that you need to do, you can also respond to events that are passed to you by the field.

IMPORTANT NOTE

Probably the trickiest development issue associated with using fields is that if you want to set the text to be displayed in a field, you actually are responsible for allocating a memory block handle that is attached to the UI portion of the field, copying the desired text to the memory block, and passing it to the field. Although tempting, you cannot simply write

```
FldSetText(myfield, "Hello There");
```

Let me qualify that statement. You can actually do something like this, but the consequences are not attractive: The field will not be able to resize the field object's memory if the user changes the text in the field. In this case, the field does not even make a copy of the text; it simply retains the pointer that you pass in. Although it is more cumbersome to deal with memory handles, the downsides of not doing so are too undesirable to consider.

The proper way to set the text for a field object is by calling the field function `FldSetTextHandle()`. Once set, this memory handle is managed by the field object and will even get resized automatically as is necessary based on the user's input (which is subject to the maximum character length set in the field's resource attributes). Keep in mind that if you pass a memory handle to a field object, you are responsible for freeing the field's previously associated memory handle. Confused? Think of it this way: A field object comes with its own memory handle. If you replace it, you should clean up the old one. When the field object is destroyed (usually by the form when it is itself destroyed), the field knows enough to clean up after itself by destroying its own memory handle. One lesson to be learned here is to not intermix memory handles that are passed to fields with memory handles that have a purpose and expected lifetime outside the scope of a field. If you will be using a field, create a memory block specifically for the purpose of managing the field's memory, distinct and separate from any memory associated with your application such as a structure or variable.

When the time comes that you need to get the current text associated with a field object, you can obtain the handle by calling `FldGetTextHandle()`. Again, be careful about ownership of this handle; the field will dispose of it when the field is destroyed.

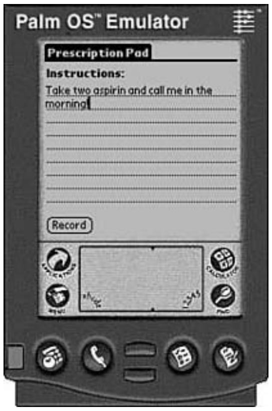
If you want a specific field object to automatically get the input focus when a form opens, you can use `FrmSetFocus` with the index of the field.

There are many other field behaviors and attributes that you can set using the field functions, but in most applications, it is sufficient to be able to set and get the field value when a form opens and closes.

Just What the Doctor Ordered

To illustrate the use of fields, we'll create a simple form that allows a doctor to write a prescription for a patient on the Palm using the pen stylus and graffiti. (We've all seen how unreadable some handwritten prescriptions are. With our application, as long as our fictional doctor can manage graffiti, we should have no problem reading the doctor's orders!)

Figure 9.4 shows the form in action.



Take two aspirin and call me in the morning.

FIGURE 9.4 Sage advice from the good doctor!

Listing 9.1 contains the code for the main form.

LISTING 9.1 The Source for the Main Form in "Prescription Pad"

```
/*
   FLD_MAI.CPP
   Main form handling functions.
   Copyright (c) Bachmann Software and Services, 2002
   Author: Glenn Bachmann
*/

// system headers
#include <PalmOS.h>
#include <SysEvtMgr.h>
```

LISTING 9.1 Continued

```
// application-specific headers
#include "fields.h"
#include "fld_res.h"

static void    MainFormInit (FormPtr formP);
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);

Int16 PalmMessageBox ( Int16 wAlertID, char * pMessage );

/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;

   switch (eventP->eType)
   {
      case menuEvent:
      {
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormMenuHandler (formP, eventP);
         break;
      }

      case ctlRepeatEvent:
      {
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormButtonHandler (formP, eventP);
         break;
      }
      case ctlSelectEvent:
      {
         // A control button was tapped and released
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormButtonHandler (formP, eventP);
         break;
```

LISTING 9.1 Continued

```
      }

      case frmOpenEvent:
      {
         // main form is opening

         FormPtr formP = FrmGetActiveForm ();

         MainFormInit (formP);

         FrmDrawForm (formP);

         handled = true;
         break;
      }

      default:
      {
         break;
      }
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   formP - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr formP)
{
   // set the controls to an initial state
}

/*
   MainFormMenuHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
```

LISTING 9.1 Continued

```
            false - not handled
*/


Boolean
MainFormMenuHandler (FormPtr /*formP*/, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.menu.itemID)
   {
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormButtonHandler (FormPtr formP, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      case MainRecordButton:
      {
         Int16 wIDField = FrmGetObjectIndex (formP, MainNotesField );
         MemHandle hText = FldGetTextHandle (pCtlField);
         char * pText = (char *)MemHandleLock (hText);
```

LISTING 9.1 Continued

```
         PalmMessageBox ( MessageBoxAlert, pText );

         MemHandleUnlock (hText);
         handled = true;
         break;
      }

   }
   return handled;
}

/*
   PalmMessageBox:

   Toolbox api providing a general purpose message box function

   Parms: wAlertID = id of alert to show
          pMessage = ptr to message text to display

   Returns: id of button chosen by user
*/

Int16
PalmMessageBox ( Int16 wAlertID, char * pMessage )
{
   Int16 err;

   err = FrmCustomAlert (wAlertID, pMessage, "", "" );

   return err;
}
```

Summary

In this chapter you learned how to deal with labels and fields in a Palm application. Although not the most exciting user interface element, labels and fields are found in almost all applications, including the ones supplied as part of the Palm OS. With some of the nuances of these elements in mind, you should now have the information you need to quickly add field and label support to your program.

10 Giving the Users a Choice: Lists and Pop-up Triggers

IN THIS CHAPTER

In the spirit of providing the users with the capability to navigate an application's user interface with a minimum of pen strokes, it is a good idea to present the users with short, predefined choices rather than make them use graffiti to make a choice. Doing so can make using your application a breeze; all the users have to do is tap a few times, and they have the information they need.

Lists are an excellent way of organizing choices for the user. You saw in Chapter 8, "Button Controls," that push buttons are one way to provide a multiple-choice, radio-button–like facility in situations where there are a few, easily recognizable choices (such as the days of the week). Obviously, there are limitations in the number of choices that can be offered using push buttons and the extent to which you can describe each choice in the label area.

In this chapter, you will explore how to use list user-interface elements in your Palm applications. I cover

- Behavior and appearance of lists on the Palm
- Guidelines for the use of lists in your application
- How to create list resources
- How to create pop-up list resources
- How to add list and pop-up list handling to your application code

Lists in the Palm Environment

Lists, or list boxes, are user-interface elements that present the users with a list of choices. The users can select any item in the list with the pen, resulting in that item's display being inverted. List boxes have built-in scrolling, with up and down arrows appearing to the right of the list if the number of items in the list exceeds the displayable area.

Examples of Lists in Action on the Palm

Figure 10.1 shows a list being used in the Palm Set Time form in the Appointments application. The hours and minutes selectors are both list boxes.

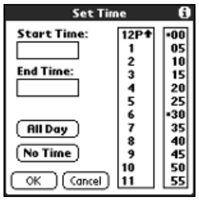


FIGURE 10.1 The Set Time form in the Appointments application.

Pop-up Lists

When combined with a pop-up trigger, a list box becomes a pop-up list. Under Windows, this functionality is bundled as a single unit, called a combo box control. When programming the Palm, you have to put the two pieces together yourself. The initial display of a pop-up list is a single displayed item representing the current selection and a down arrow indicating the availability of more choices. When any part of the pop-up list is tapped (either the arrow or the text), the full list of available choices "drops down." The list stays visible until either an item from the list is tapped or another area of the form is tapped.

Figures 10.2 and 10.3 show a pop-up list in the Address Book application's edit form before tapping and after the list is popped up.

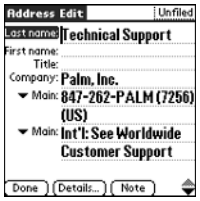

Address Edit Unfiled
Last name: Technical Support
First name:
Title:
Company: Palm, Inc.
Main: 847-262-PALM (7256) (US)
Main: Int'l: See Worldwide Customer Support

FIGURE 10.2 The Address Edit form has a pop-up list for each phone number field that allows you to choose the type of number represented in each field (work, home, fax, and so on).

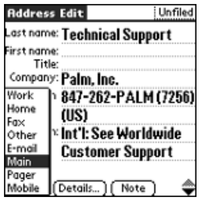

Address Edit Unfiled
Last name: Technical Support
First name:
Title:
Company: Palm, Inc.
847-262-PALM (7256) (US)
Int'l: See Worldwide Customer Support

FIGURE 10.3 The Number Type pop-up list in its “popped up” state.

Guidelines for Using Lists in an Application

You should use lists in situations where there are a predefined number of static options. The number of items in your list should be more than one or two; otherwise, consider using push buttons.

At the other end of the spectrum, ideally your list will not contain more items than are visible on the list display at one time. Scrolling is a nuisance for the users, and lists also have a shortcoming in the scroll department; they do not automatically scroll when the users choose the hard scroll button located in the bottom middle row of buttons on the device.

Also note that lists do not natively provide for advanced features such as in-place editing, nor are they designed to contain other types of user-interface elements such as buttons or bitmaps. This functionality is available with a different type of user-interface element, a table. Tables are significantly more complex to understand and use than lists, and are covered in Chapter 11, “Tables.”

Creating List Resources

List resources are created with the help of a resource editor (such as Metrowerks’ Constructor or PilRC), and if all you need is a simple list, the operation is pretty straightforward. Other than sizing and positioning, the attributes you need in the layout properties pane are Usable, Visible Items, and List Items.

Usable

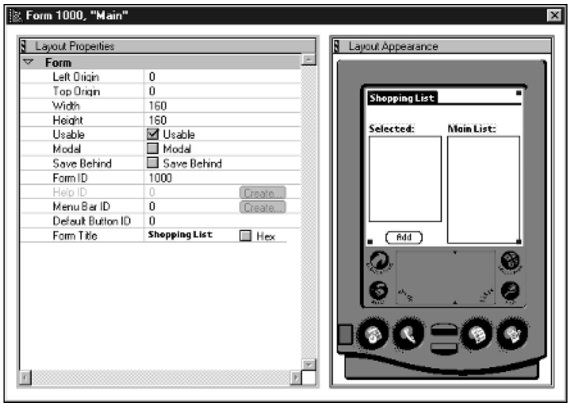
For normal lists, leave the Usable box checked. For lists to be associated with pop-up triggers, you should uncheck this box so that the list does not draw until triggered.

Visible Items

Set Visible Items to the desired height of the list box display, expressed in terms of the number of items. This attribute is used to vertically size the list on the form, rather than explicitly specify the height in pixels.

List Items

If you know the possible values for your list ahead of time, it can be convenient to directly type them into the list's resource definition. Highlight the List Items attribute, and choose Edit, New Item Text from the Constructor menu to add a new item. Tab over to the item text, and type the text for the item. Repeat this process until you have all the items entered. Note that it is perfectly fine to leave List Items blank. Depending on how you will use your list, it might be preferable to set the list items at runtime or at resource design time. Figure 10.4 shows Constructor being used to create a list resource.



Form	
Left Origin	0
Top Origin	0
Width	160
Height	160
Usable	✓ Usable
Modal	☐ Modal
Save Behind	☐ Save Behind
Form ID	1000
Help ID	0
Menu Bar ID	0
Default Button ID	0
Form Title	Shopping List ☐ Hex

FIGURE 10.4 Creating a list resource in Constructor.

One thing you should know is that there is no built-in sorting capability associated with lists, so you might want to define the list items in the most desirable sort order.

Note that the list attributes in the resource are important in getting your list to behave correctly. It is all too common to discover that the source of a program bug is as simple as an incorrectly specified list attribute in Constructor!

Creating Pop-up List Resources

Creating a pop-up list requires adding two resources to a form and connecting them: a pop-up trigger button and a list.

Step 1: Add a pop-up trigger resource to your form.

Give the pop-up trigger an object identifier because you will be referring to this identifier in your application code. You can leave the `List ID` field blank for the moment.

Step 2: Add a list resource to your form.

You should position the list so that its upper-left border corresponds to the upper-left border of the pop-up trigger.

Step 3: Set the list attributes.

Set the `Usable` attribute to `false`, type the list items to be displayed, and set the `Visible Items` attribute to exactly match the number of items.

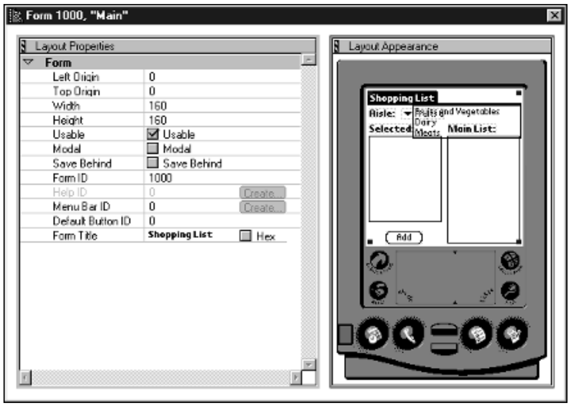
Step 4: Give the list an object identifier.

It is okay to use the same ID as the one for the pop-up trigger because Constructor will append the object type to your chosen identifier. Note the `List ID` attribute for the list resource.

Step 5: Assign the `List ID` to the pop-up trigger's list ID attribute.

You will find that it is awkward (or impossible) to select the pop-up trigger once the list resource is located on top of it. To get around this, Constructor provides another way to get at resource properties: under the Layout menu, choose Show Object Hierarchy. This displays a list of resources that are associated with the current form. Clicking a resource brings up the resource's properties just as if you had clicked it on the visual display. Figure 10.5 shows Constructor being used to create a pop-up list resource.

Using PilRC to create your form is one way to avoid this problem, which is unique to the visual design approach of Constructor.


Form 1000, "Main"

Layout Properties

Form	
Left Origin	0
Top Origin	0
Width	160
Height	160
Usable	☑ Usable
Modal	☐ Modal
Save Behind	☐ Save Behind
Form ID	1000
Help ID	0 Create...
Menu Bar ID	0 Create...
Default Button ID	0
Form Title	Shopping List ☐ Hex

FIGURE 10.5 Creating a pop-up list resource in Constructor.

Adding List and Pop-up List Handling to Your Application

The effort required for supporting lists in your application can range from almost none to moderately extensive. For a list or pop-up list that has a predefined set of items associated with the resource, your only real task is to determine the currently selected item at the proper time in your form's event handling (for example, when the OK button is tapped).

To obtain the currently selected item in a list, you call `LstGetSelection`, passing it the pointer to your list object. This pointer is obtained in the usual way:

```
Word wIDList = FrmGetObjectIndex (pForm, MyList );
ListPtr pList = (ListPtr) FrmGetObjectPtr (pForm, wIDList);
```

Given the list pointer, you obtain the zero-based index of the currently selected item:

```
uIndex = LstGetSelection (pList);
```

Given the index of any item on the list, you can obtain its text:

```
CharPtr pText = LstGetSelectionText (pList, uIndex);
```

This function is somewhat poorly named because it will give you the text associated with the item at index `uIndex`, regardless of whether it is selected.

One shortcoming of the list object is that there is no provision for setting a programmer-defined value to be associated with each list item. You must actually compare the item's text value in order to understand the item selected. This unfortunately results in code such as the following:

```
if (0 == StrCompare (pText, "Red"))
{
   // Item chosen was the red item;
}
else if (0 == StrCompare (pText, "Blue"))
{
   // Item chosen was the blue item
}
else if (0 == StrCompare (pText, "No Color"))
{
   // Item chosen was no color
}
```

If the text in the list item changes, you have to change your code. Keeping this in mind, you might want to isolate this type of comparison code to a helper function that returns a numeric value representing the chosen item. This way, you will have to change only one place in your code. I use this technique in the sample application that you will examine in a moment.

More Complex Handling—Drawing Your Own List

For situations where you want to control the appearance and behavior of lists at runtime, you have a little more work to do. Contrary to what you might expect, there are no `LstAddItem` and `LstRemoveItem` functions that allow you to manipulate list contents on a granular basis. Rather, you must supply the entire list at once using

```
LstSetListChoices (pList, ppItemText, itemCount);
```

`pList` is a pointer to the list object, `ppItemText` is a pointer to an array of text strings, and `itemCount` is the number of items you are passing.

If you hadn't gotten the idea that lists were designed with predefined choices in mind as the primary goal, it should now be crystal clear. Although you could theoretically do it this way, it would be a lot of work to create a special array of text pointers and reset the list items every time a dynamic list changed. As you can imagine, this type of interface is not terribly conducive to displaying a variable number of database records, for example.

That said, there is a way to work around this problem using a technique called a *draw function*. In this scenario, you pass `NULL` to `LstSetListChoices` and pass it the number of items you want in the list. In this scenario, all the list really knows is how many items it has; everything else is up to the program to handle. Because the list will not know how to draw your list's items without any text, it is up to you to draw it yourself.

This is accomplished with the use of a callback draw function. If you set a draw function using `LstSetDrawFunction`, the list object will call it every time each list item needs to be drawn, passing the zero-based index of the item. You can use this item as an index into your own data storage, which can be an in-memory list, an indexed array of object pointers, a database, or anything you want. This is the technique I adopt in the sample application, and because of its flexibility, I recommend it for those situations where your list display needs are beyond the most basic, but still not complex enough to warrant the use of a table.

More Complex Handling—Trapping List Events

If you need to trap pen down or up events in a list box, you can do so. `LstSelectEvent` is generated when an item is selected (a pen down followed by a pen up). For more granular control, a `LstEnterEvent` is generated on a pen down, and a `LstExitEvent` is generated on a pen up. Most applications do not need this level of control, but it's nice to know it's there if you need it.

Pop-up List Handling

Pop-up lists are slightly different in terms of event handling. For the most part, you will want to know when the users have changed the item selection in the pop-up trigger's associated list. To trap this event, you don't look for an event from the list object. Rather, you look for a `popSelectEvent`, which is generated each time the users select an item from a pop-up list. The IDs of both the trigger that generated the event and the associated list object are passed along with the event. You will need the ID of the list object in order to query the new selected item.

A Shopping List Revisited

Back in Chapter 6, "Interacting with the User: Forms," we built a rudimentary shopping list form that allowed the users to use check boxes and push buttons to select items. Unfortunately, the only items this application allowed you to shop for were milk and cookies (hardly the type of diet your doctor would recommend).

Here, you will take another pass at a shopping list application, and in doing so, you will rely heavily on lists and pop-up lists to provide a more extensive set of choices to the users. The user interface for the new version is shown in Figure 10.6. The Selected and Main lists are list boxes that you draw yourself. The Aisle control is a pop-up list.

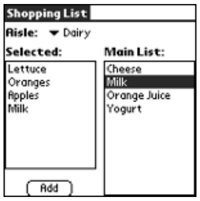


FIGURE 10.6 The new shopping list application.

As you can see, the application supports different classes of items based on the aisle in which they are located. All available items for the currently selected aisle are displayed in the Main list on the right side of the form. If you want to add an item to your shopping list, you highlight it and tap the Add button.

The source code that implements the form shown is in Listing 10.1.

LISTING 10.1 The Code for the Main Form in the Shopping List Application

```
/*
   LIST_MAI.CPP
   Main form handling functions.
   Copyright (c) Bachmann Software and Services, 1999-2002
   Author: Glenn Bachmann
*/

// system headers
#include <PalmOS.h>
#include <SysEvtMgr.h>

// application-specific headers
#include "lists.h"
#include "list_res.h"

static void    MainFormInit (FormPtr pForm);
static Boolean MainFormButtonHandler (FormPtr pForm, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr pForm, EventPtr eventP);

// description of a single shopping item
typedef struct
{
   char     szItem[50];
} ShoppingItem;
```

LISTING 10.1 Continued

```
// pre-defined arrays of available shopping items
static ShoppingItem FruitsAndVegetables[] =
{
   { "Apples" },
   { "Lettuce" },
   { "Oranges" },
   { "Strawberries" },
   { 0 }
};

static ShoppingItem Dairy[] =
{
   { "Cheese" },
   { "Milk" },
   { "Orange Juice" },
   { "Yogurt"},
   { 0 }
};

static ShoppingItem Meats[] =
{
   { "Beef" },
   { "Chicken" },
   { "Fish"  },
   { "Turkey"},
   { 0 }
};

// array to hold selected items
static ShoppingItem s_Selected[13];

// count of how many items are selected thus far
static int s_iNumSelected;

// local helper functions
void MainFillMainList (FormPtr pForm);
void MainListDrawFunction (Int16 itemNum, RectanglePtr bounds,
                           char **pUnused);

void MainFillSelectedList (FormPtr pForm);
void SelectedListDrawFunction (Int16 itemNum, RectanglePtr bounds,
                               char **pUnused);
```

LISTING 10.1 Continued

```
char * MainGetAisleSelection (FormPtr pForm);
char * MainGetMainSelection (FormPtr pForm);

char * MainGetMainText (FormPtr pFrom, Int16 wSelect);

/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;

   switch (eventP->eType)
   {
      case menuEvent:
      {
         FormPtr pForm = FrmGetActiveForm ();
         handled = MainFormMenuHandler (pForm, eventP);
         break;
      }

      // note we grab popSelectEvent here to trap the pop-up trigger
      case popSelectEvent:
      case ctlSelectEvent:
      {
         // A control button was tapped and released
         FormPtr pForm = FrmGetActiveForm ();
         handled = MainFormButtonHandler (pForm, eventP);
         break;
      }

      case frmOpenEvent:
      {
         // main form is opening

         FormPtr pForm = FrmGetActiveForm ();

         MainFormInit (pForm);
```

LISTING 10.1 Continued

```
         FrmDrawForm (pForm);

         handled = true;
         break;
      }

      default:
      {
         break;
      }
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   pForm - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr pForm)
{
   // get a ptr to the main list
   UInt16 wIDMainList = FrmGetObjectIndex (pForm, MainMainList );
   ListPtr pCtlMainList = (ListPtr) FrmGetObjectPtr (pForm, wIDMainList);

   // get a ptr to the selected list
   UInt16 wIDSelectedList = FrmGetObjectIndex (pForm, MainSelectedList);
   ListPtr pCtlSelectedList = (ListPtr) FrmGetObjectPtr (pForm,
                                                      wIDSelectedList);

   // get a ptr to the aisle (pop-up) list
   UInt16 wIDAisleList = FrmGetObjectIndex (pForm, MainAisleList );
   ListPtr pCtlAisleList = (ListPtr) FrmGetObjectPtr (pForm, wIDAisleList);

   // set the combo box to select the first item
   LstSetSelection ( pCtlAisleList, 0 );

   // fill the main list based on the current aisle
   MainFillMainList (pForm);
```

LISTING 10.1 Continued

```
   // set the drawing function for the main list
   LstSetDrawFunction ( pCtlMainList, MainListDrawFunction );

   // fill the selected list
   MainFillSelectedList (pForm);

   // set the drawing function for the selected list
   LstSetDrawFunction ( pCtlSelectedList, SelectedListDrawFunction );
}

/*
   MainFillMainList:
   Fills the main list based upon the currently selected aisle
   Parms: pForm - pointer to main form
   Returns: none
*/
void
MainFillMainList (FormPtr pForm)
{
   // get a ptr to the main list
   UInt16 wIDMainList = FrmGetObjectIndex (pForm, MainMainList );
   ListPtr pCtlMainList = (ListPtr) FrmGetObjectPtr (pForm, wIDMainList);

   int iNumMainChoices = 0;

   // determine the current aisle
   char * pAisle = MainGetAisleSelection (pForm);

   // get the count of available main list items from the selected aisle
   if (0 == StrCompare (pAisle, "Fruits and Vegetables"))
   {
      iNumMainChoices =
        sizeof ( FruitsAndVegetables ) / sizeof ( ShoppingItem ) - 1;
   }
   else if (0 == StrCompare (pAisle, "Dairy"))
   {
      iNumMainChoices = sizeof ( Dairy ) / sizeof ( ShoppingItem ) - 1;
   }
   else if (0 == StrCompare (pAisle, "Meats"))
   {
      iNumMainChoices = sizeof ( Meats ) / sizeof ( ShoppingItem ) - 1;
   }
```

LISTING 10.1 Continued

```
   // fill the main list based on the combo selection
   LstSetListChoices ( pCtlMainList, NULL, iNumMainChoices );

   // redraw the list
   LstDrawList (pCtlMainList);

   return;
}

/*
   MainFillSelectedList:
   Fills the shopping selection list
   Parms: pForm - pointer to main form
   Returns: none
*/
void
MainFillSelectedList (FormPtr pForm)
{
   // get ptr to selected list
   UInt16 wID = FrmGetObjectIndex (pForm, MainSelectedList);
   ListPtr pList = (ListPtr) FrmGetObjectPtr (pForm, wID);

   // fill the selected list based on how many have been selected so far
   LstSetListChoices ( pList, NULL, s_iNumSelected );

   LstDrawList (pList);

   return;
}

/*
   MainListDrawFunction:
   Draws a single item in the main list
   Parms: itemNum = 0 based index of item to draw
          bound = ptr to rectangle providing drawing bounds
          pUnused = ptr to char string for item to draw. In this app, we
          never give the actual item data to the list, so we ignore this parm
   Returns: none
*/
void
MainListDrawFunction (Int16 itemNum, RectanglePtr bounds, char **pUnused)
{
```

LISTING 10.1 Continued

```
   FormPtr pForm = FrmGetActiveForm ();

   // get the text to be drawn by looking it up in the appropriate
   // aisle
   char * pText = MainGetMainText (pForm, itemNum);

   if ( pText )
   {
      // draw it
      WinDrawChars (pText, StrLen (pText), bounds->topLeft.x,
                    bounds->topLeft.y);
   }
   return;
}

/*
   SelectedListDrawFunction:
   Draws a single item in the selected list
   Parms: itemNum = 0 based index of item to draw
          bound = ptr to rectangle providing drawing bounds
          pUnused = ptr to char string for item to draw. In this app, we
          never give the actual item data to the list, so we ignore this parm
   Returns: none
*/

void
SelectedListDrawFunction (Int16 itemNum, RectanglePtr bounds, char **pUnused)
{
   FormPtr pForm = FrmGetActiveForm ();
   char * pText = NULL;

   // get the text to be drawn by looking it up in our selected list
   pText = s_Selected[itemNum].szItem;

   if ( pText )
   {
      // draw it
      WinDrawChars (pText, StrLen (pText), bounds->topLeft.x,
                    bounds->topLeft.y);
   }
   return;
}
```

LISTING 10.1 Continued

```
/*
   MainGetAisleSelection:
   Gets a ptr to the text representing the currently selected aisle
   Parms: pForm
   Returns: ptr to selected item (can be null)
*/

char *
MainGetAisleSelection (FormPtr pForm)
{
   UInt16 wID = FrmGetObjectIndex (pForm, MainAisleList);

   ListPtr pList = (ListPtr) FrmGetObjectPtr (pForm, wID);

   UInt16 wSelect = LstGetSelection (pList);

   char * pszSelect = LstGetSelectionText (pList, wSelect);

   return pszSelect;
}

/*
   MainGetMainSelection:
   Gets a ptr to the text representing the currently selected main list
   shopping item
   Parms: pForm
   Returns: ptr to selected item (can be null)
*/
char *
MainGetMainSelection (FormPtr pForm)
{
   char * pText = NULL;

   // get the selection index from the main list
   UInt16 wID = FrmGetObjectIndex (pForm, MainMainList);
   ListPtr pList = (ListPtr) FrmGetObjectPtr (pForm, wID);
   UInt16 wSelect = LstGetSelection (pList);

   // get the text at that index in the currently selected aisle
   pText = MainGetMainText (pForm, wSelect);

   return pText;
```

LISTING 10.1 Continued

```
}

/*
   MainGetMainText:
   Gets a ptr to the text representing an item at index wSelect
   in the main list
   Parms: pForm, wSelect
   Returns: ptr to item (can be null)
*/

char *
MainGetMainText (FormPtr pForm, Int16 wSelect)
{
   char * pText = NULL;

   // get the selected aisle
   char * pAisle = MainGetAisleSelection (pForm);

   // get the selected text based upon the aisle and index
   if (0 == StrCompare (pAisle, "Fruits and Vegetables"))
   {
      pText = FruitsAndVegetables[wSelect].szItem;
   }
   else if (0 == StrCompare (pAisle, "Dairy"))
   {
      pText = Dairy[wSelect].szItem;
   }
   else if (0 == StrCompare (pAisle, "Meats"))
   {
      pText = Meats[wSelect].szItem;
   }
   return pText;
}

/*
   MainFormMenuHandler:
   Handle a command sent to the main form.
   Parms:   pForm    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
```

LISTING 10.1 Continued

```
Boolean
MainFormMenuHandler (FormPtr /*pForm*/, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.menu.itemID)
   {
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   pForm    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormButtonHandler (FormPtr pForm, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      case MainAislePopTrigger:
      {
         // the aisle might have changed - refill the list
         MainFillMainList (pForm);

         break;
      }

      case MainAddButton:
      {
         // get a ptr to the main list
```

LISTING 10.1 Continued

```
         UInt16 wIDMainList = FrmGetObjectIndex (pForm, MainMainList );
         ListPtr pCtlMainList =
                (ListPtr) FrmGetObjectPtr (pForm, wIDMainList);

         // get the text from the main list
         char * pszSelect = MainGetMainSelection (pForm);

         if (pszSelect)
         {
            // add the item to the selected list
            StrCopy ( s_Selected[s_iNumSelected].szItem, pszSelect);

            // increment the number of selections
            s_iNumSelected++;
         }

         // refill selected list
         MainFillSelectedList (pForm);

         handled = true;
         break;
      }

   }
   return handled;
}
```

To make life easier, I created several helper functions that are called from various places in the code:

`MainFillMainList`	Fills the main list box with all the items that are available in the current "aisle."
`MainFillSelectedList`	Fills the shopping list "selected" list with all the items that have been chosen thus far.
`MainGetAisleSelection`	Returns the text of the currently selected aisle in the aisle pop-up list.
`MainGetMainSelection`	Returns the text of the currently selected item in the main list.

Armed with these functions, the rest of the form handling becomes much easier. Let's walk through the main areas of the code that support the shopping list form.

When the form is initialized, I take the opportunity to set the current aisle to the first one ("Fruits and Vegetables"), and then, I call `MainFillMainList`, which proceeds to determine the current aisle and fill the main list with available items. For the demonstration purposes of this simple application, the available items for each aisle are stored in static arrays at the top of the source file. `MainFillMainList` does not pass the text of the items into `LstSetListChoices`. Rather, it passes `NULL` and the total item count. As I've said, if you do this, you will then be responsible for drawing the items yourself. I take care of that by setting the main list's draw function as `MainListDrawFunction`. For completeness, I do the exact same initialization for the selection list (which starts out empty).

Take a look at the two draw functions, `MainListDrawFunction` and `SelectedListDrawFunction`. Each receives an item number representing the item to be drawn and a rectangle that serves to bind the drawing. For the main list, the item number serves as a lookup into the array of available items associated with the currently selected aisle. For the shopping list, the item number is a direct index into the array `s_Selected`. Given the text to draw, I simply call `WinDrawChars`, passing it the text string and the bounds rectangle. (I cover window drawing in more depth in Chapter 13, "Drawing Graphics.") Again, observe that I could have done anything I wanted with the item number; given how easy it is to code a draw function, I'll take the flexibility of making my own storage decision any day.

Now that the two lists know how to draw themselves (recall that the pop-up aisle list was prepopulated with items in the form resource and thus requires no special draw handling), the only work that remains is to handle changing the selected aisle and tapping the Add button.

Handling a change in aisle selection is easy, now that I've taken the trouble to code a generic routine that re-fills the main list based upon the current aisle. All I have to do is trap the `popSelectEvent` event and call `MainListFillList`. The old list contents are blown away and the list is re-filled.

Handling the Add button is not much more difficult. I trap the button event in the button handler and obtain the text associated with the currently selected main list item. I copy the text into the next available slot in the `s_Selected` array and increment the number of selected items. Then, I call `MainFillSelectedList` to blow away the previous list contents and re-fill from `s_Selected`.

Summary

In this chapter, you were able to review a fairly complex user interface using lists and pop-up lists. With a little bit of effort, you can put the ability to draw your own lists to work for you, resulting in a flexible display that changes based upon the users' selections.

The example presented here, although it's certainly more functional than the earlier button-based version, is still far from complete, and I obviously took some design shortcuts for the sake of clarity and brevity. Adding more functionality (a Remove button, a more dynamic means of storing a list, and so on) is left as an exercise for you.

11 Tables

IN THIS CHAPTER

Presenting large amounts of data is one of the hardest user-interface design problems to solve on a device with a limited display area. Inevitably, your application has more data than can be displayed at one time. The most likely circumstance under which this problem occurs is when you allow the users to browse your application's database records.

In this chapter, you'll learn about

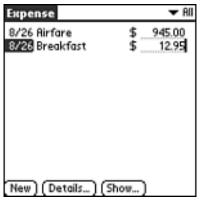
- What a table is and what it is used for
- What functionality is supported and not supported in a table
- When to use a table and when to use a list
- How to add support for tables in your program
- How to handle scrollbars

The chapter ends by presenting an example of a table in action.

What Is a Table?

A table object, in many ways, visually resembles a grid or spreadsheet interface on other platforms. It displays rows and columns of data and even lets you directly edit or manipulate cells (more on this later). Based on the interface and behavior, it would appear to be well suited for tabular data.

Figure 11.1 shows the main view of the Expense application, which employs a table to display your expense records, laid out with columns for the date, type, and amount.



8/26	Airfare	$	945.00
8/26	Breakfast	$	12.95

FIGURE 11.1 A table in the built-in Expense application.

Table Functionality

A table supports the display of a group of cells, organized as rows and columns of data. Each column can represent one of several supported data types: check box, date, time, label, pop-up trigger, numeric, text, text with a note, and narrow text. (Technically, cells in the same column of a table can even have different data types, but in practice, all cells of a column are set to the same type.)

Some data types can be edited and manipulated directly in their cells. Cells containing check boxes, pop-up triggers, and the various text types allow you to use graffiti to directly select and manipulate their contents.

In addition to the predefined data types, you can use a custom data type to render other kinds of data: custom drawing, bitmaps, and so on.

Although most examples of tables appear to have scroll bars, in fact scroll bars are a separate type of user-interface element: They have to be added separately. Even more significant, it is up to you as the programmer to coordinate scroll events with the appearance of your table.

Tables "support" vertical scrolling (although you'll see that this is not automatic). Tables do not have any support for horizontal scrolling. You cannot scroll columns left and right.

Also, note that there is no user-controlled column-manipulation support; columns cannot be moved or resized using the pen.

When to Use Tables Instead of Lists

On the surface, it would appear that tables are not much more than lists with support for columns. If columns aren't critical to your display needs, why not use a list, especially because (as you'll see) tables are much more complicated to create?

The Palm user-interface guidelines state that you should use lists for making a selection from a set of choices. Following these guidelines contributes to consistency both in your own applications and across all applications, resulting in a better experience for the users.

There's another reason, however. Because tables are only aware of the number of rows that they can display at one time, they are much more suited to browsing large numbers of data records. In this sense, they are a kind of "virtual list," like a window that hovers over your data, only showing you what is directly under the window. Lists, on the other hand, must be told how many items they will hold. Technically, you could tell a list it has 2,000 items and rely on the draw function callback to inform you which items must be drawn when. The issue is less black and white when you have, say, 20-odd items to display, and you don't expect significantly more items to be added.

As a rule of thumb, follow the Palm guidelines: Use lists for the presentation of small numbers of items, and use tables for situations where the number of items can grow large and is expected to be variable (as is the case with databases).

Adding Tables to Your Program

Adding a table to your program involves roughly the same steps as adding a list. You add the table to a form using Constructor and then add code as necessary to set up the table and use it in your program. Overall, my experience is that programming with tables is a complicated affair, and the topic of tables is not one of the subjects that the SDK documentation is particularly helpful with. Creating the table resource in Constructor is the easy part, so let's start there.

For such a complex object, adding a table resource to a form in Constructor is actually pretty easy. Simply drag it on to your form, and set the attributes in the Layout Properties pane. The only attributes of note are Editable, Rows, and the Column Widths:

- Editable—This attribute is not described in the Palm documentation for the table resource but is later described under the table objects section. If set, it allows the users to edit the table's contents, assuming that you've programmed the table to permit the users to manipulate its cells.
- Rows—This is the number of rows that you want to be visible in the table. Note that Constructor will adjust the height of the rows to squeeze them all into the height of the table you have specified. Set the number of rows to 50 or 2, and you'll see what I mean. For normal tables, I use a row height of 11 pixels as a guide. A table with a height of 121 pixels that fills most of the display but leaves space for buttons at the bottom of a form works out to 11 rows.
- Column Widths—Here's where you define how many columns you want in your table and how wide they should be. To add a column, highlight the

Column Width label in the Layout Properties pane and choose Edit, New Column Width. Widths are specified in pixels. Add as many columns as you need and do your best to guess how wide each column should be. Unfortunately, Constructor is not aware of the data types that will be used, nor the fonts, nor the width of the data, so it cannot offer more help in this area. Note that you must define all your columns here because there is no way to add them programmatically later.

As a programmer, you have a lot of responsibilities to handle when adding a table to a form. In the following sections, I break the somewhat daunting overall task into manageable chunks. Along the way, I explain why each task is necessary.

Table Initialization

When your form initializes, your job is to set up your table's row and column definitions so that they work correctly later, when it's time to draw them.

It's important to understand before you go any further that the number of rows a table contains has absolutely nothing to do with how many rows of data your program has. This is somewhat counter-intuitive. It is your job to track how far up or down the users have scrolled so that you can draw the correct rows of data into the fixed rows of the table.

During initialization, you need to set the data type associated with each cell for each row in the table. In practice, it's hard to imagine a situation where cells in the same column have different data types, so the following code is fairly standard:

```
UInt16 wRows = TblGetNumberOfRows (pTable);
for (i = 0; i < wRows; i++)
{
   TblSetItemStyle (pTable, i, colno, datatype);
   // Set other columns as well...
   ...
}
```

The column number is a zero-based value and increases for each column, left to right. Rather than embed these numbers everywhere, what I usually do is use `#defines` to assign "names" to my columns such as `COL_LASTNAME` and `COL_FIRSTNAME`. Then, I use the column names rather than the numeric values, making my code much more readable and maintainable.

In addition to setting the data type, you need to mark each column as usable because the default is false. Columns not marked as usable are invisible. You set this attribute by calling the `TblSetColumnUsable` function.

Like columns, rows also are marked either usable or unusable, with unusable rows being invisible (meaning they are not drawn). At this point in table initialization, you have not yet associated table rows with the data, so the safest thing to do is to set each row to be unusable using the `TblSetRowUsable` function. A good example of when you set a row to be unusable is if you have a table with 11 rows but only 10 rows of data. You set the 11th row to unusable.

The last task in setting up your table is to set any custom-handling functions for your columns. For columns of type `customTableItem`, you need to set a draw function using `TblSetCustomDrawProcedure` for the column. This function is a callback and is similar to what you saw in Chapter 10, "Giving the Users a Choice: Lists and Pop-up Triggers." It is called whenever a cell in that column needs to be drawn, passing you the row and column being referenced.

For columns using any of the text data types, which represent editable columns, you need to define a "load" procedure and a "save" procedure. Because cells in these columns essentially are field objects (see Chapter 9, "Labels and Fields"), it is your responsibility to allocate a memory handle for each cell and associate it with the cell when the users edit the data. You do this via `TblSetCustomLoadProcedure`, which is called when a cell is going to be drawn or is about to be edited. You also get the opportunity to capture changes when the users leave the cell. For this, you call `TblSetCustomSaveProcedure`, setting a callback function that will give you the handle to the cell's contents.

Probably the best advice regarding how to handle direct editing of text fields and load and save procedures is to carefully examine how the built-in applications do it. They use a couple of techniques. For example, the Expense application has an editable Amount field. However, it sets the column's style to custom and actually allocates a new field object on the fly when that area of the table is tapped.

The To Do application, however, uses the text style along with custom load and save procedures. In the load procedure, the actual record handle is retrieved and set as the text cell's memory handle, so in effect, the table's text field is synchronized with the database record field.

Table View Management

Because the table is an "n" row viewport into your data rows, you will have to track the current data row as well as the top-most visible item from your data.

Suppose you have an 11-row table, but you have 22 rows in your database. When you first load your table, you want rows 1 through 11 of your table to correspond with your database rows 1 through 11. If the users scroll down (I explain scrolling later) by one row, you need to redisplay the table such that it shows rows 2 through 12. You can do this most easily by tracking the top visible row along with the current selected row. Any time your table's viewport on the database changes (the user adds or deletes a record or the user scrolls up or down), you re-evaluate which 11 items need to be displayed.

This sounds like a mess, but the following boilerplate code pretty much takes care of things:

```
   UInt16 wIDTable = FrmGetObjectIndex (pForm, MainTableTable );
   TablePtr pTable = (TablePtr) FrmGetObjectPtr (pForm, wIDTable);

 // Get the number of rows defined in Constructor for the table.
 // This is the number that will be visible
   UInt16 wRows = TblGetNumberOfRows (pTable);

 // This all sets up what the top visible record
 // should be
   UInt16 uRecordNum = 0;

 // Is there a current record?
   if ( s_wCurrentRecord != -1)
   {
      if ( s_wTopVisibleRecord > s_wCurrentRecord )
      {
       // The current record is before the first visible record
       // Force the current record to be the first visible record
         s_wTopVisibleRecord = s_wCurrentRecord;
      }
      else
      {
       // This tries to seek from the top visible record to the
       // last visible record.
         uRecordNum = DmSeekRecordInCategory (pDB,
s_wTopVisibleRecord,
                                             wRows - 1,
                                             dmSeekForward,
                                             dmAllCategories  );

         if ( uRecordNum < s_wCurrentRecord)
         {
          // The last visible record in the table is less than the
          // current record (so the current record is after the visible
          // portion of the table). We need to reset the top
          // visible record such that it is the current record.
```

```
        // This is a "scroll down"
          s_wTopVisibleRecord = s_wCurrentRecord;
        }
      }
    }

  // Now adjust to make sure that we have a full number of records even
  // if it's the last screen. If there's less than a screen full left,
  // we have to push TopVisible backwards until there is a full screen left
    uRecordNum = dmMaxRecordIndex;

    if ( pDB )
    {
      DmSeekRecordInCategory ( pDB,
                               &uRecordNum,
                               wRows - 1,
                               dmSeekBackward,
                               dmAllCategories);
    }

    s_wTopVisibleRecord = min (s_wTopVisibleRecord, uRecordNum);
```

What this code does is use a couple of static variables—`s_wCurrentRecord` and `s_wTopVisibleRecord`—to set the topmost visible record in the table. It makes two checks to determine whether it needs to adjust the visible range of records so that the current record will display. First, if the current record is “above” the last visible row of the table, `s_wTopVisibleRecord` is set to be the current record. Otherwise, you seek forward 11 rows from the record that is currently the topmost visible to check whether the last existing record in the range encompasses the current record. If it doesn’t, again you set `s_wTopVisibleRecord` to be the current record. Finally, because it’s nice to always show a screen full of records where possible, if you are at the end of the list of records, you set `s_wTopVisible` to be the last record minus the number of rows.

Table Data Mapping

Once you know what range of records will be visible in the table, you need to either set the data for each cell so that the table can draw it or provide a way for the table to get at that data when it is needed.

The first time you get ready to display, and every time after that the values of the records or the range of visible records change, you will walk through the displayable database records and “link” them to the correct rows in the table. How you do this depends on the type of data you have associated with the cell.

For label, date, check box, and numeric cells, you can make the association by directly setting the value for the cell. Use `TblSetItemInt` or `TblSetItemPtr` to set the cell's value based on the matching database record's field.

For text cells, use `TblSetLoadDataProcedure` to associate a function that will be called when the text is drawn or edited. This function passes back a handle to the proper text value. The table takes care of rendering the text.

For custom cells, use `TblSetCustomDrawProcedure` to associate a callback function that will be called when the cell is about to be drawn. This function is responsible for rendering the contents of the cell, much the same way as the draw procedure for lists was.

There's an added step for custom cells: At draw time, you are only passed a row and column to be drawn. You need to determine which database record is associated with the row. Luckily, you can use the `TblSetItemInt` function to associate a two-byte value with each row. Although you can use this for anything, you typically will use this to store a record number or index into the database.

The following code illustrates how you walk through the rows of the table and verify that a database record exists to match each row. In the example, I've elected to set the two columns as custom types. Thus, I set a record number to be associated with the row:

```
UInt16 uRecordNum = s_wTopVisibleRecord;

// Load records into the table, starting at top visible
// Get the record number of the top visible record
uRecordNum = s_wTopVisibleRecord;

for (i = 0; i < wRows; i++, uRecordNum++)
{
   // Is there a record at this index?
   if (DmQueryRecord ( pDB, uRecordNum ))
   {
      // Yes there is.
      // Remember the associated record number for this row
      TblSetItemInt (pTable, i, COL_FISH_NAME, uRecordNum);
      TblSetItemInt (pTable, i, COL_FISH_DESC, uRecordNum);
      TblSetRowUsable (pTable, i, true);
   }
   else
   {
      // If we are out of records, mark row as unusable
      TblSetRowUsable (pTable, i, false);
   }
   // Force a row redraw
   TblMarkRowInvalid (pTable, i );
}
```

What data types should you associate with your columns and cells? The answer mostly depends on whether you will be providing direct editing of text within the table itself. If so, you will most likely want to use one of the text data types and rely on the `TblSetLoadDataProcedure` and `TblSetSaveDataProcedure` callbacks to set and save the contents of the cell. Even so, there is a lot of other handling you need to put in to trap when the users switch to a different row, pop up another form, and so on.

If you can get away without supporting direct cell editing, your life is much easier. In that case, you can use a combination of directly setting values for numbers, dates, and labels or you can keep everything consistent, set all cells to custom, and draw them yourself.

Table Drawing

For custom type cells, your draw procedure will be called each time the table needs to draw the cell. You are given a pointer to the table object, the row and column being drawn, and a bounding rectangle to draw in.

Handling the call is fairly straightforward:

```
void
MainListDrawFunction (VoidPtr pTable, Int16 row, Int16 column,
➥ RectanglePtr bounds)
{
   FormPtr pForm = FrmGetActiveForm ();

   Char *        pText;
   MemHandle      hRecord = NULL;
   UInt16 recordNum = 0;

   // Get the record num we are drawing
   recordNum = TblGetItemInt ((TableType *)pTable,
                              row,
                              column );

   // Find the record at this index
   hRecord = DmQueryRecord ( pDB, recordNum );
   if ( COL_LASTNAME == column)
   {
      // Get the last name from the record
      pText = ....
   }
   else
   {
      // Get another field from the record
      pText = ....
   }

   // Draw it
    if ( pText )
   {
       WinDrawChars ( pText, StrLen (pText), bounds->topLeft.x,
       ➥ bounds->topLeft.y);
   }
return;
}
```

Given the row's associated record number, it becomes a simple task to obtain the appropriate field from the record and draw it within the bounds supplied.

Given how much work you've seen so far that needs to go into supporting tables, it might be hard to believe there can be more, but there is one final thing to handle: scrolling.

You might have been dismayed when I noted at the beginning of the chapter that scrolling support was not part of the table object. Now that you understand the programming model for the table, it is clear that the table has no concept of how many data "rows" are associated with it and thus would not know how to correctly maintain the scroll positions. (I should point out that there are plenty of examples of data-bound grid controls and widgets that solve this problem, but you have to work with what you have.)

You have two options to add scroll handling to your table, using a scroll bar object or adding a set of up-arrow/down-arrow repeat buttons. The built-in applications use repeat buttons on their main table forms, so that's what I do. Note that before you dive into the next section, you might want to refresh your understanding of how to create the repeating arrow buttons by reviewing Chapter 8, "Button Controls."

As you did in Chapter 8, go back into Constructor and define two repeat buttons and place them near the bottom right of your table.

Remember that repeat buttons generate `ctlRepeatEvent` events. You'll need to trap those and update the `w_sTopVisible` variable appropriately. In addition, the Palm unit has physical scroll buttons. These generate special `keyDownEvent` events and should also update the `s_wTopVisible` variable. Here's some code that illustrates the necessary handling:

```
case ctlRepeatEvent:
       if (event->data.ctlRepeat.controlID == MainScrollUpRepeating)
    {
              // Scroll up
          DmSeekRecordInCategory ( pDB, &s_wTopVisible, 1,
          ➥dmSeekBackward, dmAllCategories);
              // Force a reload of records to the table
              .......
    }
       else if (event->data.ctlRepeat.controlID ==
       ➥MainScrollDnRepeating)
       {
              // Scroll down
             DmSeekRecordInCategory ( pDB, &s_wTopVisible, 1,
             ➥ dmSeekForward, dmAllCategories);
              // Force a reload of records to the table
              .......
       }
    break;
   case keyDownEvent:
       if (event->data.keyDown.chr == pageUpChr)
    {
              // Scroll up
          DmSeekRecordInCategory ( pDB, &s_wTopVisible, 1,
          ➥ dmSeekBackward, dmAllCategories);
              // Force a reload of records to the table
              .......
          handled = true;
    }
       else if (event->data.keyDown.chr == pageDownChr)
       {
              // Scroll down
             DmSeekRecordInCategory ( pDB, &s_wTopVisible, 1,
             ➥ dmSeekForward, dmAllCategories);
              // Force a reload of records to the table
              .......
          handled = true;
```

```
        }
      break;
```

Putting It All Together: FishTable

To illustrate all the programming steps you've made in this chapter, I've taken the Fish List program from Chapter 18, "Palm Databases and Record Management," and replaced the list object from that example with a table object. The fish database now has both a name and a description field, so I use two custom draw columns in the table. (For examples of how to employ the other data types in a table, I encourage you to explore the source code for the built-in applications.)

Figure 11.2 shows the main form for the new FishTable program, and Listing 11.1 contains the code.

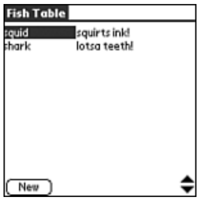


FIGURE 11.2 The table-driven fish database application.

LISTING 11.1 Source Code for the Main Form and the Fish Database Layer

```
/*
   TBLS_MAI.CPP
   Main form handling functions.
   Copyright (c) Bachmann Software and Services, 1999-2002
   Author: Glenn Bachmann
*/

// system headers
#include <PalmOS.h>
#include <SysEvtMgr.h>

// application-specific headers
#include "tables.h"
#include "tbls_res.h"
```

LISTING 11.1 Continued

```
static void    MainFormInit (FormPtr formP);
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);

// local helper functions
void MainListFill (FormPtr pForm);
void MainListDrawFunction (void * pTable, Int16 row, Int16 column,
                           RectanglePtr bounds);
void MainListCalcTopVisible (FormPtr pForm);

Boolean AddFish ( void );
Boolean AddFish_EventHandler (EventPtr pEvent);

// current record in the table
static Int16 s_wCurrentRecord;
static Int16 s_wTopVisibleRecord;

#define COL_FISH_NAME   0
#define COL_FISH_DESC   1

/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr pEvent)
{
   Boolean  handled = false;

   switch (pEvent->eType)
   {
      case menuEvent:
      {
         FormPtr pForm = FrmGetActiveForm ();
         handled = MainFormMenuHandler (pForm, pEvent);
         break;
      }
```

LISTING 11.1 Continued

```
    case ctlSelectEvent:
    {
       // A control button was tapped and released
       FormPtr pForm = FrmGetActiveForm ();
       handled = MainFormButtonHandler (pForm, pEvent);
       break;
    }

    case tblSelectEvent:
    {
       // set the current record based on user selection

       s_wCurrentRecord = TblGetItemInt (
                         (TableType *)pEvent->data.tblSelect.pTable,
                          pEvent->data.tblSelect.row,
                          pEvent->data.tblSelect.column );
       break;
    }

    // physical scroll buttons
     case keyDownEvent:
    {
     if (pEvent->data.keyDown.chr == pageUpChr)
                 {
          FormPtr pForm = FrmGetActiveForm ();
          s_wTopVisibleRecord =
                       FishGetPrevRecordNum ( s_wTopVisibleRecord, 1 );
          MainListFill ( pForm );
          FrmDrawForm ( pForm );
           handled = true;
                 }
          else if (pEvent->data.keyDown.chr == pageDownChr)
                {
          FormPtr pForm = FrmGetActiveForm ();
          s_wTopVisibleRecord =
                       FishGetNextRecordNum ( s_wTopVisibleRecord, 1 );
          MainListFill ( pForm );
          FrmDrawForm ( pForm );
           handled = true;
       }
       break;
    }
```

LISTING 11.1 Continued

```
      case frmOpenEvent:
      {
         // main form is opening

         FormPtr pForm = FrmGetActiveForm ();

         MainFormInit (pForm);

         FrmDrawForm (pForm);

         handled = true;
         break;
      }

      default:
      {
         break;
      }
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   pForm - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr pForm)
{
   // set the controls to an initial state
   s_wCurrentRecord = -1;

   // walk through fish records and fill list
   MainListFill ( pForm );
}

/*
   MainFormMenuHandler:
   Handle a command sent to the main form.
```

LISTING 11.1 Continued

```
   Parms:   pForm    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormMenuHandler (FormPtr /*pForm*/, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.menu.itemID)
   {
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   pForm    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/

Boolean
MainFormButtonHandler (FormPtr pForm, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      case MainNewButton:
      {
         if ( AddFish () )
         {
            MainListFill ( pForm );
```

LISTING 11.1 Continued

```
            FrmDrawForm ( pForm );
         }

         handled = true;
         break;
      }

       case MainScrollUpRepeating:
      {
         s_wTopVisibleRecord =
                   FishGetPrevRecordNum ( s_wTopVisibleRecord, 1 );
         MainListFill ( pForm );
         FrmDrawForm ( pForm );
         break;
      }
         case MainScrollDownRepeating:
      {
         s_wTopVisibleRecord =
                    FishGetNextRecordNum ( s_wTopVisibleRecord, 1 );
         MainListFill ( pForm );
         FrmDrawForm ( pForm );
         break;
      }


      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainListFill:
   Fills the fish list
   Parms: pForm - pointer to main form
   Returns: none
*/
void
MainListFill (FormPtr pForm)
```

LISTING 11.1 Continued

```
{
   // get a ptr to the fish table
   UInt16 wIDFishTable = FrmGetObjectIndex (pForm, MainFishTable );
   TablePtr pTable = (TablePtr) FrmGetObjectPtr (pForm, wIDFishTable);

   // init the table

   // get the number of rows defined in Constructor for the table
   // this is the number that will be visible
   UInt16 wRows = TblGetNumberOfRows (pTable);
   UInt16 i;

   for (i = 0; i < wRows; i++ )
   {
      // set the cells to have custom drawing
      TblSetItemStyle  (pTable, i, COL_FISH_NAME, customTableItem );
      TblSetItemStyle  (pTable, i, COL_FISH_DESC, customTableItem );

      // set the row to be unusable to start
      TblSetRowUsable (pTable, i, false);
   }

   // set the columns to be usable
   TblSetColumnUsable  (pTable, COL_FISH_NAME, true );
   TblSetColumnUsable  (pTable, COL_FISH_DESC, true );

   // set our draw procedure callback for each column
  TblSetCustomDrawProcedure (pTable, COL_FISH_NAME, MainListDrawFunction );
  TblSetCustomDrawProcedure (pTable, COL_FISH_DESC, MainListDrawFunction );

   // reset the top visible row based on size of table and current record
   MainListCalcTopVisible (pForm);

   UInt16 uRecordNum = 0;

   // load records into the table, starting at top visible
   // get the record number of the top visible record
   uRecordNum = s_wTopVisibleRecord;

   for (i = 0; i < wRows; i++, uRecordNum++)
   {
```

LISTING 11.1 Continued

```
      UInt16 uNextRecordNum = 0;

      if (FishQueryFish ( uRecordNum ))
      {
         // remember the associated record number for this row
         TblSetItemInt (pTable, i, COL_FISH_NAME, uRecordNum);
         TblSetItemInt (pTable, i, COL_FISH_DESC, uRecordNum);
         TblSetRowUsable (pTable, i, true);
      }
      else
      {
         // if we are out of records, mark row as unusable
         TblSetRowUsable (pTable, i, false);
      }
      // force a row redraw
      TblMarkRowInvalid (pTable, i );
   }

   return;
}

/*
   MainListCalcTopVisible:
   Sets the top visible item in the table
   Parms: none
   Returns: none
*/
void
MainListCalcTopVisible (FormPtr pForm)
{
   // get a ptr to the fish table
   UInt16 wIDFishTable = FrmGetObjectIndex (pForm, MainFishTable );
   TablePtr pFishTable = (TablePtr) FrmGetObjectPtr (pForm, wIDFishTable);

   // get the number of rows defined in Constructor for the table
   // this the number that will be visible
   UInt16 wRows = TblGetNumberOfRows (pFishTable);

   // this all sets up what the top visible record
   // should be
   UInt16 uRecordNum = 0;
```

LISTING 11.1 Continued

```
   if ( s_wCurrentRecord != -1)
   {
      if ( s_wTopVisibleRecord > s_wCurrentRecord )
      {
         // the current record is before the first visible record
         // "scroll up"
         s_wTopVisibleRecord = s_wCurrentRecord;
      }
      else
      {
         // this tries to seek from the top visible record to the
         // last visible record.
         uRecordNum = FishGetNextRecordNum ( s_wTopVisibleRecord,
                                             wRows - 1 );

         if ( uRecordNum < s_wCurrentRecord)
         {
            // the last visible record in the table is less than the
            // current record (so the current record is after the visible
            // portion of the table). We need to reset the top
            // visible record so that it is the current record.
            // this is a "scroll down"
            s_wTopVisibleRecord = s_wCurrentRecord;
         }
      }
   }

   // now adjust to make sure that we have a full number of records even
   // if it's the last screen. If there's less than a screenful left,
   // we have to push TopVisible backwards until there is.
   uRecordNum = FishGetPrevRecordNum ( dmMaxRecordIndex, wRows - 1);

   s_wTopVisibleRecord =
    (s_wTopVisibleRecord < uRecordNum) ? s_wTopVisibleRecord : uRecordNum;
   return;
}

/*
   MainListDrawFunction:
   Draws a single cell in the main table
   Parms: pTable = table ptr
          row = row of item
```

LISTING 11.1 Continued

```
          column = cell
          bounds = bounding rectangle for cell
   Returns: none
*/
void
MainListDrawFunction (void * pTable, Int16 row, Int16 column,
                      RectanglePtr bounds)
{
   FormPtr pForm = FrmGetActiveForm ();

   char *         pText;
   MemHandle      hFish = NULL;
   UInt16 recordNum = 0;

   // get the record num we are drawing
   recordNum = TblGetItemInt ((TableType *)pTable,
                              row,
                              column );

   // find the record at this index
   FishGetFish ( recordNum );

   if ( COL_FISH_NAME == column)
   {
      // get the fish name
      pText = FishGetName ();
   }
   else
   {
      // get the fish desc
      pText = FishGetDesc ();
   }

   // draw it
   if ( pText )
   {
    WinDrawChars ( pText, StrLen (pText),
                   bounds->topLeft.x, bounds->topLeft.y);
   }

   // release it
   FishReleaseFish ();
```

LISTING 11.1 Continued

```
   return;
}


/*
   AddFish:
   Invokes the Add Fish popup form
   Parms:   none
   Return:  none
*/
Boolean AddFish ( void )
{
   // initialize the form
   FormPtr  pForm = FrmInitForm (AddFishForm);

   // create a new fish record
   if ( FishNew () )
   {
      // edit it!
      MemHandle hRecord = FishGetFish ( 0 );

      // create a mem buffer for the new fish name
      MemHandle hName = MemHandleNew ( FISH_NAME_LEN + 1 );
      char * pName = (char *)MemHandleLock (hName);
      // default the name of the fish
      StrCopy ( pName, "<new fish>" );
      MemPtrUnlock ( pName );

      // create a mem buffer for the new fish description
      MemHandle hDesc = MemHandleNew ( FISH_DESC_LEN + 1 );
      char * pDesc = (char *)MemHandleLock (hDesc);
      // default the desc of the fish
      StrCopy ( pDesc, "" );
      MemPtrUnlock ( pDesc );

      UInt16 wIDName = FrmGetObjectIndex ( pForm, AddFishNameField );
      FieldPtr pNameField = (FieldPtr) FrmGetObjectPtr ( pForm, wIDName);
      // copy the text into the edit field
      FldSetTextHandle ( pNameField, (MemHandle)hName);
```

LISTING 11.1 Continued

```
      UInt16 wIDDesc = FrmGetObjectIndex ( pForm, AddFishDescField );
      FieldPtr pDescField = (FieldPtr) FrmGetObjectPtr ( pForm, wIDDesc);
      // copy the text into the edit field
      FldSetTextHandle ( pDescField, (MemHandle)hDesc);

      // display the dialog
      if ( FrmDoDialog (pForm) )
      {
         // get the name from the dialog
         hName = FldGetTextHandle (pNameField);
         char * pName = (char *)MemHandleLock (hName);
         hDesc = FldGetTextHandle (pDescField);
         char * pDesc = (char *)MemHandleLock (hDesc);

         // set it in the record
         FishSetName ( pName );
         FishSetDesc ( pDesc );

       MemPtrUnlock ( pName);
       MemPtrUnlock ( pDesc);
      }

      // un-edit the fish record, saving changes
      FishReleaseFish ();

      // destroy the form
      FrmDeleteForm (pForm);

      return true;
   }
   return false;
}
```

Tables and Databases

Although tables are not specifically tied to databases, it is extremely common to see tables used as a visual method for presenting a scrollable list of database records. Although the example code in this chapter does make use of Database Manager functions before the topic of databases has been properly introduced in Chapters 17 and 18, the two concepts are very closely tied together in Palm programming, so it is important to see how they are used in conjunction with each other.

For an in-depth exploration of databases in Palm programming, refer to Chapters 17, “Understanding Palm OS Databases,” and 18, “Palm Databases and Record Management.”

Summary

Tables are by far the most complex user-interface element in terms of programming effort. If you need to manage the display and manipulation of databases so that your users can easily browse and edit their data, tables are the right choice. I hope this chapter provided enough background and sample code to help you add table support to your application with minimum frustration.

12 Menus

IN THIS CHAPTER

On most modern graphical user interfaces, menus are one of the primary means of providing the users with the ability to control a program's execution. One of the reasons this is so is that menus are generally ubiquitous and thus most computer users are familiar with them: Virtually every Windows application has at least a main menu.

On the Palm, things are a bit different. Palm users place a premium on being able to control their applications with a minimum of pen tapping and user-interface navigating. For a true PDA user on-the-go, tapping a command button once with a pen (or a fingertip) is infinitely preferable to tapping on a menu to have it drop down, finding the menu command you want, and tapping again to execute it. The dexterity and patience level required for menus on mobile handheld devices is surprisingly high.

To illustrate my point, as a fun exercise, stand up, put a briefcase or large book in one hand, your PDA in the other hand, start walking, and attempt to open Address Book and change the sort order of your contacts from "Sort By Name" to "Sort By Company," without stopping or putting down your briefcase. This is how real PDA owners will be using your software program. Get the picture?

Something else that might surprise those of you who are more experienced PDA users is that a hefty number of new or even moderately experienced people have no idea that there are even menus on Palm applications. I was asked once by a friend who had been a PDA user for two years how he could make the game software he was playing stop and start a new game. I asked him if he had tapped the menu and looked for a menu command. He said, "Tap the what?" "You know, the menu," I replied. "That little icon to the left of the graffiti area on your Palm." He tapped the icon, a menu appeared, and behold, there was a New Game menu item there!

Despite their challenges, menus do have an important role to play in many Palm applications, and there is support in the Palm developer toolset to create simple menu systems. In this chapter, you will come to understand the following:

- How menus are used in the Palm environment
- Guidelines for menu usage in your application
- How to create menu resources
- How to respond to menu events within your application

What Is a Menu?

In general, menus provide a means for users to control an application's execution by exposing a number of commands corresponding to individual menu items. As I've said, on most platforms menus are one of the main ways to provide this capability through the user interface. The Palm user interface de-emphasizes the importance of the menu system, preferring on-screen buttons as the main command interface. In fact, the menu system is not even visible on Palm applications unless the special menu icon is tapped.

The menu system is usually not visible. Combine that with the fact that it requires multiple pen taps to navigate to a menu item, and it becomes clear that you should design your user interface so that the most important commands in your application are available using one-tap methods, such as buttons, lists, and so on. Good candidates for options that should be placed on menus are preferences dialogs, Clipboard commands, and database operations.

The first time you tap a menu button to display an application's menu, the Palm OS automatically displays the top-level menu bar. Every time the menu button is tapped subsequently, it will return to the last displayed menu. This is a helpful convenience that aids in executing the most common commands. Note that once you switch to a different application, Palm OS once again reverts to displaying only the top-level menu names. After a menu command is executed or the pen is tapped outside the menu area, the menu bar is dismissed.

Menus in Action on the Palm

Figure 12.1 shows the main menu for the To Do application on the Palm in response to a tap of the menu button.

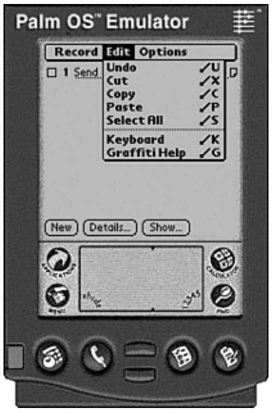


FIGURE 12.1 The To Do main menu.

Menu Nomenclature

Before you go further, it is important to understand the terms used in the Palm documentation to refer to various menu constructs. This will be especially crucial as you walk through how to create menus in Constructor, which relies heavily on understanding these terms. As you've seen in other chapters, some of the Palm terms can be confusing. Furthermore, some of the terms have different meanings from the same names on other platforms, which I view as significant because most of us come to Palm programming from programming on other computer platforms.

The following definitions should help clarify the main concepts used in the Palm documentation (see Figure 12.2 for an example of each term):

- **Menu bar**—The menu bar displays the names of each available drop-down menu for the current application view. Note that applications might define different menu bars for different views or forms.
- **Menu name**—A menu name is the text that appears in the menu bar. When a menu name is tapped, the associated drop-down menu is displayed.
- **Menu**—This is what drops down below the menu name when it is tapped. The menu is a container for a list of menu items.
- **Menu item**—A menu item is what is mapped to an actual application command. Menu items can have optional graffiti shortcuts associated with them.
- **Shortcut**—Although not unique to menus, shortcuts are special graffiti strokes that provide quick access to an application function. You invoke a shortcut by

using the special graffiti command stroke (a diagonal move, starting at the bottom left and ending at the top right of the graffiti area), followed by the unique letter associated with the command. The application then responds just as if you chose a menu item.

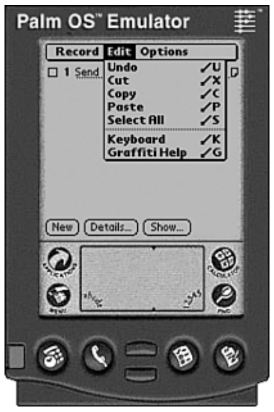


FIGURE 12.2 The elements of a menu.

Guidelines for Menu Usage

The Palm user-interface design guides make several recommendations regarding the use of menus in an application. Where applicable, strive for consistency with the built-in applications by using the same menu structure, menu item names, and shortcuts. Provide shortcuts for each menu item. If you have a menu command in common with other applications, use the same shortcut for consistency. A good example of this is the Beam command found in many applications.

A quick survey of the Palm built-in applications yields the following common menu items shown in Table 12.1.

TABLE 12.1 Common Menu Items and Their Shortcuts

Menu	Item Name	Shortcut
Record	New	N
	Delete	D
	Beam	B
Edit	Undo	U
	Cut	X
	Copy	C

TABLE 12.1 Continued

Menu	Item Name	Shortcut
Edit	Paste	P
	Select All	S
	Keyboard	K
	Graffiti Help	G
Options	Font	F
	Preferences	R

As a general rule, do not gray out unusable menu items; remove them instead. This rule requires some explanation because there is no way to dynamically add or remove items from a menu. Instead, if your application has modes in which certain commands become unavailable, you need to define multiple versions of your menu resource, one that has the command and one that doesn't, and then manually swap them in and out within your program. An easier alternative is to keep one menu system that contains all the possible commands and simply pop up an alert to the user explaining why the command is unavailable in the current mode.

Use separators to logically group related menu items and distinguish them from other menu items in the same menu. Make the standard copy, cut, and paste Clipboard commands available whenever you have a form containing editable fields. In a form with editable fields, you should also make the standard system keyboard dialog and graffiti reference available from your Edit menu. Do not use your menu system to replicate commands that are available on-screen through buttons.

Using Constructor to Create Menus

Constructor is criticized (sometimes unfairly) for having an awkward user interface. When creating menu resources, I am forced to agree with the critics: Creating menu resources in Constructor is an awkward, error-prone adventure for the developer. Fortunately, you will not often edit menu resources during the development life cycle.

Figure 12.3 shows the main Constructor windows involved in creating a menu resource.

The following steps outline how you use Constructor to create a menu bar and associate it with a form:

1. Open your application's resource in Constructor.
2. Create a new menu bar resource in one of two following ways:
    - Click the menu bar's resource type in the main project and choose Edit, New Menu Bar Resource from the main Constructor menu. Type a name

for the menu bar resource, and double-click it to open the menu bar resource window.

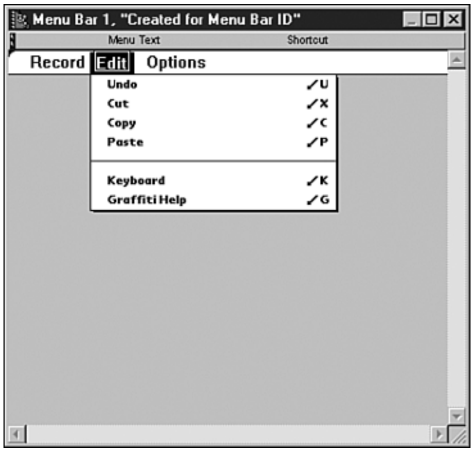


FIGURE 12.3 Creating a menu bar in Constructor.

- Edit the form you want to associate the menu bar with, and in the Layout Properties Menu Bar ID field, type a unique ID (such as 1). When you press Enter or click outside of the ID field, the Create button to the right of the ID field activates. Click the Create button to open the menu bar resource window.

3. The initial display of the menu bar resource window is not terribly helpful in guiding you to your next step. What you need to do is define your first menu, which will have a menu name—the top-level text that will appear on your menu bar. To do this, choose Edit, New Menu from the Constructor menu. Recall that the entire menu system is called a menu bar, and the individual drop-down lists are called menus.
4. Enter a menu name for the new menu created in step 3.
5. Create the first menu item for the current menu by choosing Edit, New Menu Item from the Constructor menu. Type the text that should appear on the menu item.
6. Repeat step 5 for each menu item you want to associate with the current menu. If you want to create a separator to logically group items, choose Edit, New Separator Item.
7. Repeat steps 3–6 for each drop-down menu that should appear on your menu bar.

8. For each menu item that you want to associate with a shortcut, you can double-click the menu item to open the Property Inspector window. In the Shortcut key field, enter the single letter to be associated with the menu item.
9. When you are happy with the menu bar, save your resource file.
10. If you created your menu bar resource by starting from a form, your work is done; the new menu bar is associated with the form. If you created your menu bar resource as a standalone menu bar, you should edit the form you want to associate with the menu bar and enter the ID of the menu bar in the Menu Bar ID field.

It's been my experience during this process that it is easy to make mistakes that are not so easily fixed. I am in the habit of saving my work in Constructor often, so I can always abort my current session and return to the last correct version.

I've also learned that it is prudent to design your menu system on paper first and use Constructor to enter the final design once it is solid, rather than interactively play with the menu appearance in Constructor.

If you look at the main project window in Constructor, you will notice that Constructor creates two kinds of resources as a result of your session. The first is a menu bar resource—the ID that you associate with the form. The individual drop-down menus are created as separate menu resources under the menus resource type. If you need to change any part of your menu system, you can either navigate down from the menu bar resource or go directly to the menu resource.

Handling Menu Events in Your Application

You handle menus in your application in much the same way as buttons or other user-interface objects. Like most user-interface objects, menus generate events that are routed to your form's event-handler procedure. Specifically, when the user selects a menu item (through a pen tap or a shortcut), you receive an event of type `menuEvent`.

If you determine in your event handler that you have received a menu event, you can examine the event's `data.menu` pointer to see which menu item was selected. The menu structure has many data members, but the main item of interest is the `itemID` field, which contains the ID of the menu item that was selected by the user. The IDs for your menu system, like all other application resources, are created and stored automatically in your application's resource header file.

To help you see this as it is implemented in actual code, the following snippet is from the example program accompanying this chapter:

```
Boolean
MainFormMenuHandler (FormPtr /*formP*/, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.menu.itemID)
   {
      case RecordNew:
      {
         PalmMessageBox (MessageBoxAlert, "New Record");
         break;
      }
      case RecordDelete:
      {
         PalmMessageBox (MessageBoxAlert, "Delete Record");
         break;
      }
…handle other menu item ids…
```

Given the selected menu ID, you can take the appropriate action from your event handler and return `TRUE` to the Palm OS, indicating that you have handled the event.

To understand this process more fully, you may want to examine the complete main form handler in the sample application. It traps menu selections and simply displays a message box indicating which selection was made. For the vast majority of applications, this is the only type of programming you need to add to handle menus. Figure 12.4 shows an application in action.

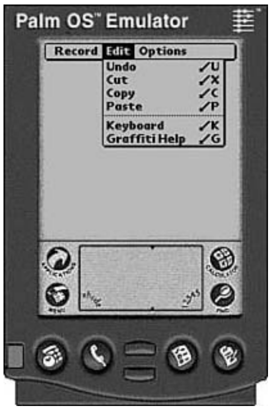


FIGURE 12.4 The menus application.

Listing 12.1 shows the code for the application's main form.

LISTING 12.1 The Main Form Code for the Menus Application

```
/*
   MENU_MAI.CPP
   Main form handling functions.
   Copyright (c) 1999-2002 Bachmann Software and Services
   Author: Glenn Bachmann
*/

// system headers
#include <PalmOS.h>
#include <SysEvtMgr.h>

// application-specific headers
#include "menus.h"      // common app header
#include "menu_res.h"   // resource ids

static void    MainFormInit (FormPtr formP);
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);

Int16 PalmMessageBox ( Int16 wAlertID, char * pMessage );

/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;

   switch (eventP->eType)
   {
      case menuEvent:
      {
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormMenuHandler (formP, eventP);
         break;
```

LISTING 12.1 Continued

```
      }

      case ctlSelectEvent:
      {
         // A control button was tapped and released
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormButtonHandler (formP, eventP);
         break;
      }

      case frmOpenEvent:
      {
         // main form is opening

         FormPtr formP = FrmGetActiveForm ();

         MainFormInit (formP);

         FrmDrawForm (formP);

         handled = true;
         break;
      }

      default:
      {
         break;
      }
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   formP - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr formP)
{
   // set the controls to an initial state
```

LISTING 12.1 Continued

```
}

/*
   MainFormMenuHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormMenuHandler (FormPtr /*formP*/, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.menu.itemID)
   {
      case RecordNew:
      {
         PalmMessageBox (MessageBoxAlert, "New Record");
         break;
      }
      case RecordDelete:
      {
         PalmMessageBox (MessageBoxAlert, "Delete Record");
         break;
      }
      case RecordPurge:
      {
         PalmMessageBox (MessageBoxAlert, "Purge");
         break;
      }
      case EditUndo:
      {
         // get the currently active field object, and call the FldUndo
         // api
         PalmMessageBox (MessageBoxAlert, "Undo");
         break;
      }
      case EditCut:
      {
         // get the currently active field object, and call the FldCut
```

LISTING 12.1 Continued

```
            // api
            PalmMessageBox (MessageBoxAlert, "Cut");
            break;
         }
         case EditCopy:
         {
            // get the currently active field object, and call the FldCopy
            // api
            PalmMessageBox (MessageBoxAlert, "Copy");
            break;
         }
         case EditPaste:
         {
            // get the currently active field object, and call the FldPaste
            // api
            PalmMessageBox (MessageBoxAlert, "Paste");
            break;
         }
         case EditKeyboard:
         {
            // invoke the keyboard dialog, with the default keyboard
            // NOTE: this only works if there is a field object with the focus!
            // SysKeyboardDialog (kbdDefault);
            PalmMessageBox (MessageBoxAlert, "Keyboard");
            break;
         }
         case EditGraffitiHelp:
         {
            // invoke the graffiti ref dialog, with the default reference
            SysGraffitiReferenceDialog (referenceDefault);
            break;
         }
         case OptionsAboutMenus:
         {
            // invoke the graffiti ref dialog, with the default reference
            PalmMessageBox (MessageBoxAlert, "About");
            break;
         }
         default:
         {
            handled = false;
            break;
```

LISTING 12.1 Continued

```
      }
   }
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormButtonHandler (FormPtr formP, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   PalmMessageBox:

   Toolbox api providing a general purpose message box function

   Parms: wAlertID = id of alert to show
          pMessage = ptr to message text to display

   Returns: id of button chosen by user
*/

Int16
PalmMessageBox ( Int16 wAlertID, char * pMessage )
```

LISTING 12.1 Continued

```
{
   Int16 err;

   err = FrmCustomAlert (wAlertID, pMessage, "", "" );

   return err;
}
```

The action starts in `MainEventHandler`, where we trap events of type `menuEvent`. We route these events to the menu event handler `MainFormMenuHandler`, passing it a pointer to the form object as well as a pointer to the event itself.

Inside `MainFormMenuHandler`, we switch on the menu object's `itemID` field and trap each menu item ID that we are interested in. In this basic application, for the most part, we simply call the handy `PalmMessageBox` function to alert the user about which menu item was selected.

I have also provided some placeholders for handling common menu items related to the Clipboard and field objects. If you have editable field objects on your form, you should support an undo operation, cut, copy, and paste Clipboard operations, as well as the keyboard dialog and graffiti help.

Although you do have to write specific code to handle undo and Clipboard operations, the code is boilerplate and easy to write. Built-in Palm functions directly correspond to each operation. When you receive an undo, cut, copy, or paste menu item, you should determine the form object that currently has the input focus (if any) by performing the following calls:

```
FrmGetActiveForm
FrmGetFocus
```

If an object on the form has the input focus (in which case `FrmGetFocus` returns something other than `noFocus`), you can determine the object's type by calling `FrmGetObjectType`. If the object type is `frmFieldObj`, call `FrmGetObjectPtr` to obtain the currently active field object.

If a field object has the input focus, you pass the object's pointer to the corresponding Palm field function, as in Table 12.2.

TABLE 12.2 Clipboard-Related Field Functions

Operation	**Field Function**
Undo	`FldUndo`
Cut	`FldCut`
Copy	`FldCopy`
Paste	`FldPaste`

It's even easier to handle a request for the on-screen keyboard: Call the Palm function `SysKeyboardDialog()`. Note that this function will work only if it senses there is an active field object to type into.

To handle a request for graffiti help, once again Palm provides the: `SysGraffitiReferenceDialog()`.

Summary

This chapter essentially provides all the information most developers will ever need to know to incorporate menus into their applications. Although some aspects of designing the user interface can be cumbersome, menu handling is fairly straightforward once you work out all of the naming conventions and tool usage issues.

Although the Palm developer toolkit features more advanced techniques to manually load menus from resources and dispose of them, the built-in applications do not use them, and the vast majority of developers can get what they need without ever calling them.

13 Drawing Graphics

IN THIS CHAPTER

- The Palm Display
- Palm OS Graphics Primitives
- The Palm OS Window Model
- PalmDraw: A Graphics-Drawing Demonstration Program

Most Palm applications take advantage of the forms functionality (described in Chapter 6, “Interacting with the User: Forms”) and other Palm user-interface resource types in presenting and managing their appearance. However, for specialized functionality, some developers find the need to perform more custom user interface effects. In these situations, it can be more convenient to bypass fields, labels, and such, and render text output directly on the device display. Furthermore, display of non-textual *graphics primitives* is sometimes required.

This chapter covers

- The characteristics of the Palm display
- Graphics primitives such as lines, rectangles, and bitmaps
- The Palm OS windowing model
- Multiple window issues

The Palm Display

For the first few years of the existence of the Palm OS, the physical device displays that were available were limited to 160 pixels wide by 160 pixels high and were black and white (no color or grayscale). There are now several Palm OS based handheld devices that break the 160×160 form factor, including units from Handera, Sony, and Alphasmart. It seems safe to assume that handheld displays will continue to improve in resolution and quality, and that we will also see the Palm OS being incorporated into devices that have “non-traditional” form factors.

Screen Sizes

It is now clear that it is no longer possible to target only 160×160 screen sizes if you want to create applications that take advantage of the universe of handheld devices available on the market. Thus far, devices with expanded displays have generally offered a "compatibility" mode in which they can provide a reasonable screen display for applications that were hard-coded to 160×160 dimensions, but some applications work better than others in those modes.

Also, the new screen dimensions offer definite opportunities for some applications to deliver significantly more functionality for owners of these enhanced devices. For example, a photo-viewing program can display a much larger and sharper image than before. Of course, to take advantage of the increased display size, you have to write special code to detect the screen size available to the program at runtime.

All this makes for an interesting challenge to Palm software developers. On the one hand, if you want to create an application that is extremely easy to use (Zen of Palm!) and appeals to a wide audience, it is critical to account for the likelihood of a 160×160 screen size. On the other hand, expanded screen sizes mean some applications have an opportunity to work better and provide more value. The owners of such devices appreciate applications that take special advantage of these screen sizes.

The Palm SDK documentation does offer functions for obtaining the screen dimensions programmatically (indeed Palm has long warned that one could assume new display sizes would be coming). In the end, my advice is to carefully consider your application's functionality, your target users, and whether it is worthwhile to offer high-resolution display support in your application. In my experience, forms-based applications generally do not scale well anyway, so they are probably not worth the effort. On the other hand, imaging applications, drawing programs, and programs that display and edit files such as documents, spreadsheets, and Web pages have much to gain from high-resolution screens.

Color and Grayscale

The first Palm OS devices were exclusively black and white. This is no longer the case: In fact pure black and white devices are rapidly going the way of the dinosaur, and new devices tend to be either sharp grayscale or color.

As in the previous discussion of screen sizes, developers face an increasingly complex set of choices as to what level of color or grayscale support to recognize in their applications. Once again, the general rule I follow is that forms-oriented applications need not be concerned with such issues, but other applications can deliver significant benefits to users of color devices.

This chapter does touch upon some functions that make use of color and grayscale. However, Chapter 15, "Adding Color to Palm Applications," covers color and grayscale programming in more depth.

Palm OS Graphics Primitives

The Palm OS supports a set of basic, raw functions for rendering basic graphics elements (called *primitives*) on the Palm display. At present, the level of support is rather minimal—don't look for functions that draw circles, curved lines, polygons, or triangles. They don't exist, although you could certainly (with some effort) build these capabilities yourself.

Basically, you can use the Palm SDK to draw lines, rectangles, text, and bitmaps. Let's examine how to use each of these capabilities.

Lines

The following functions provide line-drawing support:

```
WinDrawLine
WinDrawGrayLine
WinEraseLine
WinFillLine
```

`WinDrawLine` is the most basic function. You pass `WinDrawLine` the x,y coordinates of two points: the start of the line and the end of the line. Lines drawn using `WinDrawLine` are a single pixel in width and are drawn in black. In fact, lines drawn by any of these functions are a single pixel in width. If you need to draw thicker lines, you must call `WinDrawLine` twice, shifting the coordinates for the second call by one pixel.

`WinDrawGrayLine` accepts the same parameters as `WinDrawLine` and draws a line that appears "gray." This gray appearance is achieved by alternating each pixel along the length of the line, producing a dotted line.

`WinEraseLine` will "erase" lines by drawing a line using the screen background "color" as the drawing color.

`WinFillLine` is the most flexible, but correspondingly hardest to use, line-drawing function. `WinFillLine` takes the same parameters as the other line functions but uses the current fill pattern to render the line. You will encounter patterns again in the discussion of rectangles, so it's worth examining what it means to set the current pattern.

Fill Patterns

You can set the current pattern on the Palm by calling the function `WinSetPattern`. `WinSetPattern` takes one parameter, a `CustomPatternType`. You might be disappointed (as I was) when you realize that a custom pattern type is simply an array of four words. Because a word is two bytes long, you wind up with a structure that represents eight bytes, or 64 bits. Interpreted visually as a set of eight bytes stacked vertically, you get a binary pattern eight bits (pixels) wide by eight bits (pixels) high.

Because you can define any pattern you want in this 8×8 matrix, this is obviously a flexible drawing mechanism. Unfortunately, that flexibility comes with a cost: You might spend an inordinate amount of time doing binary math with a pencil and paper to come up with the four words that represent your pattern. Other graphics systems provide common predefined values that map to either well-known colors or patterns. One could argue it would have been nice for the folks at Palm Source to define such qualities as gray, light gray, and dot-dash-dot, but they didn't, so you have to.

Once you've determined the magic four-word sequence that will produce the desired pattern, you set it by calling `WinSetPattern`. To be a good citizen, you should first call `WinGetPattern` to obtain the current pattern so that you can restore it after you are done drawing.

Rectangles

You draw rectangles by calling one of the following functions:

```
WinDrawRectangle
WinEraseRectangle
WinFillRectangle
```

These functions are parallel to those that support line drawing. Instead of a set of x,y coordinates, the rectangle functions take a pointer to a `RectangleType` structure. Rectangles contain a "top-left" member, which defines the upper-left x,y coordinate, and an "extent" member, which defines how many pixels the rectangle extends in the vertical and horizontal directions. Be careful not to confuse the extent member with an absolute coordinate that defines the bottom-right corner of the rectangle.

Rectangles are always drawn filled with no border. Thus, `WinDrawRectangle` draws a solid black-filled rectangle. `WinFillRectangle`, as is the case with `WinFillLine`, fills the rectangle with the current pattern as set by `WinSetPattern`.

If you want to draw a rectangle without filling the rectangle's interior, use the `WinDrawRectangleFrame` function.

Drawing Text

Palm OS has basic character-drawing support in the Windowing functional area. This support is supplemented by functions in the Font function family.

To render text on the display, you use `WinDrawChars`, which takes a character string, a string length, and an x,y coordinate as parameters. The x,y coordinate represents the upper-left corner of the first character to be drawn.

There is no out-of-the-box support for advanced features such as text justification or word wrapping, although again, you can use the existing Palm functions to build this capability. The Font function family will give you what you need to develop this kind of support by providing such information as the width in pixels of text represented in a given font.

Bitmaps

You draw bitmaps using the function `WinDrawBitmap`. `WinDrawBitmap` expects a pointer to a bitmap and an x,y coordinate representing the top-left corner of the display area where the bitmap should be drawn.

Bitmaps are defined as resources, so you create them using Constructor (or a compatible third-party resource utility). Given a bitmap resource, you must first load the bitmap using `DmGetResource (`*`bitmapRsc`*`, `*`resourceID`*`)`, where *`bitmapRsc`* tells Palm that you want a resource of type bitmap, and *`resourceID`* is the ID assigned to your bitmap by Constructor. If successful, `DmGetResource` returns a valid bitmap handle, which can be turned into a pointer via `MemHandleLock`. (More on the memory and database managers in Chapter 16, “Palm OS Memory Management.”)

The Palm OS Window Model

You will not find sophisticated windowing support in the Palm SDK. Rather, you will find a set of windowing primitives that, with a little bit of work and understanding, can provide the foundation for whatever windowing you want to support in your application.

What Is a Window?

Although I haven’t made a point of it, I’ve already covered some forms of windowing: Forms are actually based on the Palm OS windowing API! Simply put, a window in the Palm OS is a non-resizable, non-moveable object that has a rectangular dimension and is positioned either directly on the screen or off-screen. A form can be considered a window that has a lot of extra user interaction associated with it, but you can use the windowing APIs to interact with the display area of a form.

Assuming you have a user interface in your program, there is always at least one window associated with the display. Usually, this window is both the "active" window and the "draw" window.

The draw window is the window to which all graphics-drawing functions render their output. The coordinate system of the draw window is *window-relative,* meaning that if two windows divide the screen in half vertically, each window has a coordinate space that begins at 0,0.

An active window has been set to respond to user actions such as pen strokes; it receives all user input. No other window can receive user input while another window is currently active. By default, setting the active window also sets the draw window.

Why would you create one or more windows in your application? If you do not need to draw graphics directly on the screen, you really do not need to bother with windows at all. And because Palm only allows the users to work with one application and one "window" at a time, most applications will have no need to create windows. However, for some applications, windows can simplify your drawing logic by allowing you to use a window-specific coordinate system without worrying about drawing outside the bounds of the window.

Creating a Window

You create a window by calling the function `WinCreateWindow`. You place windows on the display by setting the Bounds parameter. The Frame parameter allows you to set whether your window will have a visible frame and, if so, how thick it will be. The Modal parameter indicates that the window will be the top-level window on the display, and will not go away until explicitly dismissed by the users. If you were building up your own modal dialog-like window, you might set the Modal parameter to `TRUE`. The Focusable parameter indicates that the window is capable of being the active window and thus can receive user input.

In contrast with most other Palm functions, the `WinCreateWindow` function contains an `Out` parameter that receives an error value, if any. (Most Palm functions pass back errors via the return code.)

One "gotcha" in creating a window is that the window will not automatically draw its own frame: You must manually call `WinDrawWindowFrame` after creating your window if you want to immediately see a border on your window.

Also, note that the coordinate system associated with a window does not automatically omit the width of the window frame. You must offset your drawing within the window by the width of the frame, or else you will draw on top of your frame.

Speaking of coordinates, you can obtain the width and height of the current draw window by calling `WinGetWindowExtent`. It is highly recommended that if you are going to do any graphics drawing on the display, you do not rely on the current 160×160 scheme, but rather, you retrieve the extent of the screen display via `WinGetDisplayExtent`. This feature allows you to write more display-independent code that will work on Palm devices with larger screens.

Multiple Windows

You can create multiple windows, each having separate (or overlapping) locations on the screen. You can also use `WinCreateOffscreenWindow` to create an off-screen window for rendering graphics off-screen and quickly swapping display bits in and out (which is useful for such things as animation).

In a multiple-window display, you must manually set each window to be the draw window before drawing on it by calling `WinSetDrawWindow`.

PalmDraw: A Graphics-Drawing Demonstration Program

Now that I've covered the majority of the graphics and windowing functionality in Palm OS, I'll put some of that knowledge to work in a demonstration program. PalmDraw demonstrates the use of graphics and multiple windows by dividing the Palm display into four quadrants, each containing a window. Each of the four windows demonstrates one of the areas of graphics drawing that I've covered in this chapter: lines, rectangles, text, and bitmaps.

Figure 13.1 illustrates PalmDraw's four-quadrant display.

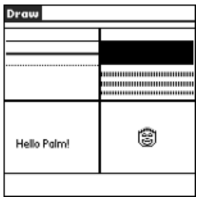


FIGURE 13.1 Multiple windows on the Palm display.

The code for implementing the main form's windowing and drawing functionality appears in Listing 13.1.

LISTING 13.1 The Main Form Code for PalmDraw

```
/*
   DRAW_MAI.CPP
   Main form handling functions.
   Copyright (c) Bachmann Software and Services, 1999-2002
   Author: Glenn Bachmann
*/

// system headers
#include <PalmOS.h>
#include <SysEvtMgr.h>

// application-specific headers
#include "draw.h"
#include "draw_res.h"

static void    MainFormInit (FormPtr formP);
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);

static void  DrawLines ( void );
static void  DrawRectangles ( void );
static void  DrawBitmap ( void );
static void  DrawText ( void );

/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;

   switch (eventP->eType)
   {
      case menuEvent:
      {
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormMenuHandler (formP, eventP);
```

LISTING 13.1 Continued

```
         break;
      }

      case ctlSelectEvent:
      {
         // A control button was tapped and released
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormButtonHandler (formP, eventP);
         break;
      }

      case frmOpenEvent:
      {
         // main form is opening

         FormPtr formP = FrmGetActiveForm ();

         MainFormInit (formP);

         FrmDrawForm (formP);

         handled = true;
         break;
      }

      default:
      {
         break;
      }
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   formP - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr formP)
{
```

LISTING 13.1 Continued

```
// set the controls to an initial state
// create the windows: two on top, one below
RectangleType rect;
Err err;
WinHandle whUpperLeft, whUpperRight, whBottomLeft, whBottomRight, whOrig;
Int16 iWidth = 160;
Int16 iHeight = 160;

WinGetDisplayExtent ( &iWidth, &iHeight );

// Upper left
rect.topLeft.x = 1;
rect.extent.x = ( iWidth / 2 ) - 1;
rect.topLeft.y = 20;
rect.extent.y = 60;
whUpperLeft = WinCreateWindow ( &rect, simpleFrame, false,
                                false, &err );

whOrig = WinSetDrawWindow ( whUpperLeft );
WinDrawWindowFrame ( );
WinSetDrawWindow ( whOrig );

// Upper right
rect.topLeft.x = iWidth / 2;
rect.extent.x = ( iWidth / 2 ) - 1;
rect.topLeft.y = 20;
rect.extent.y = 60;
whUpperRight = WinCreateWindow ( &rect, simpleFrame, false,
                                 false, &err );
WinSetDrawWindow ( whUpperRight );
WinDrawWindowFrame ( );
WinSetDrawWindow ( whOrig );

// bottom left
rect.topLeft.x = 1;
rect.extent.x = ( iWidth / 2 ) - 1;
rect.topLeft.y = 80;
rect.extent.y = 60;
whBottomLeft = WinCreateWindow ( &rect, simpleFrame, false,
                                 false, &err );
WinSetDrawWindow ( whBottomLeft );
WinDrawWindowFrame ( );
```

LISTING 13.1 Continued

```
   WinSetDrawWindow ( whOrig );

   // bottom right
   rect.topLeft.x = iWidth / 2;
   rect.extent.x = ( iWidth / 2 ) - 1;
   rect.topLeft.y = 80;
   rect.extent.y = 60;
   whBottomRight = WinCreateWindow ( &rect, simpleFrame, false,
                                     false, &err );
   WinSetDrawWindow ( whBottomRight );
   WinDrawWindowFrame ( );
   WinSetDrawWindow ( whOrig );

   WinSetDrawWindow ( whUpperLeft );
   DrawLines ();

   WinSetDrawWindow ( whUpperRight );
   DrawRectangles ();

   WinSetDrawWindow ( whBottomLeft );
   DrawText ();

   WinSetDrawWindow ( whBottomRight );
   DrawBitmap ();

   WinSetDrawWindow ( whOrig );
}

/*
   DrawLines:
   Draws example lines
   Parms: none
   Returns: none
*/
void
DrawLines ( void )
{
   Int16 wX, wY;

   // get the bounds of the current draw window
   WinGetWindowExtent ( &wX, &wY );
```

LISTING 13.1 Continued

```
   // single solid line
   // draw from one point to another
   WinDrawLine ( 1, 10, wX -2, 10 );

   // double solid line
   WinDrawLine ( 1, 20, wX -2, 20 );
   WinDrawLine ( 1, 21, wX -2, 21 );

   // single gray line
   WinDrawGrayLine ( 1, 30, wX -2, 30 );

   // patterned lines
   CustomPatternType patternOld;
   CustomPatternType patternDotted = { 0xAAAA, 0xAAAA, 0xAAAA, 0xAAAA };
   CustomPatternType patternDashed = { 0xCCCC, 0xCCCC, 0xCCCC, 0xCCCC };

   WinGetPattern ( &patternOld );

   // single dotted line
   WinSetPattern ( &patternDotted );
   WinFillLine ( 1, 40, wX -2, 40 );
   WinSetPattern ( &patternOld );

   // single dashed line
   WinSetPattern ( &patternDashed );
   WinFillLine ( 1, 50, wX -2, 50 );
   WinSetPattern ( &patternOld );

   return;
}

/*
   DrawRectangles:
   Draws example rectangles
   Parms: none
   Returns: none
*/
void
DrawRectangles ( void )
{
   Int16 wX, wY;
   RectangleType rect;
```

LISTING 13.1 Continued

```
   // get the bounds of the current draw window
   WinGetWindowExtent ( &wX, &wY );

   // Black Rectangle
   rect.topLeft.x = 1;
   rect.extent.x = wX - 2;

   rect.topLeft.y = 10;
   rect.extent.y = 20;
   WinDrawRectangle ( &rect, 0 );

   // Filled pattern rectangle
   CustomPatternType patternOld;
   CustomPatternType patternDotted = { 0xAA55, 0xAA55, 0xAA55, 0xAA55 };

   rect.topLeft.y = 35;
   rect.extent.y = 20;
   WinGetPattern ( &patternOld );
   WinSetPattern ( &patternDotted );
   WinFillRectangle ( &rect, 0 );
   WinSetPattern ( &patternOld );

   return;
}

/*
   DrawBitmap:
   Draws example bitmaps
   Parms: none
   Returns: none
*/
void
DrawBitmap ( void )
{
   MemHandle hBitmap;

   hBitmap = DmGetResource ( bitmapRsc, HappyBitmap );

   if ( hBitmap )
   {
      BitmapPtr pBitmap;
      Int16 wX, wY;
```

LISTING 13.1 Continued

```
      // get the bounds of the current draw window
      WinGetWindowExtent ( &wX, &wY );

      pBitmap = (BitmapPtr)MemHandleLock ( hBitmap );
      WinDrawBitmap ( pBitmap, wX / 2 - 8, 22 );
      MemHandleUnlock ( hBitmap );

      DmReleaseResource ( hBitmap );
   }

   return;
}

/*
   DrawText:
   Draws example text
   Parms: none
   Returns: none
*/
void
DrawText ( void )
{
   WinDrawChars ( "Hello Palm!", StrLen ( "Hello Palm!" ), 10, 30 );

   return;
}

/*
   MainFormMenuHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormMenuHandler (FormPtr /*formP*/, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.menu.itemID)
   {
```

LISTING 13.1 Continued

```
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormButtonHandler (FormPtr formP, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}
```

The code for PalmDraw is fairly simple. Let's step through the most interesting areas of the code to see how things are done.

Setting Up for Multiple Windows

The first thing you do is call `WinGetDisplayExtent` in the main form handler's `Init` handler to obtain the width and height of the display for safety:

```
Int16 iWidth = 160;
Int16 iHeight = 160;

WinGetDisplayExtent ( &iWidth, &iHeight );
```

You then create four windows, representing four quadrants within the display's region, by setting up a structure for the bounding rectangle and calling `WinCreateWindow`:

```
// Upper left
RectangleType rect;
WinHandle whUpperLeft, whUpperRight, whBottomLeft, whBottomRight,
➥ whOrig;

   rect.topLeft.x = 1;
   rect.extent.x = ( iWidth / 2 ) - 1;
   rect.topLeft.y = 20;
   rect.extent.y = 60;
```

Finally, you have to force Palm to draw the windows' frames by setting the current draw window, drawing the frame, and then restoring the draw window to what it was:

```
   whOrig = WinSetDrawWindow ( whUpperLeft );
   WinDrawWindowFrame ( );
   WinSetDrawWindow ( whOrig );
```

The Line-Drawing and Rectangle-Drawing Windows

Once the windows are set up, you have to set each window in turn to be the current draw window, and then render the appropriate graphics on the window. The following segment is from the `DrawLines` function for handling the upper-left window:

```
   Int16 wX, wY;

   // Get the bounds of the current draw window
   WinGetWindowExtent ( &wX, &wY );

   // Single solid line
   // Draw from one point to another
   WinDrawLine ( 1, 10, wX -2, 10 );
```

```
// Double solid line
WinDrawLine ( 1, 20, wX -2, 20 );
WinDrawLine ( 1, 21, wX -2, 21 );

// Single gray line
WinDrawGrayLine ( 1, 30, wX -2, 30 );

// Patterned lines
CustomPatternType patternOld;
CustomPatternType patternDotted = { 0xAAAA, 0xAAAA, 0xAAAA, 0xAAAA };
CustomPatternType patternDashed = { 0xCCCC, 0xCCCC, 0xCCCC, 0xCCCC };

WinGetPattern ( &patternOld );

// Single dotted line
WinSetPattern ( &patternDotted );
WinFillLine ( 1, 40, wX -2, 40 );
WinSetPattern ( &patternOld );

// Single dashed line
WinSetPattern ( &patternDashed );
WinFillLine ( 1, 50, wX -2, 50 );
WinSetPattern ( &patternOld );
```

You can see that I obtain the window's width and height and adjust the draw calls as appropriate to account for the single pixel frame width. The patterned line drawing illustrates the use of `SetFillPattern`. (The dashed line pattern values came from using good old pencil and paper and a binary layout of a two-byte word structure.)

The logic for the rectangle-drawing window is similar to the line-drawing logic, so I won't bother to review it as well.

The Text-Drawing Window

The text window is handled by a single `WinDrawChars` call:

```
WinDrawChars ( "Hello Palm!", StrLen ( "Hello Palm!" ), 10, 30 );
```

I wasn't looking to get fancy for this demonstration application, but you could certainly spend some time with the `Fnt` functions and attempt to properly center the text within the window.

The Bitmap-Drawing Window

The bitmap-drawing window is a bit more complicated, but only because it requires you to manually load a bitmap resource and deal with memory handle locking and unlocking:

```
   MemHandle hBitmap;

   hBitmap = DmGetResource ( bitmapRsc, HappyBitmap );

   if ( hBitmap )
   {
      BitmapPtr pBitmap;
      Int16 wX, wY;

      // Get the bounds of the current draw window
      WinGetWindowExtent ( &wX, &wY );

      pBitmap = (BitmapPtr)MemHandleLock ( hBitmap );
      WinDrawBitmap ( pBitmap, wX / 2 - 8, 22 );
      MemHandleUnlock ( hBitmap );

      DmReleaseResource ( hBitmap );
   }
```

If I were to do a lot of bitmap manipulation, I would cut myself a break and whip up some helper functions:

```
Void
PalmDrawBitmap (Uint16 wBitmapID, Int16 x, Int16 y)
{
   MemHandle hBitmap;

   hBitmap = DmGetResource ( bitmapRsc, wBitmapID );

   if ( hBitmap )
   {
      BitmapPtr pBitmap;

      pBitmap = (BitmapPtr)MemHandleLock ( hBitmap );
      WinDrawBitmap ( pBitmap, x, y );
      MemHandleUnlock ( hBitmap );
      DmReleaseResource ( hBitmap );
   }
   return;
}
```

The big advantage of writing and using such a utility function is that you don't have to worry about resource loading or locking and unlocking memory!

Summary

Palm OS provides a basic set of graphics primitives that let you render lines, rectangles, text, and bitmaps on the display. A similar set of windowing functions are useful for creating drawing regions on, or off screen but are somewhat awkward to use.

Nevertheless, it is my expectation that as time goes on, the graphics and windowing model for at least some future Palm devices will grow and gain support for more advanced constructs. In the meantime, if you need to add drawings to your applications, Palm provides enough raw tools for you to build your own utility functions.

14 Handling Pen Events

IN THIS CHAPTER

This chapter discusses how to directly handle pen events as they occur on the Palm display. Most applications never need to look at pen events, but developers of drawing programs, graphics applications, or games that require direct, non-character input from the users probably will want to be able to take action based on events generated by pen movements.

For many applications, the only type of user input that is required consists of either alphanumeric characters entered into the Graffiti input area, or navigational buttons such as the hard scroll button or silk-screen icons such as menu, calculator, and so on. For a forms-based program with input fields and user-interface controls like the ones explored earlier in the book, character input happens transparently to the programmer. All pen strokes are interpreted by the Palm OS, transcribed into recognizable characters, and funneled into the user-interface control that currently has the input focus.

However, not all applications are forms-driven. A "Scribble" application, for example, allows you to draw directly on the main window, and does not treat the pen movements as part of normal Graffiti input. This type of application must interface with Palm OS at a lower level than Graffiti, and must deal with a type of event called a *pen event*.

This chapter examines the events that are associated with pen strokes and then presents two sample applications that demonstrate what you can do with these events.

What Is a Pen Event?

If you have ever written code that handles mouse events on PC platforms, you may recall that there are three primary actions from which other actions derive: mouse down, mouse up, and mouse move. Similarly, on Palm OS the three pen events are `penDownEvent`, `penMoveEvent`, and `penUpEvent`. These events are sent by the Event Manager to your form when the pen is tapped, moved, or lifted on the palm display.

The `penDownEvent` event is generated by the users touching the pen down in the display area. It passes to you the x, y coordinates of the pen relative to the current window. (Windows and coordinate spaces are discussed in Chapter 13, "Drawing Graphics.")

The `penUpEvent` event is sent when the user lifts the pen from the display. As with `penDownEvent`, you are passed the x, y coordinates of the pen. In addition, you are also passed the x, y coordinates of the beginning of the pen stroke (which should be the same as the coordinates passed in the last `penDownEvent`). Note that the start and end coordinates are in display coordinates and are not window-relative.

The `penMoveEvent` event is sent when the pen is down and it is moved on the display. You are passed the x, y coordinates relative to the current window.

How Pen Events and Control Events Coexist

It might have occurred to you that if you attempt to trap all the pen events that occur on the display, any controls or other user-interface elements on your form will not be able to process the pen events for their own purposes. Well, you are correct. If your form's event handler traps `penDown` events and returns `TRUE`, the `penEvent` will never be translated into a `ctlSelectEvent` (or other user-interface event). It is your responsibility to be aware of the location of the user-interface elements on your form, and to return `FALSE` from your event handler if you detect a pen event in those locations.

The best advice I've seen in this situation is to divide the screen into regions, some of which contain user-interface elements that need to process their own pen events. Track the coordinates of those regions you should process pen events for and others for which you should pass the event back to the event dispatcher. If you carefully check the x, y coordinates passed to you in the event handler, your pen movements should be able to coexist peacefully with other user-interface elements.

Pen Events and Graffiti

Although they are both driven by pen strokes, pen events actually have nothing to do with graffiti. Graffiti occurs when the user performs pen strokes in the special graffiti area on the Palm device. Capturing pen events does not interfere with the Palm's ability to perform graffiti handwriting recognition, nor will graffiti pen strokes generate pen events on your form.

Graffiti activity is handled directly by the Graffiti Manager, which translates pen strokes into key events. These key events are then passed into your application via the normal event loop mechanism.

Palm SDK functions allow you to interact in some ways with the Graffiti Manager and its dictionary handling. For instance, it is possible to trap a `penUpEvent` from the application display area and ask Graffiti Manager to process the stroke (see `GrfProcessStroke`) and "manually" translate it into a key event. This is in fact what the Palm OS SDK Graffiti tutorial does when it needs to recognize your strokes.

Doodle: Creating a Pen-Drawing Program

The amount of paper wasted every day by idle doodlers in meetings around the world is abominable. Let's do something about that, and create an electronic doodle pad that doesn't waste precious trees.

This program is called Doodle, and it does its magic with very little coding. If by now you have guessed that you will implement doodle by taking advantage of pen events, you have guessed right.

Figure 14.1 shows what doodle looks like after a few pen strokes; the code for the main form appears in Listing 14.1.

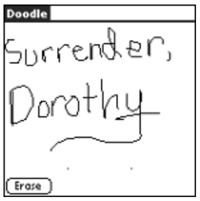


FIGURE 14.1 Doodling away.

LISTING 14.1 Source Code for the Main Form of Doodle

```
/*
   DRAW_MAI.CPP
   Main form handling functions.
   Copyright (c) Bachmann Software and Services, 1999-2002
   Author: Glenn Bachmann
*/
```

LISTING 14.1 Continued

```
// system headers
#include <PalmOS.h>
#include <SysEvtMgr.h>

// application-specific headers
#include "draw.h"
#include "draw_res.h"

static void    MainFormInit (FormPtr formP);
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);

void  DrawGameBoard ( void );

static UInt16 s_x = 0;
static UInt16 s_y = 0;

static Boolean s_bPenDown = false;

/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;

   switch (eventP->eType)
   {
      case menuEvent:
      {
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormMenuHandler (formP, eventP);
         break;
      }

      case penDownEvent:
      {
         if ( eventP->screenY < 130 )
```

LISTING 14.1 Continued

```
        {
           s_x = eventP->screenX;
           s_y = eventP->screenY;

           s_bPenDown = true;
           return true;
        }
        else
        {
           return false;
        }
     }

     case penUpEvent:
     {
        if ( s_bPenDown && eventP->screenY < 130 )
        {
           UInt16 xNew = eventP->screenX;
           UInt16 yNew = eventP->screenY;

           WinDrawLine ( s_x, s_y, xNew, yNew );

           s_x = 0;
           s_y = 0;

           s_bPenDown = false;
        }

        break;
     }

     case penMoveEvent:
     {
        // only do something if the pen is down
        if ( s_bPenDown && eventP->screenY < 130 )
        {
           UInt16 xNew = eventP->screenX;
           UInt16 yNew = eventP->screenY;

           WinDrawLine ( s_x, s_y, xNew, yNew ) ;
```

LISTING 14.1 Continued

```
            s_x = xNew;
            s_y = yNew;
         }
         break;
      }

      case ctlSelectEvent:
      {
         // A control button was tapped and released
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormButtonHandler (formP, eventP);
         break;
      }

      case frmOpenEvent:
      {
         // main form is opening

         FormPtr formP = FrmGetActiveForm ();

         MainFormInit (formP);

         FrmDrawForm (formP);

         handled = true;
         break;
      }

      default:
      {
         break;
      }
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   formP - pointer to main form.
   Return:  none
*/
```

LISTING 14.1 Continued

```
void
MainFormInit (FormPtr formP)
{
   // set the controls to an initial state

   DrawGameBoard ();

}

/*
   DrawGameBoard:
   Redraws the tic tac toe display
   Parms: none
   Returns: none
*/
void
DrawGameBoard ( void )
{
   // our game area is 120 pixels high,  starts after the title area,
   // and ends above the button area on the bottom
   WinDrawLine ( 53, 16, 53, 136 );
   WinDrawLine ( 106, 16, 106, 136 );

   // horizontal lines
   WinDrawLine ( 0, 56, 159, 56 );
   WinDrawLine ( 0, 96, 159, 96 );
   return;
}

/*
   MainFormMenuHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormMenuHandler (FormPtr /*formP*/, EventPtr eventP)
{
   Boolean handled = false;
```

LISTING 14.1 Continued

```
   switch (eventP->data.menu.itemID)
   {
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormButtonHandler (FormPtr formP, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      case MainEraseButton:
      {

         RectangleType rect;
         rect.topLeft.x = 0;
         rect.topLeft.y = 16;
         rect.extent.x = 159;
         rect.extent.y = 136 - rect.topLeft.y;

         WinEraseRectangle ( &rect, 0);
         handled = true;
         break;
      }
      default:
      {
```

LISTING 14.1 Continued

```
         handled = false;
         break;
      }
   }
   return handled;
}
```

Let's take a look at how Doodle does its thing.

To draw on the screen as you move the pen around, you need to know when the pen is down, when it moves, and when it is up, as well as the pen's coordinates in all of these situations. As you've seen, this information is available via the pen events.

The following code processes the `penDown` event:

```
case penDownEvent:
{
    if ( eventP->screenY < 130 )
    {
       s_x = eventP->screenX;
       s_y = eventP->screenY;

       s_bPenDown = true;
       return true;
    }
    else
    {
      return false;
    }
 }
```

Because there's a button at the bottom of the form, you want to avoid processing `penDown` events that occur in that region of the screen. You can give the button a healthy buffer zone of 30 pixels, hence the test to see if the `screenY` value is less than 130.

If you care about the `penDown` event, you set a flag indicating that the pen is down, and you also store the current coordinates in some static variables. No drawing yet—the user has to move the pen first.

Here's the `penMove` handling:

```
case penMoveEvent:
      {
         // Only do something if the pen is down
         if ( s_bPenDown && eventP->screenY < 130 )
         {
            UInt16 xNew = eventP->screenX;
            UInt16 yNew = eventP->screenY;

            WinDrawLine ( s_x, s_y, xNew, yNew );

            s_x = xNew;
            s_y = yNew;
         }
         break;
      }
```

Here, you again determine whether the y coordinate is less than 130 to avoid conflicts with the push button. If you want the event, you get the new coordinates and draw a line from the last saved pen coordinates to the new coordinates using the window function `WinDrawLine`. You finish up by again saving the new coordinates in the static variables.

The `penUp` event is virtually identical to `penMove`, except that you also set our pen down flag to `FALSE`.

NOTE

A quick word about the pen down flag. As it happens, it is possible to get spurious `penUp` events when the user taps the buttons located below the bottom of the display area. To combat this problem, simply refuse to do anything on a `penUp` if the flag is not set.

All that's left to do is to provide the ability for the electronic doodler to erase his or her artwork (a handy feature if the boss happens to walk by the old cubicle!). You do this with a normal button event handler, which defines the rectangular region you want to erase (again leaving clearance for the button) and calling the `WinEraseRectangle` function.

Even More Fun: Tic-Tac-Toe

If doodling isn't enough of a waste of time, not to worry. Now that you understand how to trap pen events, there's no end to the fun you can have. If you add just a little bit more drawing logic to the main form, you create a fine game of Tic-Tac-Toe.

You can hold on to all of the same pen-handling logic you saw in the last application, so I won't bother to repeat it here. All that's new is the following function, which paints the game board:

```
/*
   DrawGameBoard:
   Redraws the tic tac toe display
   Parms: none
   Returns: none
*/
void
DrawGameBoard ( void )
{
   // Our game area is 120 pixels high,  starts after the title area,
   // And ends above the button area on the bottom
   WinDrawLine ( 53, 16, 53, 136 );
   WinDrawLine ( 106, 16, 106, 136 );

   // Horizontal lines
   WinDrawLine ( 0, 56, 159, 56 );
   WinDrawLine ( 0, 96, 159, 96 );
   return;
}
```

To render the initial game board, call this function from the form's initialization routine:

```
/*
   MainFormInit:
   Initialize the main form.
   Parms:   formP - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr formP)
{
   // Set the controls to an initial state
   DrawGameBoard ();
}
```

Finally, when you erase the form in response to the button, you are obligated to redraw the board as well.

Figure 14.2 shows what the game looks like. (I'm X, in case you hadn't guessed.)

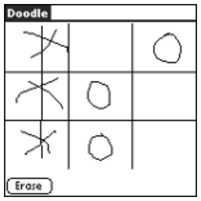


FIGURE 14.2 Tic-Tac-Toe: A winner!

Summary

For those developers who want to allow users to draw freehand on the display, or for whatever reason require interaction with the users that the standard user-interface controls do not provide, pen events are worth examining.

Some important gotchas come with the territory, such as being aware of the location of your form's controls. But that's typical for software development: The lower you delve into what goes on behind the scenes, the more power you have to control and react to things. Along with that power, though, comes responsibility and the need to be a well-behaved citizen in the use of system resources such as the display.

15 Adding Color to Palm Applications

IN THIS CHAPTER

- Color and Bits
- Color and User Interface Elements
- RGB Values, the Palette, and the Color Table
- Setting the Color of Your User Interface Elements
- Using Color with Drawing Routines
- Letting Users Pick a Color
- Putting Colors to Work: The `ColorTest` Sample

There is something about color that makes people stand up and take notice. Part of this is because colors can convey meanings, emotions, style, and attitude. If you take an object that is black, white, or gray, and then add colors or patterns, the character of the object will instantly change.

Color can also be used to convey information that is important in and of itself. To take an every-day example, a red traffic light has a very different meaning than a green traffic light. Graphs and charts typically employ color to display otherwise dry numeric information in a more visual manner, making the same information more understandable at a glance. Color is wonderful for adding emphasis to selected data.

Color can also affect people's moods and feelings. A tranquil blue sky can evoke peace and comfort. In contrast the use of multiple high-intensity colors that clash with each other can create discomfort and stress. People feel comfortable with color in their surroundings, but color can also be misused and overused. Most of us—at one time or another—have seen poorly designed presentation slides that made use of every color in the rainbow, resulting in sensory overload and confusion. Further, it is important to note that a significant percentage of the human population is afflicted with some level of color-blindness, which makes it important to avoid relying exclusively on differences in color to describe information.

From movies to televisions to computer displays, color seems to be an inevitable goal, eventually becoming the standard rather than the exception. It is hard to find a household with a black-and-white television in the United States, and aside from a few films that use black and white for effect, virtually every movie made is in color.

Computer displays seem to follow the same progression; virtually all PC displays sold today are color displays. In the world of handheld devices, the same trend appears to be occurring. The technology for producing bright color screens is available; it has simply been a matter of bringing the cost down to the point where the advanced screens are affordable.

In today's market, color screens largely remain in the domain of higher end devices, but as earlier color devices are replaced by new models, as components get cheaper, and as competition grows among device manufacturers, prices have dropped dramatically. The once top-of-the-line Handspring Visor Prism, with its strikingly beautiful color screen, now sells for less than half of its original price, a bargain for such a capable PDA. It appears to be inevitable that at some point in the not-too-distant future virtually all handhelds will sport a color display.

Beginning in OS 3.5, Palm started adding operating system support for color displays. Palm also exposed a limited set of developer APIs, allowing Palm programmers some control over the use of color in their applications when running on a color device.

This chapter explains the current support for color in the Palm OS in terms of the developer APIs available today. As part of the chapter, I present a simple example application that demonstrates how to choose colors both at the user and developer levels, and how to set the visible color used when drawing.

Color and Bits

As mentioned earlier, Palm OS 3.5 introduced the first color support, using 8 bits for 256 colors. Palm OS 4.0 improved on that by adding 16-bit color support.

I should note that unless you are doing imaging work or intensive color translation and scaling, you need not be terribly concerned with the bit depth or number of colors. Imaging and bit-blasting are a whole separate topic: This chapter focuses on the use of color within the standard user Palm user interface elements.

Color and User Interface Elements

Color control is employed in Palm programming by changing the color used for selected screen elements in the current drawing window (for a discussion of drawing windows, refer to Chapter 13, "Drawing Graphics"). Full control over every user interface element is not exposed at present, but there are enough options to give you some good flexibility in the presentation of your user interface.

The user elements covered in the following sections are capable of being associated with a programmer-selectable color.

General User Interface Elements

- `UIObjectFrame`: The border color of buttons, triggers, check boxes, and other user interface elements
- `UIObjectFill`: The background color of a user interface element
- `UIObjectForeground`: The foreground color of a user interface element
- `UIObjectSelectedFill`: The background color of the currently selected user interface object
- `UIObjectSelectedForeground`: The foreground color of the currently selected user interface object

Menus

- `UIMenuFrame`: The border color of a menu
- `UIMenuFill`: The background color of a menu item
- `UIMenuForeground`: The foreground color of a menu's text
- `UIMenuSelectedFill`: The background color of the selected menu item
- `UIMenuSelectedForeground`: The foreground color of the text of the selected menu item

Input Fields

- `UIFieldBackground`: The background color of an editable text field
- `UIFieldText`: The foreground color of the text in the editable field
- `UIFieldTextLines`: The color of underlines in an editable field
- `UIFieldCaret`: The color of the cursor in an editable text field
- `UIFieldTextHighlightBackground`: The background color for selected text in an editable text field
- `UIFieldTextHighlightForeground`: The color of the selected text in an editable text field

Forms and Dialogs

- `UIFormFrame`: The color of the border and title bar on a form
- `UIFormFill`: The background color of a form
- `UIDialogFrame`: The color of a border and title bar on a modal form
- `UIDialogFill`: The background color of a modal form

Alerts

- `UIAlertFrame`: The color of the border and title bar on an alert
- `UIAlertFill`: The background color of an alert panel

Note that tables are colored by using the `Field` functions.

RGB Values, the Palette, and the Color Table

Palm OS maintains a palette of available colors in the system "color table." Each entry in this table represents a single color available to the OS, and is stored as an RGB value. (RGB is a standard way of representing a color by assigning a value between 0 and 255 to each of three colors—red, green, and blue.)

Naturally, the system color table does not have an entry for every possible RGB color combination; to do so would be wasteful and inefficient. Rather, the table contains RGB values that are commonly used in computer displays, including some standard entries dictated by HTML.

Normally if you ask the system to draw using a specific RGB value, the system will take your requested color and match to the nearest color in the color table. If instead you simply must have that specific shade of chartreuse, for example, you can actually change the entries in the color table using the `WinPalette` call.

You can also ask for a color that is presently in the color table by specifying the index into the table that contains the color you desire. Thus for a given palette, an RGB value and a color index can be used to represent the same thing. Palm OS supplies conversion functions that allow you to retrieve the RGB value given an index, as well as the index given an RGB value.

Setting the Color of Your User Interface Elements

An earlier section described the user interface elements that can be set with a color of your own choice. Now that you understand RGB values and indexes into the color table, you can take a look at how to set a specific user interface element's color scheme to something other than the default.

Palm supplies a function called `UIColorSetTableEntry` that will allow you to specify a type of user interface element along with an RGB value that should be used to color items of that type.

For example, to set the background color of the forms used in your application to bright red, you would write:

```
RGBColorType rgb;
rgb.r =255;
rgb.g = 0;
rgb.b = 0;

UIColorSetTableEntry ( UIFormFill, &rgb );
```

You simply force a form redraw using `FrmDrawForm`, and you'll have your users seeing red in no time! Note that in this example if you wanted your entire user interface to be red, you would also need to make the same function call for each type of user interface element that appeared on your form.

One disappointing note: the `UIColorSetTableEntry` calls you make are good only while your application is running. When your application exits, Palm OS resets the colors associated with each element back to their defaults. Thus at present it is not possible (at least using the normal Palm OS APIs) to create "color themes" that are applied to every application on your Palm. Maybe this capability will be exposed in a future release of the OS.

Using Color with Drawing Routines

The previous sections dealt with how to attach color values to user interface elements. If you render any of your user interface manually using the window-drawing commands such as `WinDrawRectangle` or `WinDrawChars`, you might want to control how they appear as well.

What you need to do is set the current window's foreground and background colors whenever you are about to perform a drawing command. The proper way to do this is as follows:

```
// get the index of the color you wish
newIndex = WinRGBToIndex ( &rgb );

//set the foreground color
oldIndex = WinSetForeColor ( newIndex );
```

```
//set the bounds of the rectangle to draw
RctSetRectangle ( &rect, 80, 31, 80, 14);

// draw the rectangle
WinDrawRectangle (&rect, 0);

// reset the foreground color to what it was before
WinSetForeColor (oldIndex );
```

There are a couple of things to note in this piece of code. First, the `WinSetForeColor` function does not take an RGB value, but rather it takes an index into the color table. This is where the RGB-to-index conversion function `WinRGBToIndex` comes in handy.

Second, just like you should clean up after yourself when you are a guest in someone's house, it is safest to make sure you leave the current window's color settings the way you found them. In the previous example, you only wanted to use a specific color for drawing a single rectangle. The current window may get confused if you do not set the drawing color back to the way it was, with undesirable results. Luckily, the `WinSetForeColor` function always gives you the previous setting, so its easy enough to finish up by simply resetting that value.

Letting Users Pick a Color

If you want your users to be able to select a color in your application, Palm OS does provide a function that lets you do it. `UIPickColor` is a simple-to-use function that displays a dialog box containing either a grid of small, selectable colored boxes, one for each palette color, or a set of three sliders (one each for red, green, and blue), allowing you to set the level of each color as an RGB value. The users can switch the viewing mode by simply selecting the drop-down selector at the bottom right.

Figure 15.1 shows `UIPickColor` in action.

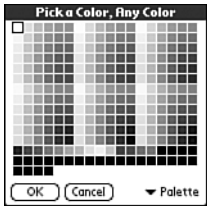


FIGURE 15.1 Pick a color, any color!

To use `UIPickColor`, you need to make a call like the following:

```
Boolean b;
b = UIPickColor (NULL, &rgb, UIPickColorStartRGB,
     "Pick a Color, Any Color", NULL);
```

You can initialize the `UIPickColor` dialog to either viewing mode, as well as to any initial color selection by either RGB value or index. In the previous example, I chose to pass in an RGB structure, thus I simply left the first parameter (an index) as `NULL`.

Putting Colors to Work: The `ColorTest` Sample

`ColorTest` is a simple application I put together that illustrates the concepts discussed in this chapter. It has a very simple main form, consisting of two buttons, a `UIPickColor` button and a `FormFill` button.

Tapping on the `UIPickColor` button will invoke (you guessed it) the `UIPickColor` function, displaying the Palm color-picker. To prove that you successfully received the selected color, upon returning from tapping OK, you'll use `WinDrawRectangle` to draw a colored rectangle as a band across the top part of the form.

Tapping on `FormFill` also invokes the `UIPickColor` function, but upon return it sets the background color of the application's form to the chosen color, using the `UIColorSetTableEntry` function.

The code for `ColorTest`'s main form handler appears in Listing 15.1.

LISTING 15.1 Source Code for the Main Form of `ColorTest`

```
/*
   PLM_MAIN.CPP
   Main form handling functions.
   Copyright (c) Bachmann Software and Services, 1999-2002
   Author: Glenn Bachmann
*/

// system headers
#include <PalmOS.h>
#include <SysEvtMgr.h>

// application-specific headers
#include "plm_main.h"
#include "plm_res.h"

static void    MainFormInit (FormPtr formP);
```

LISTING 15.1 Continued

```
static void    MainFormClose( FormPtr pForm );
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);


/*
   MainFormEventHandler (EventPtr)
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr pEvent)
{
   Boolean  handled = false;
   FormPtr pForm = FrmGetActiveForm ();


   switch (pEvent->eType)
   {
      case menuEvent:
      {
         handled = MainFormMenuHandler (pForm, pEvent);
         break;
      }
      case ctlSelectEvent:
      {
         handled = MainFormButtonHandler ( pForm, pEvent );
         break;
      }
      case frmCloseEvent:
      {
         MainFormClose ( pForm );
         break;
      }
      case frmOpenEvent:
      {
         // main form is opening
         MainFormInit (pForm);
         FrmDrawForm ( pForm );
         handled = true;
         break;
```

LISTING 15.1 Continued

```
      }
      default:
      {
         break;
      }
   }

   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   pForm - pointer to main form.
   Return:  none
*/
void
MainFormInit ( FormPtr pForm )
{

}


void
MainFormClose ( FormPtr pForm )
{
}


/*
   MainFormMenuHandler:
   Handle a command sent to the main form.
   Parms:   pForm    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormMenuHandler (FormPtr /*pForm*/, EventPtr eventP)
{
   Boolean handled = false;
```

LISTING 15.1 Continued

```
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   pForm    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/

Boolean
MainFormButtonHandler (FormPtr pForm, EventPtr eventP)
{
   Boolean handled = false;
   SystemPreferencesType prefType;

   switch (eventP->data.ctlEnter.controlID)
   {
      case MainPickColorButton:
      {
         Boolean b;
         IndexedColorType index;
         RGBColorType rgb;

         Err err;
        UInt32 uiRomVersion;
        UInt32 uiVersion35 = sysMakeROMVersion(3,5,0,sysROMStageRelease,0);

        err = FtrGet(sysFtrCreator,
                 sysFtrNumROMVersion, &uiRomVersion);

         index = 0;
         rgb.r = 0;
         rgb.g = 0;
         rgb.b = 0;

         b = UIPickColor (&index,
                      &rgb,
                    UIPickColorStartPalette,
                    "Pick a Color, Any Color",
```

LISTING 15.1 Continued

```
                    NULL);
         if (b)
         {
            char szRGB[40];
            IndexedColorType oldIndex, newIndex;
            RectangleType rect;

            StrPrintF ( szRGB, "%d red %d green %d blue",
                        rgb.r, rgb.g, rgb.b );

            FrmCustomAlert (ErrorAlert, szRGB, NULL, NULL );

            newIndex = WinRGBToIndex ( &rgb );
            oldIndex = WinSetForeColor ( newIndex );
              RctSetRectangle ( &rect, 0, 17, 159, 14);
            WinDrawRectangle (&rect, 0);
            WinSetForeColor (oldIndex );

         }
         break;
      }

      case MainFormFillButton:
      {
         Boolean b;
         RGBColorType rgb;
         Err err;

         IndexedColorType oldIndex, newIndex;
         RectangleType rect;

         UIColorGetTableEntryRGB ( UIFormFill, &rgb );

         b = UIPickColor (NULL,
                      &rgb,
                    UIPickColorStartRGB,
                    "Pick a Color, Any Color",
                    NULL);
         if (b)
         {
            newIndex = WinRGBToIndex ( &rgb );
            oldIndex = WinSetForeColor ( newIndex );
```

LISTING 15.1 Continued

```
              RctSetRectangle ( &rect, 80, 31, 80, 14);
            WinDrawRectangle (&rect, 0);
            WinSetForeColor (oldIndex );

            UIColorSetTableEntry ( UIFormFill, &rgb );
            FrmDrawForm (pForm);
         }
         break;
      }
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}
```

Summary

For users of your programs on color-enabled devices, the Palm OS color APIs are rich enough to allow you to make very effective use of color in your application. Of course with that capability comes the need to tastefully apply color where it makes the most impact, which is more of a design than a programming issue.

Using the `UIPickColor` function, you can even abdicate the responsibility for having good taste by letting your program's users decide!

PART III

Memory, Databases, and Files

IN THIS PART

16 Palm OS Memory Management

IN THIS CHAPTER

- Overview of the Palm OS Memory Manager
- Allocating and Using Memory
- Memory Allocation Functions
- Memory Manipulation

It seems as though every programming book that describes how to develop applications for a given operating system contains the obligatory chapter on memory management. These books include pages of discussion revolving around concepts such as heaps, free stores, and stack pointers, as well as one or two impressive-looking diagrams that map out how the operating system divides memory.

I usually plow through such chapters, as if I should feel guilty if I don't commit the internals of the operating system to memory. Inevitably, I read this material and then forget it over time. On those rare occasions when I have a programming task in front of me that requires that I be intimate with the actual memory architecture, I generally consult the operating system vendor's documentation anyway.

As I explained in my coverage of the Palm OS database storage model, this book is intended to provide practical knowledge and examples of how to intelligently use the Palm SDK to create applications. If you are looking for a detailed under-the-hood treatment of the Palm OS memory management model, this chapter is not it. If however you are looking for a practical explanation of the major issues in the use of Palm memory, and examples of how to properly allocate, use, and free memory in your program, you are in the right place!

This chapter therefore will (very briefly) give you a sense of the memory architecture of the Palm OS. You will actually use this knowledge primarily as background for the memory allocation limitations your application will face as it runs on the Palm. After the architectural overview, I will get into the practical matters that all programmers face every day about how to acquire, use, and release memory.

Overview of the Palm OS Memory Manager

The memory for a Palm device lives on something called an internal memory card. Although the Palm documentation states that the Palm can theoretically support up to 256 "cards," the concept of cards that fit in this model simply never happened. The one exception to this is the Handspring Visor line of devices, which supports a proprietary expansion slot called *Springboard*, which could be mapped to "card 1." Aside from Springboard modules, no shipping Palm OS devices contain more than one card (card 0).

Each card contains either one or two groupings of memory, called *stores*. Today's single card contains two stores, one for ROM (read-only memory) and one for RAM (random-access memory). The ROM store contains the Palm OS itself, the default built-in applications (such as Address Book), and the default databases associated with the built-in applications.

RAM is where most of the action occurs for programmers and users. It contains applications other than the built-in ones, system preferences, dynamic memory required by running applications, and user data (stored in databases).

Memory Cards versus Expansion Cards

An important distinction must be made here. The term "card" is loosely used in two ways on Palm OS devices. In this chapter, a memory card is discussed in terms of a storage area mapped into the Palm memory model.

However, as you will see in Chapter 20, "Expansion Memory Cards and the Virtual File System," the Palm platform also supports "expansion cards" such as SD, MMC, and Compact Flash. It is very important not to confuse the two types of cards; they are treated very differently in Palm OS programming.

Data Storage, Heap Size, and Dealing with Memory Constraints

The good news is that the Palm has a theoretical physical address space of 4GB. Exciting, right? Can't wait to use all that memory? Forget it. First, recall that the vast majority of Palm devices manufactured to date have anywhere from 2MB to 16MB of memory on-board. You say, "Okay, it's not 4GB, but 16MB still sounds like plenty of memory for my application." It's time for an explanation of dynamic RAM and storage RAM.

The architects of the Palm OS made a design decision to favor small, fast applications and large storage space. That design resulted in the division of the RAM store into two areas: dynamic RAM and storage RAM. Dynamic RAM contains the memory heap available to applications as working space that's allocated dynamically. Any memory in the RAM store that is not dedicated to dynamic RAM is designated for storage RAM. Storage RAM is primarily devoted to databases.

Now you might be starting to worry (and rightly so). "Just how much of my precious 2MB is available to my application for its dynamic memory needs?" you might wonder. The answer is not pretty no matter how you slice it, but things are getting better with each release of the Palm OS. The dynamic heap on Palm OS version 2.0 was only 64KB, of which approximately 12KB was likely to be available to your application for dynamic allocations.

Luckily, it is fairly safe to ignore older devices that run Palm OS 2.0. Things are a bit better with version 3.0, but still fairly tight. On Palm OS 3.0 thru version 3.3, the dynamic heap is 96KB in size, leaving approximately 36KB available to your application. Although it is your choice to ignore or support devices running Palm OS 3.3, you will be excluding a sizable number of potential customers who own devices such as the Palm V, Palm VII, and Palm III.

Starting with Palm OS 3.5, the operating system will in fact decide how much dynamic heap to make available to applications based upon the remaining free storage memory on the device's main memory card.

The implication of these heap size limitations on your application's design is significant. In some ways, it requires you to completely rethink how you develop software. Even if you could guarantee that all your users are running OS 3.5 or greater, it is entirely possible that—because of device conditions—you will still have a relatively small amount of dynamically allocatable memory available to your application. We've all been spoiled by Moore's law, with superfast PCs containing hundreds of megabytes of RAM now common. I've certainly written my share of large applications that expect, need, and routinely receive, megabytes of memory.

It helps to think back to earlier days of PC computing, when DOS applications were designed in low memory conditions. Many of the same ground rules apply:

- Memory is limited.
- Allocate only as much memory as you need.
- Free memory as soon as you are done with it.
- Carefully consider the bare minimum information that you need to load into memory from permanent storage.

- Do not attempt to load arrays of user data into dynamic memory. Rather, read and write data "in place."
- Processor- or memory-intensive modules that cannot be eliminated or redesigned should be offloaded to external systems.

Do not despair. With the right mindset and a healthy respect for the Palm device, you can develop highly functional and useful applications that look great and run fast on the Palm. You just need to shed the notion that you can port that multi-megabyte desktop application lock, stock, and barrel onto the Palm.

The Persistence of Memory

When the device is powered off or goes to sleep, the Palm's batteries retain the memory contents. If the batteries are removed, the contents are still retained for approximately one minute.

If the device experiences a "soft" reset, the dynamic heap, including the running application and all its allocations, is rebuilt. During a "hard" reset, the storage heap (including all stored user data) is also reset, resulting in the device reverting to its default configuration.

As with many operating systems, the Palm OS will clean up after your program exits, freeing all allocations still not freed by your application. Naturally, you should not rely on this safety net, and you should diligently work to remove any memory leaks in your application.

Allocating and Using Memory

On the Palm, dynamic memory allocations are called *chunks* and are limited to slightly less than 64KB for each allocation.

Memory allocations result in either a memory handle or a memory pointer being returned to you by the memory manager. Handles and pointers are the source of some confusion, so let's define them.

A *memory handle* is a reference to a moveable block of memory. This handle must be locked in order to yield a pointer that is usable by your code.

A *memory pointer* is a pointer to a non-moveable memory chunk. A pointer can be allocated either directly by the programmer, or it can be returned as the result of locking a moveable memory handle. It is important to remember that a pointer that results from a handle lock is only valid while that handle is locked. If a handle is subsequently unlocked and then relocked, there is no guarantee that the same pointer value will be returned.

When should you use a handle, and when should you use a pointer? Remember that a pointer returned from locking a handle is virtually the same in terms of its usability as a pointer that was directly allocated. But there is one important difference: A handle can be moved by the operating system as it sees fit in response to further memory allocations. Allocating your memory as moveable handles gives the operating system much more flexibility in how it manages memory, and thus, it has a much better chance of being able to allocate more memory for your application.

With this in mind, for short-lived, small allocations, it is fine to directly allocate memory pointers. For longer-lived or larger allocations that might seriously create a sandbar in the dynamic heap, it is advisable to allocate a memory handle and lock it down to a pointer only as needed.

Memory Allocation Functions

The majority of memory manager functions that are applicable to most application needs can be grouped into handle-based functions and pointer-based functions.

One important point to make: Never use these functions to manipulate memory associated with database records, and never use database functions to manipulate memory blocks allocated by the memory manager functions. They address two different kinds of heaps, and you are asking for trouble by mixing the two.

Handle Functions

You use `MemHandleNew` to allocate a block of moveable memory, returning a handle. It is similar in usage to the old runtime library `malloc`, except for the handle part. When you want to actually use the memory, you need to lock it down and obtain a pointer. You do this by calling `MemHandleLock`. Locks should be brief in nature: After you are done using the memory, you should use `MemHandleUnlock` to return the block to a moveable state. Finally, when you no longer need the memory, `MemHandleFree` returns it to the operating system.

You can obtain the size of a memory block using a handle by calling `MemHandleSize`. You can resize memory blocks by calling `MemHandleResize`, with the caveat that you can only be successful in resizing a block if it is unlocked.

Pointer Functions

Pointer functions are essentially parallel to the handle-based functions, with the exception that they don't need to be locked and unlocked. `MemPtrNew` allocates a new, non-moveable block of memory and returns a pointer to the caller, ready for use. Use `MemPtrFree` to free the pointer. Pointers can also be resized, but be aware that because the underlying block is non-moveable, the resize can only be successful if the extra space is available directly after the block's location in the heap.

Memory Manipulation

Memory manipulation functions are few in the Palm SDK. You can use `MemSet` and `MemMove` to manipulate the contents of blocks of memory, and `MemCmp` allows you to do a byte comparison of two blocks of memory. Note that there isn't a separate `MemCopy` function; you must use `MemMove`.

Summary

As you've seen, using the Palm dynamic heap as a memory allocation resource is not difficult in practice, and there are actually just a very few function names and techniques to remember.

More importantly, the memory architecture of the Palm presents significant restrictions and challenges to the application designer, requiring a shift in mindset by the developer to accommodate the needs of the platform. Well-designed applications will work with the Palm OS design and not fight it. Those who produce such applications will be rewarded with happy users who can make a large number of diverse applications work on their devices simultaneously.

17 Understanding Palm OS Databases

IN THIS CHAPTER

- The Palm Database Model
- The Palm Data Manager
- Using the Data Manager to Create and Manipulate Databases
- Browsing Palm Databases with DBBrowse

If you've been reading the chapters in this book sequentially, you might have been wondering just how long I was going to put off discussing the topic of databases. The database support on the Palm is one of the most interesting and unique features of the Palm. It is also a topic that—sooner or later—most application developers need to understand because databases are one of the few ways for developers to retain information across program invocations. Without databases, information entered on the Palm would be lost as you switch from application to application.

The focus of this book, however, is to give Palm programmers the knowledge they need to use the Palm SDK in creating real-world applications. Although it is certainly informative to learn about the behind-the-scenes architecture, my feeling is that such a level of detail is not required to begin working with databases. If you want to learn more, the Palm SDK documentation, as well as a raft of articles and other resources, do a better and more accurate job of presenting that information than I ever could.

The next few chapters cover various aspects of Palm database programming. I start in this chapter by laying down what I hope is a firm foundation for understanding how databases and how the Palm Data Manager work.

In this chapter, you will

- Learn the underlying database model for the Palm
- Get an overview of the Palm Data Manager
- Learn how to create, open, and delete a database

- Learn how to browse the databases on your Palm
- Learn how to obtain information on installed databases

The Palm Database Model

If you are unfamiliar with handheld architectures, you might find the data storage system on a Palm device to be wildly different from anything you've experienced on desktop systems. In reality, if you understand how file systems are implemented on desktop systems, you'll realize that the Palm system is not so far off from those systems. The Palm SDK does expose a bit more of the actual storage model, but essentially, it is similar.

The first surprise is that there is no file system per se. On the Palm, there is no such thing as a generic binary or ASCII file that can be read from and written to as a stream of bytes. Rather, any non-volatile information is stored in databases.

What Is a Database?

A *database* is a collection of related records that's read from and written to by one or more Palm applications. A database is not a contiguous file of records; rather, it is a way to organize any number of individual records that can be distributed anywhere in the Palm's non-volatile memory. The records themselves are actually chunks of memory in the Palm's storage.

Databases are opened, closed, created, and deleted, just as files are on traditional file systems. For the most part, there is no need for the Palm programmer to be overly concerned with the actual distributed nature of Palm's databases. Simply use the Data Manager's functions and structures to interface with the database, and you should be okay.

What Is a Record?

The Palm Data Manager organizes all the records associated with a database and provides functions that allow the programmer to create, query, update, and delete those records. The notion of a record on the Palm is fairly loosely defined. Simply stated, a *record* is a memory chunk that has no implicit structure. The Data Manager has no data dictionary support and has no concept that corresponds to a "field." It is up to the programmer to write application code that correctly interprets the layout of a record. In fact, each record in a database is created according to the size specifications passed by the programmer. It is up to the programmer to correctly size the record according to the requirements of the parent database.

The Palm Data Manager

As you might expect, the Palm Data Manager presents a helpful layer of functionality on top of the physical storage of the data itself. Through approximately 30 functions, the Data Manager helps you create, open, close, delete, and browse databases. It also provides functions for creating, modifying, and deleting records.

Based on this description, you can consider the Data Manager akin to the single-tier or flat-file database systems commonly found on today's PC systems. Perhaps the most well-known examples of such database systems are dBASE and Btrieve. Like those systems, the Data Manager gives a Palm application direct access to the database itself and deals with records one at a time. Contrast this system with more advanced systems that support the relational model, referential integrity, SQL access, and set-oriented query and update operations.

Although vendors have announced the intention to bring some of the features of relational database management systems to the Palm, the limitations of the platform make the overhead imposed by these more advanced systems an undesirable tradeoff. The benefits measured against the speed and memory gains to be made by direct access just don't make it worth it. One of the costs (there's always a cost!) of this tradeoff is that the responsibility for maintaining the data's integrity lies with each application that manipulates the database.

Using the Data Manager to Create and Manipulate Databases

If your application will need to track and manage data across program executions, you need to create a database.

How Do I Create a Database?

Due to the lack of any data-dictionary–like knowledge, creating a database is actually a simple affair. You effectively create only a database header with which records will later be associated. The Palm function that creates a database is `DmCreateDatabase`. Take a look at what you have to give `DmCreateDatabase` for it to work its magic:

```
Err DmCreateDatabase (UInt cardNo, CharPtr nameP, Ulong creator,
➥ Ulong type, Boolean resDB);
```

Each of these parameters requires some explanation:

- `cardNo`—Aside from Handspring's Visor line of devices, Palm devices support only one memory card. It is important to distinguish a memory card in the context of database programming from expansion memory cards such as SD, MMC, and Compact Flash. Originally, Palm anticipated the potential need for multiple internal memory cards, so it exposed a way for the developer to specify on which memory card a database will reside. This scenario never really developed, and as of Palm OS 4.0, the Virtual File System (VFS) and Expansion Manager have set the standard for expanding memory storage on a Palm OS device. For more information on VFS and expansion memory cards, refer to Chapter 20, "Expansion Memory Cards and the Virtual File System."

- `nameP`—This is the human-readable name for the database that can be up to 31 characters long, not counting the `NULL` terminator. It is used in other Data Manager calls that identify and locate databases on the Palm.
- `creator`—This is the creator ID that uniquely identifies the parent application. For the purposes of this book, I am using a test ID, but you should visit Palm Computing's Developer Zone Web site, where you can register a guaranteed unique creator ID for your application. Note that this is passed as `Ulong`, cast from the four-byte character representation.
- `type`—You can actually create several types of databases on the Palm. For most applications, the only type you will use is `data`. You might be interested to know, however, that applications and resources are also stored in databases and have associated types.

  Like the creator ID, the type is actually passed as a `Ulong`, cast from the `data` definition.
- `resDB`—This is simply a Boolean flag that if true indicates a resource database is being created. Most calls to `CreateDatabase` will leave this as false.

Using Databases

Once you've created a database, the easiest way to open it is to use the `DmOpenDatabaseByTypeCreator` function. This function takes the same type and creator ID parameters that you used to create the database and also takes a mode parameter that indicates how you want to open the database. Mode constants can be specified using the bitwise `OR` operator and can be one or more of the values listed in Table 17.1.

TABLE 17.1 Mode Constants Used in Opening Databases

Mode Constant	Meaning
`DmModeReadWrite`	Opens the database for read-write access. Use this if you intend to insert, update, or delete database records.
`DmModeReadOnly`	Opens the database for read-only access.
`DmModeLeaveOpen`	Leaves the database open even after the application quits.
`DmModeExclusive`	Prevents other applications from opening the database while this application has it open.

The following example is standard code for determining whether a database exists by trying to open it. If the database does not exist, it is created by the application:

```
DmOpenRef s_dbRef = DmOpenDatabaseByTypeCreator ( 'data',
                                                  'GBBK',
                                                  dmModeReadWrite);


if ( 0 == s_dbRef )
{
   UInt        cardNo;
   // P5. card containing the application database
   LocalID    dbID;
   // P5. handle for application database
   UInt        dbAttrs;
   // P5. database attributes

       // Not there - create it now
   Err err = DmCreateDatabase ( 0,
                           "Glenn Database",
                           'GBBK',
                           'data',
                           false);

    ErrFatalDisplayIf ( err, "Could not create new database.");

    // Let's try again...
          s_dbRef = DmOpenDatabaseByTypeCreator ( 'data',
                                                      'GBBK',
                                                      dmModeReadWrite);
    }
```

The `DmOpenDatabaseByTypeCreator` call returns a `DMOpenRef`. This is a magic handle that must be used in most subsequent function calls against the database. In particular, when you are ready to close the database, call `DmCloseDatabase`, passing the `DMOpenRef`.

Browsing Palm Databases with DBBrowse

You can obtain information on the currently installed databases by walking the database list and calling `DmDatabaseInfo` on each database found.

You can walk the database list in one of two ways: by getting the number of databases with `DmNumDatabases` and iterating from 0 to the count or by using the

GetFirst/GetNext-like mechanism in DmGetNextDatabaseByTypeCreator. I prefer the latter because it does not force me to propagate the assumption that the memory card number is 0. The card number for the current database is returned to me by the function itself.

The key pieces of information returned by DmGetNextDatabaseByTypeCreator are the memory card number and the local database ID of the found database. This ID simply serves to uniquely identify the location of the database on the related memory card number and should be treated as a "magic value."

You can pass the memory card number and local database ID to DmDatabaseInfo. In exchange, the Data Manager will give you all sorts of interesting information about the database in question, including the name, the creation data, the version, and so on. You can obtain additional information by calling DmDatabaseSize, which returns the number of records and the size of the database in bytes.

Listing 17.1 contains the relevant source code for the sample database browser application DBBrowse. DBBrowse provides for demonstration purposes a minimal message-box–based user interface for walking the databases on your Palm database. For each database, DBBrowse displays the database name, the number of records, and the size of the database in bytes. Figure 17.1 shows DBBrowse in action.

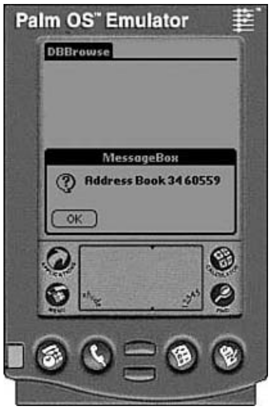


FIGURE 17.1 Walking the Palm database list.

LISTING 17.1 The Main FormCode for DBBrowse

```
/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
```

LISTING 17.1 Continued

```
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormButtonHandler (FormPtr formP, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      case MainOKButton:
      {
         Err err;
         DmSearchStateType state;
         UInt uCardNo;
         LocalID dbID;

         err = DmGetNextDatabaseByTypeCreator (
                                           true, // New search
                                           &state,
                                           'data',    // All types
                                           0,    // All creators
                                           true, // Only latest version
                                           &uCardNo, // Card number
                                           &dbID );
         while ( 0 == err )
         {

            char szName[32];

            Ulong ulNumRecords;

            DmDatabaseInfo ( uCardNo,
                             dbID,
                             szName,
                             0,
                             0,
                             0,
                             0,
                             0,
                             0,
```

LISTING 17.1 Continued

```
                           0,
                           0,
                           0,
                           0 );

            DmDatabaseSize ( uCardNo,
                             DbID,
                             UlNumRecords,
                             0, 0);

            StrPrintF ( szMessage, "%s %d", szName, ulNumRecords );

            PalmMessageBox ( MessageBoxAlert, szName );


            err = DmGetNextDatabaseByTypeCreator (
                                                false, // Not a new search
                                                &state,
                                                0,    // All types
                                                0,    // All creators
                                                true, // Only latest version
                                                &uCardNo, // Card number
                                                &dbID );
         }

         handled = true;
         break;
      }
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}
```

Summary

This chapter introduced the Palm data storage model and showed you how to create, open, and close databases. In addition, you did some spelunking through the Palm's list of databases and learned just what is taking up all that space on your Palm device.

The next few chapters continue the examination of database and file storage on Palm OS–based devices, including record management and category handling. I conclude the data-management chapters with an explanation of the Palm OS expansion card subsystem, the Virtual File System.

18

Palm Databases and Record Management

IN THIS CHAPTER

In the preceding chapter, I introduced the Palm data storage model and walked you through how to use the Database Manager SDK functions to create, open, and close databases. In addition, I briefly covered what a record in a Palm database is.

This chapter presents a closer look at how to deal with the most important component of a Palm database: the records themselves. In this chapter you will

- Learn how records are associated with databases
- Understand what record management functionality is available in the Palm Database Manager
- Learn how to query and iterate through the records in a database
- Learn how to create, edit, and delete a record

I close out the chapter with a simple database application that illustrates some of these concepts in action.

How Are Databases and Records Connected?

As you saw in Chapter 17, "Understanding Palm OS Databases," a database is essentially a special block of memory that serves as a "header" record. The actual records in the database are allowed to exist as individual chunks strewn throughout the Palm device's non-volatile memory storage. There is no need for individual records to be located contiguously, or even remotely near the database header or other records in the same database card.

This is possible because the Palm database header tracks everything about the database, including not only the number of records, but also a list of all the records themselves. Think of the database header as the troop leader for the database.

The list of records in the database header does not contain the records themselves. Rather, each entry in the record list is a special eight-byte block, which contains the local ID of the record, some attribute bits that track the record's state, a category ID (more on categories in Chapter 19, "Categories"), and a special unique ID for the record.

Lest you get confused, let's differentiate between a record's local ID and its unique ID. The local ID, which I briefly described in the preceding chapter, is used to locate the handle of a record anywhere on a specific "memory card" (for more on memory card numbers, please see Chapter 16, "Palm OS Memory Management"). The Palm documentation states that all records for a database must reside on the same memory card as the database header.

The record's unique ID, on the other hand, is used to establish uniqueness within the database itself and is unrelated to the memory used to store the record. In addition, the unique ID is used during desktop synchronization to track records on both sides of the sync process. The Palm documentation states that this unique ID will remain the same no matter how many times a record is updated. Therefore, if you want to refer to another database record within your record (for example, if you have a note associated with a record), you might want to store the unique ID as the "foreign key."

As it turns out, the database header, with the help of its list of record entries, has all the information it needs to locate and get the memory handle for any record in the database. The records themselves contain only the data portion of the record; all control information is embedded in the database header.

Using the Data Manager with Records

The Data Manager hides much of the detail I just described beneath a set of functions that allow you to query, edit, create, delete, and even sort database records.

A key point to understand is that all these operations are performed "in place." By this, I mean that when you edit a record, you are physically modifying the record as it exists on the device. No temporary copies are made. The same goes for queries; there is no such thing as a result set. The query functions take you from record to record on the physical device. This is why the Palm documentation, as well as many of the other resources on programming the Palm, often lump together the concept of databases and memory management. When you manipulate and iterate through database records, you are dealing directly with memory chunks.

My personal view on this way of looking at databases as memory is that, although it is instructional to understand the implementation of databases on the Palm device, it is not a great idea conceptually to assume that when you edit a database record you are directly editing memory. The Database Manager is there for a reason, and one of those reasons is to provide a meaningful representation of databases on a device with limited capacity and storage. My own experience with other platforms and with layers of abstraction tells me that allowing assumptions such as "database record equals memory chunk" to creep into your thinking (and ultimately your program design) might cause you trouble down the road when it just isn't true anymore.

The rest of this chapter covers how to accomplish specific tasks using Database Manager. Keep in mind that the Database Manager encompasses a rather large expanse of functionality, more than can be covered here. For documentation on all of the Database Manager APIs, please refer to the Palm OS developer documentation.

Querying Database Records

If you have a database in your hand, odds are that you will want to look at what's inside it. With the Data Manager, the key to opening the door to your records is the `DmOpenRef` returned from `DmOpenDatabase`. With a `DmOpenRef` handle in your possession, there are several ways to get at the records in a database:

- `DmFindRecordByID`—Given a unique record ID (not a local ID), this returns the index of the matching record.
- `DmGetRecord`—Given an index to a record, this returns the handle to the record's contents. It also effectively "locks" the record, preventing another call to `DmGetRecord` on the same record until `DmReleaseRecord` is called and the record is released. This locking mechanism is enabled by setting an attribute in the database header's record entry, called the *busy bit*.
- `DmQueryRecord`—Similar to `DmGetRecord` in that given an index to a record, it returns the record's handle. The difference is that the record is not "locked" after the call, and calling this function will always return successfully even if another piece of code has locked the record with `DmGetRecord`. A record handle returned from `DmQueryRecord` cannot be used to write to the record—you need to use `DmGetRecord` instead. Finally, because the database header is not updated, this is the fastest way to get a record handle.
- `DmQueryNextInCategory`—Assuming you are iterating through all the records belonging to a specific category, this function returns the handle to the next record after the passed-in record index. I cover categories in depth in Chapter 19.

- `DmSeekRecordInCategory`—A more powerful version of `DmQueryNextInCategory`, this function allows you to search forward or backward within a category and also allows you to jump records from the current record in either direction.

Manipulating Database Records

The Database Manager provides many functions that allow you to create, modify, or delete records in a database. The following sections provide an overview of each of these.

Creating a Record

You create records by calling `DmNewRecord`. `DmNewRecord` accepts an index within the database where you want the record placed and a record size. If you specify an index that is greater than the last record in the database, the record is appended to the end.

Modifying a Record

The modification functions require a pointer to a record. With a handle to a record, you obtain the pointer by using `MemHandleLock`. (In my humble opinion, mixing the database and memory routines is a flaw in the Data Manager APIs.) Other than to lock and unlock the record handle, you should never use the memory manager functions to write to a database record. Many hard-to-find problems can be traced to incorrectly intermixing calls to the memory manager and database APIs, so it is important to remember that they are not interchangeable.

The following functions should be used when modifying a database record:

- `DmWrite` writes a set of bytes into a record at a specific offset. This is akin to a `MemCopy` operation and can be used to update the entire record or a portion of it (a field, for example).
- `DmSet` writes a specific value into a contiguous set of bytes in a record, starting at a byte offset within the record. For instance, if you want to clear a string field of length 6 at offset 10 within a record, so that it contained all null characters, you call `DmSet (pRecord, 10, 6, '\0');`.
- `DmStrCopy` is similar to `DmWrite` but assumes you are copying a string value into a field and accounts for null termination.

Deleting a Record

`DmDeleteRecord` deletes the record from the database, which according to the documentation frees the record itself and sets the special delete bit in the database header's associated record entry. On the next synchronization, the record is then deleted from the device.

Getting Record Information

`DmRecordInfo` is primarily used for obtaining a record's unique ID.

Fish List: A Record Management Sample Application

When my son Nicholas was four years old, he loved to learn about fish: squid, octopuses, sharks, and eels. He knew all their names and could tell you all you ever wanted to know about how they eat, swim, and live in the ocean.

Partly as a tribute to Nicholas's vast undersea knowledge, and partly as a way for him to teach me more about the creatures living in the deep dark sea, I created a simple application that keeps the names of the fish I want to remember in a special fish database. The user interface for the fish database allows me to see the names of all the fish I've entered so far and lets me add new fish via a simple pop-up `AddFish` form. Figure 18.1 shows the Fish List in the background with the `AddFish` form in the foreground.

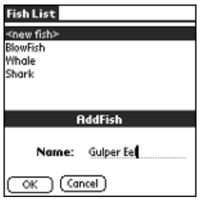


FIGURE 18.1 Adding a new denizen of the deep to your database.

Although my Fish List application appears simple, there are enough database operations that I decided to isolate the database-manipulation functionality from the main form's user-interface operation. Although this book does not primarily discuss good program design, this separation is a good practice to follow and provides a clean separation between your application's user interface and the underlying data model.

The new module is `reco_db.cpp`, and Listing 18.1 contains the main database interface functions.

LISTING 18.1 The Source Code for the Fish List's Record Management Interface

```
/*
   RECO_DB.CPP

   Records application db functions

   Copyright (c) 1999 Bachmann Software and Services, LLC
   Author: Glenn Bachmann
*/

// system headers
#include <Palmos.h>
#include <SysEvtMgr.h>

// application-specific headers
#include "record.h"
#include "reco_res.h"

#define RECORD_DB_TYPE  ('Data')
#define RECORD_DB_NAME  ("FishDB")

static DmOpenRef           s_dbFish;
static MemHandle        s_hRec;
static char *       s_pText;
static UInt16               s_recordNum = 0;

/*
   FishGetCount:
   Returns the count of records in the fish database
   Parms: none
   Returns: count

*/
UInt16
FishGetCount ( void )
{
   UInt16 uFishCount = 0;
   if ( s_dbFish )
   {
      uFishCount = DmNumRecords ( s_dbFish );
   }
   return uFishCount;
```

LISTING 18.1 Continued

```
}

/*
   FishNew:
   Creates a new Fish record
   Parms: none
   Returns: true if record created
*/
Boolean
FishNew ( void )
{
       MemHandle hFish;
       Err            err = 0;
       Char           zero = 0;
   UInt16     uIndex = 0;

       // Create a new record in the Fish database
       hFish = DmNewRecord ( s_dbFish, &uIndex, FISH_LENGTH);

   if ( hFish )
   {
          MemPtr          p;

       // get a pointer to the mem block associated with the new record
       p = (MemPtr)MemHandleLock (hFish);

       // Init the record by writing a zero
       err = DmWrite ( p, 0, &zero, 1 );

       // Unlock the block of the new record.
       MemPtrUnlock ( p );

       // Release the record to the database manager.

       DmReleaseRecord ( s_dbFish, uIndex, true);

       // Remember the index of the current record.
       s_recordNum = 0;

      return true;
   }
   else
```

LISTING 18.1 Continued

```
   {
      // handle unexpected err
          ErrFatalDisplayIf ( err, "Could not create new record.");

      return false;
   }
}

/*
   FishGetFish:
   Gets a handle to a Fish record at index uIndex
   Parms: uIndex = 0 based index of record in db
   Returns: NULL or a valid record handle
*/
MemHandle
FishGetFish ( UInt16 uIndex )
{
     Err                    err;

   s_recordNum = 0;

   if ( s_dbFish )
   {
      err = DmSeekRecordInCategory ( s_dbFish,
                                     &s_recordNum,    // start at 0
                                     uIndex,
                                     dmSeekForward,   // seek forward
                                     dmAllCategories);   // all categories

      // get the record
       s_hRec = (MemHandle)DmGetRecord ( s_dbFish, s_recordNum );
      if ( s_hRec )
      {
         // lock it down and get a ptr
       s_pText = (char *)MemHandleLock ( s_hRec );
      }
   }

   return s_hRec;
}

/*
```

LISTING 18.1 Continued

```
   FishReleaseFish:
   Releases the currently edited fish back to the database
   Parms: none
   Returns: none
*/
void
FishReleaseFish ( void )
{
   // unlock the record
   if ( s_hRec )
   {
        MemHandleUnlock ( s_hRec );
   }

     // Release the record, not dirty.
      DmReleaseRecord ( s_dbFish, s_recordNum, false);

   // reset the current record number
   s_recordNum = 0;

   return;
}


/*
   FishGetName:
   Gets the ptr to the fish name field. Since the fish record is simply one
   field, just returns the whole record
   Parms: none
   Returns: ptr to name
*/

char *
FishGetName ( void )
{
   return s_pText;
}

/*
   FishSetName:
   Sets the name of the currently edited fish record
   Parms: pName = ptr to new name
```

LISTING 18.1 Continued

```
   Returns: none
*/
void
FishSetName ( char * pName )
{
     // Lock down the block containing the record
   if ( s_hRec )
   {
       MemPtr        p;

          p = (MemPtr)MemHandleLock ( s_hRec );

    // write the name value. Add one to strlen to make sure its null termed
      Err err = DmWrite ( p, 0, pName, StrLen ( pName ) + 1 );

       MemPtrUnlock ( p );
   }

   return;
}

/*
   FishOpen:
   Opens the Fish database, retaining a ref number to the db for further
   operations
   Parms: none
   Returns: none
*/

void
FishOpen ( void )
{
   // open the db
    s_dbFish = DmOpenDatabaseByTypeCreator ( RECORD_DB_TYPE,
                                           RECORD_FILE_CREATOR,
                                           dmModeReadWrite);

    if ( 0 == s_dbFish )
    {
      // not there - create it now
        Err err = DmCreateDatabase ( 0,
                                RECORD_DB_NAME,
```

LISTING 18.1 Continued

```
                               RECORD_FILE_CREATOR,
                               RECORD_DB_TYPE,
                               false);

      // handle unexpected err
         ErrFatalDisplayIf ( err, "Could not create new database.");

     // lets try to open it again...
         s_dbFish = DmOpenDatabaseByTypeCreator ( RECORD_DB_TYPE,
                                                   RECORD_FILE_CREATOR,
                                                   dmModeReadWrite);

      // handle unexpected err
      if ( 0 == s_dbFish )
      {
        ErrFatalDisplayIf ( true, "Could not open new database.");
      }
   }

   return;
}

/*
   FishClose:
   Closes the fish db
   Parms: none
   Returns: none
*/

void
FishClose ( void )
{
   if ( s_dbFish )
   {
      DmCloseDatabase ( s_dbFish );
      s_dbFish = 0;
   }
   return;
}
```

The design goal for the new database source-code module was to remove the need for the main form user-interface code to fiddle with `DmOpenRefs`, hide idiosyncrasies of the database functions, and enforce a single way for the application to interface with the fish database. The FishDB API considerably simplifies the code for the main form.

Speaking of the main form, Listing 18.2 shows how it presents the list of fish in the database and the AddFish pop-up form. You'll note that because I separated the database operations, the code is not really all that different from the Shopping List application developed in Chapter 10, "Giving the Users a Choice: Lists and Pop-up Triggers."

LISTING 18.2 The Source for Fish List's Main Form

```
/*
   RECO_MAI.CPP
   Main form handling functions.
   Copyright (c) Bachmann Software and Services, 1999-1802
   Author: Glenn Bachmann
*/

// system headers
#include <Palmos.h>
#include <SysEvtMgr.h>

// application-specific headers
#include "record.h"
#include "reco_res.h"

static void    MainFormInit (FormPtr formP);
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);

// local helper functions
void MainListFill (FormPtr pForm);
void MainListDrawFunction (Int16 itemNum, RectanglePtr bounds, char **pUnused);

Boolean AddFish ( void );
Boolean AddFish_EventHandler (EventPtr pEvent);

/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
```

LISTING 18.2 Continued

```
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;

   switch (eventP->eType)
   {
      case menuEvent:
      {
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormMenuHandler (formP, eventP);
         break;
      }


      case ctlSelectEvent:
      {
         // A control button was tapped and released
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormButtonHandler (formP, eventP);
         break;
      }

      case frmOpenEvent:
      {
         // main form is opening

         FormPtr formP = FrmGetActiveForm ();

         MainFormInit (formP);

         FrmDrawForm (formP);

         handled = true;
         break;
      }

      default:
      {
         break;
      }
```

LISTING 18.2 Continued

```
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   formP - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr pForm)
{
   // set the controls to an initial state

   // walk through fish records and fill list
   MainListFill ( pForm );

   // set the drawing function for the main list
   UInt16 wIDMainList = FrmGetObjectIndex (pForm, MainFishList );
   ListPtr pCtlMainList = (ListPtr) FrmGetObjectPtr (pForm, wIDMainList);
   LstSetDrawFunction ( pCtlMainList, MainListDrawFunction );
}

/*
   MainFormMenuHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormMenuHandler (FormPtr /*formP*/, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.menu.itemID)
   {
      default:
      {
         handled = false;
```

LISTING 18.2 Continued

```
         break;
      }
   }
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/

Boolean
MainFormButtonHandler (FormPtr pForm, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      case MainNewButton:
      {
         if ( AddFish () )
         {
            MainListFill ( pForm );
         }

         handled = true;
         break;
      }

      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}
```

LISTING 18.2 Continued

```
/*
   MainListFill:
   Fills the fish list
   Parms: pForm - pointer to main form
   Returns: none
*/
void
MainListFill (FormPtr pForm)
{
   // get a ptr to the fish list
   UInt16 wIDFishList = FrmGetObjectIndex (pForm, MainFishList );
   ListPtr pCtlFishList = (ListPtr) FrmGetObjectPtr (pForm, wIDFishList);

   // get the count of available main list items
   Int16 iFishCount = FishGetCount ();

   // fill the main list
   LstSetListChoices ( pCtlFishList, NULL, iFishCount );

   // redraw the list
   LstDrawList (pCtlFishList);

   return;
}

/*
   MainListDrawFunction:
   Draws a single item in the main list
   Parms: itemNum = 0 based index of item to draw
          bound = ptr to rectangle providing drawing bounds
          pUnused = ptr to char string for item to draw. In this app, we
          never give the actual item data to the list, so we ignore this parm
   Returns: none
*/
void
MainListDrawFunction (Int16 itemNum, RectanglePtr bounds, char **pUnused)
{
   FormPtr pForm = FrmGetActiveForm ();

   char *         pText;
   MemHandle      hFish = NULL;
```

LISTING 18.2 Continued

```
   // find the record at this index
   FishGetFish ( itemNum );

   // get the fish name
   pText = FishGetName ();

   // draw it
     if ( pText )
   {
      WinDrawChars ( pText, StrLen (pText), bounds->topLeft.x,
      bounds->topLeft.y);
   }

   // release it
   FishReleaseFish ();
   return;
}


/*
   AddFish:
   Invokes the Add Fish popup form
   Parms:   none
   Return:  none
*/
Boolean AddFish ( void )
{
   // initialize the form
   FormPtr  pForm = FrmInitForm (AddFishForm);

   // create a new fish record
        if ( FishNew () )
   {
      // edit it!
      MemHandle hRecord = FishGetFish ( 0 );

      // create a mem buffer for the new fish name
      MemHandle hText = MemHandleNew ( FISH_LENGTH );
      char * pText = (char *)MemHandleLock (hText);

      // default the name of the fish
      StrCopy ( pText, "<new fish>" );
```

LISTING 18.2 Continued

```
       MemPtrUnlock ( pText );



      UInt16 wIDField = FrmGetObjectIndex ( pForm, AddFishNameField );
      FieldPtr pField = (FieldPtr) FrmGetObjectPtr ( pForm, wIDField);

      // copy the text into the edit field
           FldSetTextHandle ( pField, (MemHandle)hText);

      // display the dialog
      if ( FrmDoDialog (pForm) )
      {
         // get the name from the dialog
         hText = FldGetTextHandle (pField);
         char * pText = (char *)MemHandleLock (hText);

         // set it in the record
         FishSetName ( pText );

         MemPtrUnlock ( pText );
      }

      // un-edit the fish record, saving changes
      FishReleaseFish ();

      // destroy the form
      FrmDeleteForm (pForm);

      return true;
   }
   return false;
}
```

Summary

With the ability to create and manage both records and databases, you are well on your way to being able to add database support to your Palm application. You might find, as I did, that after the first couple of attempts, much of the code required for database support becomes fairly boilerplate and can be folded away in a common layer of utility functions that you can use from project to project.

19 Categories

IN THIS CHAPTER

The Palm OS has built-in support for a concept called a record *category*. Categories represent a general-purpose way for the users to group logically related records within a Palm database.

If you've worked with a Palm device, you may have found that the capability to use categories for organizing personal information is quite useful. On the other hand, I know Palm users who do not use the category system at all.

Unfortunately, for such a seemingly simple concept, categories are a rather complicated subject for programmers to deal with (perhaps unnecessarily so). Implementing category support in an application requires making non-trivial changes in three areas of your code: database creation, record management, and user interface.

In this chapter, you'll learn

- What categories are and how they are used in built-in Palm applications
- All about category support in Palm SDK
- About the programming tasks involved in adding category support to your application

At the end of the chapter, I rework the Fish List program from Chapter 18, "Palm Databases and Record Management," to add category support.

What Is a Category?

On the Palm, each database record contains a dedicated storage area to hold an application-defined category. To be precise, the record entry in the database header contains a category field. This field is present in the record entry no matter how you create your database, and regardless of whether you support categories in your application.

Categories are application-specific. At first glance, because many of the category names used in the built-in applications are the same (example “Business”), it may appear that the built-in applications might all share the same categories. But that is not the case. Each Palm application is free to define any category names that it (or its users) deem appropriate. Unless the underlying databases are shared, multiple applications will not have access to each other’s categories. This at times can be annoying to users of the built-in applications, and you could argue it would be nice to not have to redefine the same categories to be used in the Address Book, to-do list, Datebook, and so on.

Each Palm application can define up to 16 categories for its own use. These categories can be completely reserved for the user to define, or they can be partially or even completely defined and fixed by the application developer.

Finally, categories are limited to a single “level”; they cannot be nested or defined in a hierarchical manner. Depending on your application, this might not be a significant functional restriction for your users.

Figures 19.1 through 19.3 illustrate the user interface for category support in the Address Book application.

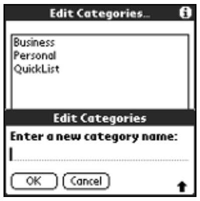


FIGURE 19.1 The main Address Book view with the categories dropped down.

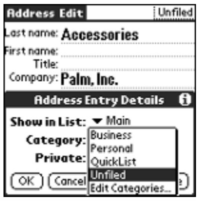
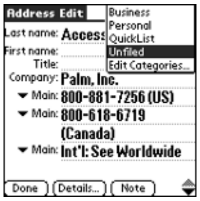


FIGURE 19.2 Setting the category for an Address Book record.



Last name: Access
First name:
Title:
Company: Palm, Inc.
Main: 800-881-7256 (US)
Main: 800-618-6719 (Canada)
Main: Int'l: See Worldwide

FIGURE 19.3 The standard Edit Categories form.

Palm SDK Support for Categories

As I hinted in this chapter's introduction, category support in the Palm SDK falls into several areas of functionality, including

- Database initialization of categories
- Database functions that use categories
- Standard category user interfaces
- `CategorySelect`

If that's not enough for you, you need to perform some steps in Constructor in order to properly initialize categories in your application.

The next section examines each of these areas in more detail.

How to Add Category Support to Your Application

Next, we walk through these areas of support and learn about the development steps required for each area. This includes how to incorporate categories in your database records, adding code to handle categories, and how to use the standard category user interface that Palm OS provides.

Preparing Your Database for Categories

The preparation is certainly the most complicated step, so it's good to discuss it first. The first requirement for providing category support in your application is to create a special `AppInfo` record and set the local ID for the record in your database header when the database is created.

An `AppInfo` block has the following structure:

```
typedef struct {
    UInt16 renamedCategories;
    Char categoryLabels [dmRecNumCategories][dmCategoryLength];
    UInt8 categoryUniqIDs[dmRecNumCategories];
    UInt8 lastUniqID;
    UInt8 padding;
} MyAppInfo;
```

Although you must define such a structure in your application, and you must allocate a database record of this size, you actually do not need to be concerned with the internal structure of the record. Assuming you do all the right things in the right order, Palm takes care of the `AppInfo` structure.

The magic sequence for adding an `AppInfo` record to your database follows:

1. Launch Constructor and load your application's resources.
2. Find App Info String Lists, and create a new string list resource.
3. Double-click the resource to edit it.
4. In the App Info String List window, repeatedly add new string list entries until there are 16 in total, including the `Unfiled` entry.
5. Type values for as many of the categories you want to predefine for your users.

After you've saved your Constructor session, you are ready to add the code to initialize your database:

1. Use `DmOpenDatabaseInfo` to obtain the memory card number and local ID associated with your open database.
2. Call `DmDatabaseInfo` to obtain the current `AppInfo` ID, if any.
3. If `AppInfo` was not previously defined, use `DmNewHandle` to allocate a new database handle large enough to hold the `AppInfo` structure.
4. Obtain the local ID associated with the newly created database handle by using the `MemHandleToLocalID` utility function.
5. Stuff the `AppInfo` handle's local ID into the `AppInfoID` database header attribute by calling `DmSetDatabaseInfo`.
6. Initialize the `AppInfo` structure by using `DmSet` to zero out all members.
7. Call `CategoryInitialize`, passing in the `AppInfo` block and the resource ID of your App Info String List resource you created in Constructor.

All of this code should be wrapped up in a function that gets called in tandem with the creation of your database.

Whew! The good news is that this is the kind of routine you can write once, call once from your code, and never worry about again. The bad news is that if you get it wrong, your database and category support will not work correctly.

Adding Category Awareness to Your Database Calls

Basically, categories affect the way you perform database operations in two ways. The first is that you can use category-aware database query and seek functions. In particular, `DmNumRecordsInCategory` will give you the number of records that fall into a given category. `DmSeekRecordInCategory` will limit its record seeking to a specific category.

The second thing to know is that you are responsible for initializing and modifying a record's category based upon user selection. For this, you use `DmSetRecordInfo` to pass a `UInt` into which the desired category is `OR`'ed.

Supporting Categories in Your User Interface

The final area you need to handle in supporting categories is your user interface. It is a good idea to follow the visual examples set by the built-in applications. The main forms of those applications all allow filtering of the main record lists via a categories drop-down list in the upper-right corner of the display. (For complete instructions on how to define a drop-down list using pop-up triggers, see Chapter 10, "Giving the Users a Choice: Lists and Pop-up Triggers.")

In addition, each application offers the users the capability to assign a record to a category as part of the record-editing user interface. The application also supports the capability for the users to define or modify the category list for the application.

Aside from the tediousness of defining the user interface in Constructor, all of this user-interface functionality is actually fairly simple to program. Palm Computing has graciously provided a handy function called `CategorySelect`. Here's how you use it: When the pop-up trigger associated with either your main list view or your record edit form is fired, you pass the IDs of the associated pop-up trigger, list, and other parameters. `CategorySelect` takes it from there, automatically filling the drop-down list with the available categories and even guiding the users through the Edit Categories process. If it turns out the users selected a category, the ID of the selected category is returned to you. That's all there is to it.

Adding Categories to the Fish List Application: What Kind of Fish Was That?

To illustrate the changes needed in an application to support categories, I modified the Fish List program introduced in Chapter 20, “Expansion Cards and the Virtual File System,” so for each fish you can now store the type of fish you are entering as a category association. The users are free to categorize their entries any way they want: Small, Medium, or Large, Crustaceans, Mollusks, and Mammals, or whatever. Fish List supports the standard look and feel of the built-in applications’ category user interfaces. Figure 19.4 shows how Fish List lets the users choose which category of fish to display in the main form, whereas Figure 19.5 shows the user interface for setting the category for a selected fish.

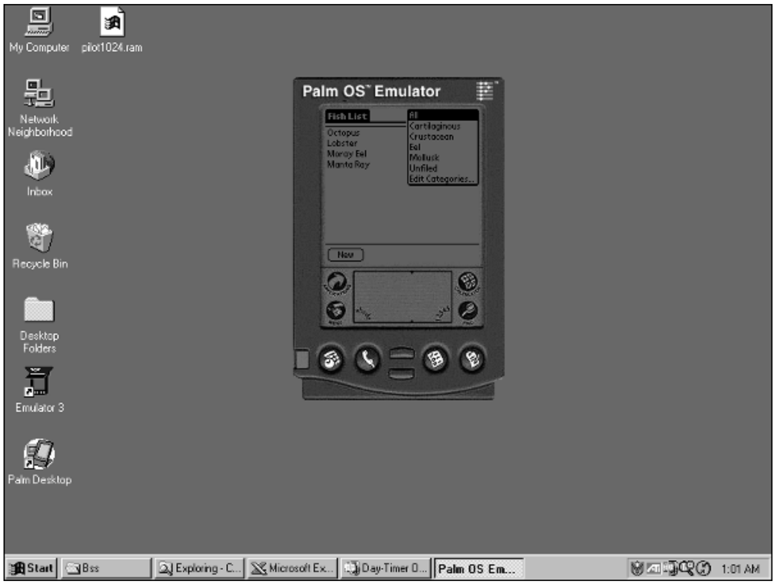
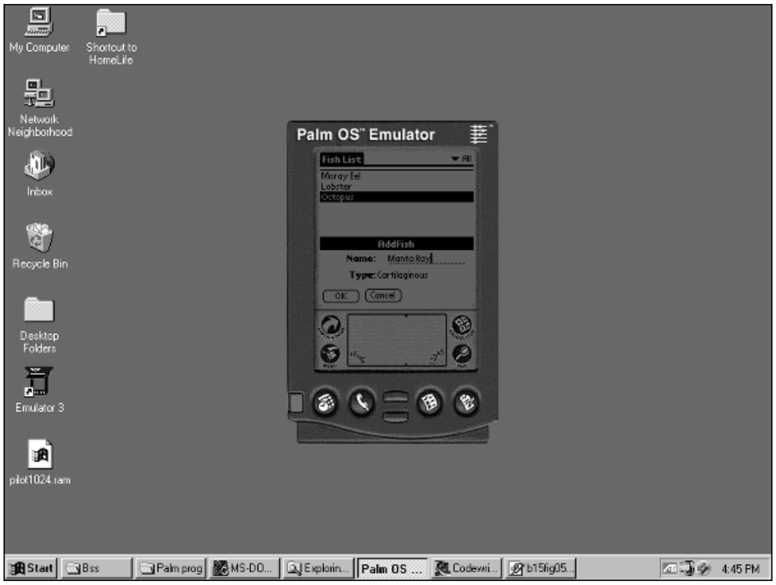


FIGURE 19.4 Choosing a category to filter fish by type.

Listings 19.1 and 19.2 represent the new Fish List implementation. Listing 19.1 is `cate_db.cpp`, which contains the most changes for categories, including a `FishInitAppInfo` function that takes care of the messy `AppInfo` initialization task.



Name: Manta Ray
Type: Cartilaginous

FIGURE 19.5 Setting the category for a manta ray.

LISTING 19.1 Database-Related Code for Fish List, with Category Support

```
/*
   CATE_DB.CPP

   Categories application db functions

   Copyright (c) 1999 Bachmann Software and Services, LLC
   Author: Glenn Bachmann
*/

// System headers
#include <Pilot.h>
#include <SysEvtMgr.h>

// Application-specific headers
#include "category.h"
#include "cate_res.h"
```

LISTING 19.1 Continued

```
#define CATEGORY_DB_TYPE  ('Data')
#define CATEGORY_DB_NAME  ("FishDB")

static Word                s_wCurrentCategory = dmAllCategories;
static char                s_szCategoryName[dmCategoryLength];
static DmOpenRef        s_dbFish;
static Handle        s_hRec;
static CharPtr       s_pText;
static UInt               s_recordNum = 0;

static Err     FishInitAppInfo (DmOpenRef pDB);

/*
   FishGetCount:
   Returns the count of records in the fish database
   Parms: none
   Returns: count

*/

UInt
FishGetCount ( void )
{
   UInt uFishCount = 0;
   if ( s_dbFish )
   {
      uFishCount = DmNumRecordsInCategory ( s_dbFish,
      ➥s_wCurrentCategory );
   }
   return uFishCount;
}

/*
   FishNew:
   Creates a new Fish record
   Parms: none
   Returns: true if record created
*/

Boolean
FishNew ( void )
{
```

LISTING 19.1 Continued

```
    Ptr        p;
    VoidHand hFish;
    UInt        uIndex = 0;
    Err        err;
    Char        zero = 0;
    UInt        attr;

    // Create a new record in the Fish database
    hFish = DmNewRecord ( s_dbFish, &uIndex, FISH_LENGTH);

   if (hFish)
   {
       // Get a pointer to the mem block associated with the new record
       p = (Ptr)MemHandleLock (hFish);

       // Init the record by writing a zero
       err = DmWrite ( p, 0, &zero, 1 );

       // Unlock the block of the new record.
       MemPtrUnlock ( p );

      // Obtain the record's attribute info, and set it to reflect the
      // currently selected category
       DmRecordInfo ( s_dbFish, uIndex, &attr, NULL, NULL);

       attr &= ~dmRecAttrCategoryMask;       // Remove all category bits

       if ( s_wCurrentCategory == dmAllCategories)
       {
           attr |= dmUnfiledCategory;
       }
       else
       {
           attr |= s_wCurrentCategory;
       }

       DmSetRecordInfo ( s_dbFish, uIndex, &attr, NULL);


       // Release the record to the database manager.
       DmReleaseRecord ( s_dbFish, uIndex, true);
```

LISTING 19.1 Continued

```
       // Remember the index of the current record.
       s_recordNum = 0;
      return true;
   }
   else
   {
      return false;
   }

}

/*
   FishGetFish:
   Gets a handle to a Fish record at index uIndex
   Parms: uIndex = 0 based index of record in db
   Returns: NULL or a valid record handle
*/

Handle
FishGetFish ( UInt uIndex )
{
    Err            err;

   s_recordNum = 0;

   if ( s_dbFish )
   {
      err = DmSeekRecordInCategory ( s_dbFish,
                                     &s_recordNum,
                                     uIndex,
                                     dmSeekForward,
                                     s_wCurrentCategory);

      // Get the record
       s_hRec = (Handle)DmGetRecord ( s_dbFish, s_recordNum );
      if ( s_hRec )
      {
         // Lock it down and get a ptr
          s_pText = (CharPtr)MemHandleLock ( s_hRec );
      }
```

LISTING 19.1 Continued

```
   }

   return s_hRec;
}

/*
   FishReleaseFish:
   Releases the currently edited fish back to the database
   Parms: none
   Returns: none
*/

void
FishReleaseFish ( void )
{
   if ( s_hRec )
   {
       MemHandleUnlock ( s_hRec );
   }

    // Release the record, not dirty.
    DmReleaseRecord ( s_dbFish, s_recordNum, false);

   s_recordNum = 0;

   return;
}

/*
   FishUpdate:
   Commits changes to the category information back to the record
   Parms: none
   Returns: none
*/

void
FishUpdate ( void )
{
    UInt        attr;

    DmRecordInfo (s_dbFish, s_recordNum, &attr, NULL, NULL);
```

LISTING 19.1 Continued

```
    attr &= ~dmRecAttrCategoryMask;

    if ( s_wCurrentCategory == dmAllCategories)
    {
        attr |= dmUnfiledCategory;
    }
    else
    {
        attr |= s_wCurrentCategory;
    }

   // Update the attributes
    DmSetRecordInfo ( s_dbFish, s_recordNum, &attr, NULL);
   return;
}

/*
   FishSelectType:
   Performs Category Selection in order to choose the fish type
   If the category is changed, updates the current category
   Parms: pForm = parent form
          pTypeID = out param for new type
          wTriggerID = id of popup trigger control
          wListID = id of listbox
   Returns: true if selection made
*/

Boolean
FishSelectType ( FormPtr pForm, Word *pTypeID, Word wTriggerID,
➥Word wListID )
{
   Boolean bChanged = false;

    Word category;
    Boolean bEdited;

    // Process the category pop-up list
   // Save the current one in a temp
    category = s_wCurrentCategory;

    bEdited = CategorySelect ( s_dbFish,
                                       pForm,
```

LISTING 19.1 Continued

```
                                       wTriggerID,
                                          wListID,
                                       true,
                                       &category,
                                       s_szCategoryName,
                                       1,
                                       0);

    if (bEdited || (category != s_wCurrentCategory))
    {
      // Category is new - save it and refill the fish list!
        s_wCurrentCategory = category;

      *pTypeID = category;
      return true;
   }
   return false;
}

/*
   FishGetCurrentType:
   Obtains the name of the currently selected category/fish type
   Parms: pType= buffer to receive name of type
   Returns: none
*/

void
FishGetCurrentType ( CharPtr pType )
{
    CategoryGetName ( s_dbFish,
                     s_wCurrentCategory,
                     s_szCategoryName);

   StrCopy ( pType, s_szCategoryName );
   return;
}

/*
   FishGetName:
   Gets the ptr to the fish name field. Because the fish record is
   simply one field, just returns the whole record
   Parms: none
```

LISTING 19.1 Continued

```
   Returns: ptr to name
*/

CharPtr
FishGetName ( void )
{
   return s_pText;
}

/*
   FishSetName:
   Sets the name of the currently edited fish record
   Parms: pName = ptr to new name
   Returns: none
*/

void
FishSetName ( CharPtr pName )
{
    // Lock down the block containing the record
   if ( s_hRec )
   {
       Ptr        p;

       p = (Ptr)MemHandleLock ( s_hRec );

      // Write the name value. Add one to strlen to make sure it's
      // null terminated
       Err err = DmWrite ( p, 0, pName, StrLen ( pName ) + 1 );

       MemPtrUnlock ( p );
   }

   return;
}

/*
   FishOpen:
   Opens the Fish database, retaining a ref number to the db for further
   operations
   Parms: none
   Returns: none
```

LISTING 19.1 Continued

```
*/

void
FishOpen ( void )
{
   // Open the db
    s_dbFish = DmOpenDatabaseByTypeCreator ( CATEGORY_DB_TYPE,
                                          CATEGORY_FILE_CREATOR,
                                          dmModeReadWrite);

    if ( 0 == s_dbFish )
    {
       UInt        cardNo;
       LocalID    dbID;
       UInt        dbAttrs;

      // Not there - create it now
        Err err = DmCreateDatabase ( 0,
                               CATEGORY_DB_NAME,
                               CATEGORY_FILE_CREATOR,
                               CATEGORY_DB_TYPE,
                               false);

      // Handle unexpected err
        ErrFatalDisplayIf ( err, "Could not create new database.");

        // Let's try to open it again...
       s_dbFish = DmOpenDatabaseByTypeCreator ( CATEGORY_DB_TYPE,
                                                CATEGORY_FILE_CREATOR,
                                              dmModeReadWrite);

        // Store category info in the application's information block.
        FishInitAppInfo(s_dbFish);
   }

    // Get the name of the current category from the app info block.
   if ( s_dbFish )
   {
       CategoryGetName ( s_dbFish, s_wCurrentCategory, s_szCategoryName);
   }
   return;
```

LISTING 19.1 Continued

```
}

/*
   FishClose:
   Closes the fish db
   Parms: none
   Returns: none
*/

void
FishClose ( void )
{
   if ( s_dbFish )
   {
       DmCloseDatabase ( s_dbFish );
   }
   return;
}

/*
   FishInitAppInfo:
   Initializes the App Info portion of the database header and sets
   the category area
   Parms: pDB = ptr to fish db
   Returns: 0 = success

*/

Err
FishInitAppInfo (DmOpenRef pDB)
{
   UInt cardNo;
   VoidHand handle;
   LocalID dbID;
   LocalID appInfoID;
   FishAppInfo * pAppInfo;
   FishAppInfo * pNull = 0;

   if ( DmOpenDatabaseInfo ( pDB, &dbID, NULL, NULL, &cardNo, NULL ))
   {
      return dmErrInvalidParam;
   }
```

LISTING 19.1 Continued

```
if ( DmDatabaseInfo ( cardNo, dbID,
                      NULL,
                      NULL,
                      NULL,
                      NULL,
                      NULL,
                      NULL,
                      NULL,
                      &appInfoID,
                      NULL,
                      NULL,
                      NULL ))
{
   return dmErrInvalidParam;
}

if ( NULL == appInfoID )
{
   handle = DmNewHandle ( pDB, sizeof (FishAppInfo) );
   if ( NULL == handle )
   {
      return dmErrMemError;
   }

   appInfoID = MemHandleToLocalID ( handle );
   DmSetDatabaseInfo ( cardNo, dbID,
                      NULL,
                      NULL,
                      NULL,
                      NULL,
                      NULL,
                      NULL,
                      NULL,
                      &appInfoID,
                      NULL,
                      NULL,
                      NULL );

   pAppInfo = (FishAppInfo *)MemLocalIDToLockedPtr ( appInfoID,
   ➥cardNo );
```

LISTING 19.1 Continued

```
      DmSet ( pAppInfo, 0, sizeof ( FishAppInfo ), 0 );

      CategoryInitialize ( (AppInfoPtr)pAppInfo, FishTypesAppInfoStr );

      MemPtrUnlock ( pAppInfo );

   }
   return 0;
}
```

Listing 19.2 shows a few (but not many) modifications to the main form-handling code in `cate_mai.cpp`.

LISTING 19.2 The New Main Form Code for Fish List

```
/*
   CATE_MAI.CPP
   Main form handling functions.
   Copyright (c) Bachmann Software and Services, 1999
   Author: Glenn Bachmann
*/

// System headers
#include <Pilot.h>
#include <SysEvtMgr.h>

// Application-specific headers
#include "category.h"
#include "cate_res.h"

static void    MainFormInit (FormPtr formP);
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);

// Local helper functions
void MainListFill (FormPtr pForm);
void MainListDrawFunction (UInt itemNum, RectanglePtr bounds,
➥CharPtr *pUnused);
Boolean AddFish ( void );
Boolean AddFish_EventHandler (EventPtr pEvent);
```

LISTING 19.2 Continued

```
/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;

   switch (eventP->eType)
   {
      case menuEvent:
      {
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormMenuHandler (formP, eventP);
         break;
      }


      case ctlSelectEvent:
      {
         // A control button was tapped and released
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormButtonHandler (formP, eventP);
         break;
      }

      case frmOpenEvent:
      {
         // Main form is opening

         FormPtr formP = FrmGetActiveForm ();

         MainFormInit (formP);

         FrmDrawForm (formP);

         handled = true;
         break;
      }
```

LISTING 19.2 Continued

```
      default:
      {
         break;
      }
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   formP - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr pForm)
{
   char szType [dmCategoryLength];

   // Set the controls to an initial state
   // Here we set the name of the currently selected category
   FishGetCurrentType ( szType );

   Word wID = FrmGetObjectIndex (pForm, MainTypePopTrigger );
   ControlPtr p = (ControlPtr) FrmGetObjectPtr (pForm, wID);

    CategorySetTriggerLabel ( p, szType );

   MainListFill ( pForm );

   // Set the drawing function for the main list
   Word wIDMainList = FrmGetObjectIndex (pForm, MainFishList );
   ListPtr pCtlMainList = (ListPtr) FrmGetObjectPtr (pForm, wIDMainList);
   LstSetDrawFunction ( pCtlMainList, MainListDrawFunction );
}

/*
   MainFormMenuHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
```

LISTING 19.2 Continued

```
            false - not handled
*/
Boolean
MainFormMenuHandler (FormPtr /*formP*/, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.menu.itemID)
   {
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/

Boolean
MainFormButtonHandler (FormPtr pForm, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      case MainNewButton:
      {
         if ( AddFish () )
         {
            MainListFill ( pForm );
         }

         handled = true;
```

LISTING 19.2 Continued

```
         break;
      }

      case MainTypePopTrigger:
      {
         FormPtr pForm;
         UInt wType;

          pForm = FrmGetActiveForm ();

         if ( FishSelectType ( pForm, &wType, MainTypePopTrigger,
         ➥MainTypeList ) )
         {
              MainListFill ( pForm );
         }
         handled = true;
         break;
      }
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainListFill:
   Fills the fish list based upon the currently selected category
   Parms: pForm - pointer to main form
   Returns: none
*/
void
MainListFill (FormPtr pForm)
{
   // Get a ptr to the fish list
   Word wIDFishList = FrmGetObjectIndex (pForm, MainFishList );
   ListPtr pCtlFishList = (ListPtr) FrmGetObjectPtr (pForm, wIDFishList);

   // Get the count of available main list items from the selected aisle
   int iFishCount = FishGetCount ();
```

LISTING 19.2 Continued

```
   // Fill the main list based on the combo selection
   LstSetListChoices ( pCtlFishList, NULL, iFishCount );

   // Redraw the list
   LstDrawList (pCtlFishList);

   return;
}

/*
   MainListDrawFunction:
   Draws a single item in the main list
   Parms: itemNum = 0 based index of item to draw
          bound = ptr to rectangle providing drawing bounds
          pUnused = ptr to char string for item to draw. In this app, we
          never give the actual item data to the list, so we ignore
          this parm
   Returns: none
*/
void
MainListDrawFunction (UInt itemNum, RectanglePtr bounds,
➥CharPtr *pUnused)
{
   FormPtr pForm = FrmGetActiveForm ();

    CharPtr        pText;
   Handle      hFish = NULL;

   // Find the record in the category
   FishGetFish ( itemNum );

   pText = FishGetName ();

    if ( pText )
   {
       WinDrawChars ( pText, StrLen (pText), bounds->topLeft.x,
       ➥bounds->topLeft.y);
   }

   FishReleaseFish ();
   return;
}
```

LISTING 19.2 Continued

```
/*
   AddFish:
   Parms:   none
   Return:  none
*/
Boolean AddFish ( void )
{
   // Initialize the form
   FormPtr  pForm = FrmInitForm (AddFishForm);

    if ( FishNew () )
   {
      // Edit it!
      VoidHand hRecord = FishGetFish ( 0 );
      VoidHand hText = MemHandleNew ( FISH_LENGTH );
      CharPtr pText = (CharPtr)MemHandleLock (hText);

      StrCopy ( pText, "<new fish>" );

       MemPtrUnlock ( pText );


      Word wIDField = FrmGetObjectIndex ( pForm, AddFishNameField );
      FieldPtr pField = (FieldPtr) FrmGetObjectPtr ( pForm, wIDField);

      // Copy the text into the edit field
        FldSetTextHandle ( pField, (Handle)hText);

      // Set the current trigger label
      char szType [dmCategoryLength];
      FishGetCurrentType ( szType );
      Word wID = FrmGetObjectIndex (pForm, AddFishTypePopTrigger );
      ControlPtr p = (ControlPtr) FrmGetObjectPtr (pForm, wID);
       CategorySetTriggerLabel ( p, szType );

       // We set an event handler for the form
       FrmSetEventHandler (pForm, AddFish_EventHandler );

      // Display the dialog
      if ( FrmDoDialog (pForm) )
      {
         hText = FldGetTextHandle (pField);
```

LISTING 19.2 Continued

```
         CharPtr pText = (CharPtr)MemHandleLock (hText);

         FishSetName ( pText );

          MemPtrUnlock ( pText );
      }

      FishReleaseFish ();

      // Destroy the form
      FrmDeleteForm (pForm);

      return true;
   }
   return false;
}


/*
   AddFish_EventHandler:
    Handles form events for form
   Parms:   pEvent = ptr to event handler
   Return:  TRUE if we handled the event
*/

Boolean AddFish_EventHandler (EventPtr pEvent)
{
   Boolean  bHandled = false;
   static UInt s_wNewType;

   switch (pEvent->eType)
   {
      case ctlSelectEvent:
        {
          // A control button was pressed and released.
           switch (pEvent->data.ctlSelect.controlID)
            {
               case AddFishOKButton:
            {
               // Save the fish!
               FishUpdate ();
```

LISTING 19.2 Continued

```
                   // Remove the details form and display the edit form.
                   bHandled = false;
                   break;
            }

               case AddFishCancelButton:
            {
                  // The cancel button was pressed, just go back to
                  // the edit form.
                   bHandled = false;
                   break;
            }

               case AddFishTypePopTrigger:
            {
               // User wants to pick a fish type
               FormPtr pForm = FrmGetActiveForm ();
               FishSelectType ( pForm, &s_wNewType,
               ➥AddFishTypePopTrigger, AddFishTypeList );
               bHandled = true;
               break;
            }
            }
        }
    }
   return bHandled;
}
```

Summary

Many developers have found that adding category support to their applications was more time-consuming and error-prone than adding database support in the first place! I hope this chapter will save you time in developing your own application's category support. Depending on your application and your users, the inclusion of good, standard category support can make quite a difference in the usability of your program.

20

Expansion Memory Cards and the Virtual File System

IN THIS CHAPTER

- A Brief History of Memory Storage on Palm-Powered Handhelds
- A Proliferation of Standards
- Palm's Initial Rollout
- Palm OS 4.0 and VFS, the Virtual File System
- A Brief History of Expansion Support in Palm OS
- Enter VFS, the Virtual File System
- VFS and Developers

Just as a CD-ROM drive on desktop computers has gone from being an expensive add-on to a required component, expansion memory card slots are now a pervasive feature on just about all Palm OS-based PDAs, including models from Palm, Sony, Handspring, and Handera. Yet this feature remains one of the most confusing and widely misunderstood aspects of mobile computing today. The reasons for this confusion are many, and after considering the history of expansion memory cards on Palm-powered devices, it is not surprising that the PDA-buying public is largely unaware of the benefits of the tiny slot at the front of their new handheld device.

The first half of this chapter attempts to clear away the confusion by:

- Reviewing the history of memory storage and expansion memory on the Palm OS platform.
- Explaining the reasons why expansion memory is an attractive feature.
- Enumerating the various devices and standards offered by PDA manufacturers.
- Providing a close examination of the Palm OS Virtual File System (VFS), on which most of the expansion memory support available today is based.

The second half of the chapter introduces the reader to a developers' view of Palm's expansion memory support.

Along the way, we take a tour of the functions available for adding expansion memory support to an application, and concludes with an actual example showing how to use those functions in an application.

A Brief History of Memory Storage on Palm-Powered Handhelds

There is an age-old adage in computing that says "data will always expand to fit available memory." Indeed, it is hard to imagine how we ever used personal computers that came with a 10 megabyte hard drive, but once upon a time, that was considered more storage than anyone could ever dream of using. In fact the lowliest PC available today comes equipped with a thousand times more storage than those early computers, and in the brave new world of MP3 digital music, digital photos, and full motion video there seems to be no end in sight for the growth of storage capacity. Given more storage, we inevitably find more ways to make use of that capacity; our voracious data appetite never seems to be quelled.

Predictably, storage capacity on handheld computers has also followed a similar growth path. Earlier devices ran on 512KB of memory or less, but soon enough there appeared models that offered a standard 1MB, 2MB, 4MB or 8MB. As I write this, many of the newer models are coming standard with 16MB of memory storage. Is the day far off when we will go to the store and buy a PDA that comes with a gigabyte of storage? If history tells us anything, the answer is obvious. Our increasing consumption of memory will dictate that we will ultimately require larger and larger storage capacities.

So with PDAs seemingly following in the footsteps of the personal computer with an ever-increasing amount of data storage, you might wonder why you need expansion memory cards. For one, without an expansion slot your PDA would not be upgradeable, meaning that if your memory storage needs grew to be more than the available space on your PDA, you would need to go out and purchase a whole new device with more memory. Aside from adding what amounts to a "second hard drive" to your PDA, expansion cards offer other interesting uses, such as the ability to share information (such as photos) with other types of electronic devices, including cameras, music and video players, and printers. Also, expansion cards are perfect for storing large files that you use less frequently than your main applications, for example a large dictionary, travel maps, or a medical database.

Given these interesting applications, it is not surprising that handheld manufacturers developed new devices that offer expansion memory to their customers. Handspring was the earliest with their Visor line, offering a proprietary expansion slot called "Springboard" as a standard component for all of their devices. Beyond expansion memory cards, the Springboard slot has become famous for supporting miniature cameras, barcode scanners, and modems. TRG (now Handera) offered an expandable model called the TRGPro, which was unique in that it embraced two industry standards, Compact Flash (CF) and Secure Digital (SD). Sony soon followed with their first Palm OS-based PDA, the Clie (pronounced "*CLEE-ay*"), which came standard with a slot that supported their Memory Stick technology, making their PDA compatible with Sony's other consumer electronics products that (naturally) supported Memory Stick.

By the year 2001, Palm was in fact the only major device manufacturer that did not offer a built-in expansion memory card slot on their devices. But they soon rectified this situation with the introduction of the m500 and m505 (followed by the m125), which came standard with a single slot for handling the SD and MMC standard. It now appears that most (if not all) of Palm's line of products will embrace expansion card support, as will most of the devices offered by Palm's OS licensees. At the same time, card manufacturers are offering cards with capacities ranging from 2MB all the way up to 512MB, and it won't be long before we see a card that holds more than 1GB of data. What's more, the price per megabyte is also dropping, making these cards more affordable than ever.

A Proliferation of Standards

Perhaps one of the most confusing aspects of expansion memory storage on PDAs is the dizzying array of supported formats. As opposed to the original floppy disk on the personal computer, which quickly became a de facto standard that all PC makers rallied around, it seems that handheld manufacturers cannot agree on single standard format for their collective customers. Compact Flash, SD/MMC, Springboard, and Memory Stick all represent different technologies and form factors, and they are incompatible, which means that you cannot insert a card from one format into a slot designed for another.

Although it would seem obvious that these vendors should, for their customers' sakes, get together and decide on a single standard, for many reasons this is not likely to happen. For one, each of the device manufacturers has a vested interest in differentiating their devices from those offered by a competitor. An example of this is Handspring's Springboard. Although one could argue there are technical reasons why existing standards were unsuitable for Handspring's requirements, it is also clear that Handspring sought to establish a unique brand for their Visor line, and the Springboard is one of the important hallmarks that ties together the Handspring device lineup.

Another reason for the diversity of formats goes back to the history of removable media. Before PDAs got into the game, one of the primary uses of expansion cards was in digital cameras. And in the digital camera world, the same situation exists, with CF, SD, MMC, and Memory Stick all being used in different models. For a company like Sony, with an investment in Memory Stick technology across its entire product line, it would be natural to expect that they would value compatibility across all Sony consumer electronics devices over compatibility with other manufacturers' PDAs.

Finally, the device manufacturer has to keep its target audience in mind. For some types of applications, certain media formats were already prevalent. Handera's CF and SD slots are in part a tribute to the established presence these formats already enjoy in some segments of their enterprise customers.

Table 20.1 attempts to lay out the various device manufacturers and their supported expansion memory formats.

TABLE 20.1 Name of Table

PDA Vendor	Model(s)	Formats Supported	Pros	Cons
Palm	m500, m505, m125, i705	SD/MMC	Embraced open standards, scalable capacities.	Poor introduction, lack of bundled software to highlight solution.
Handspring	Visor	Springboard	Numerous third-party modules on market.	Proprietary standard not supported by other device manufacturers Limited capacities. Bulky. Does not support Palm OS VFS interface.
Sony	Clie	Memory Stick	Large capacities, compatibility with other Sony products. Bundled software for mp3, photo/ video.	Not supported by other device manufacturers.
TRG/Handera	Pro, 330	CF, SD	Embraced open standards, scalable capacities.	Relatively few customers, lack of bundled software to highlight expansion card benefits.

Palm's Initial Rollout

As you can see, Palm was the last of the major device manufacturers to offer a handheld with an expansion card slot, but owing to the huge popularity of Palm's devices, as well as their position as the "standard-bearer" for the Palm OS and its licensees, Palm's implementation of this technology has a large impact on customers and the industry overall.

Unfortunately, the initial rollout of Palm's expansion slot-enabled devices may not have communicated the value of the memory slot as well as it might have. This was in part responsible for the current mystery and confusion surrounding how expansion slots work on PDAs. The first units, the m500 and m505, did indeed ship with a single SD/MMC slot. However, the device packaging told the customer almost nothing about the slot, offering no explanation of what it could be used for or indeed how to use it. The device did not come with a blank card, nor was it clear where to obtain one. Although expansion-aware applications were made available on a companion CD, many customers never installed the CD, and none of the built-in Palm software applications (Address Book or MemoPad, for example) had been modified to make use of expansion media. Palm's choice of how to give the user access to the expansion slot was also obscure, requiring the user to choose a category named Card in order to see any applications stored on the card.

Third-party software developers were also slow to enhance their applications to support Palm's expansion technology, leaving puzzled customers to figure it out for themselves. A slew of enhanced expansion-aware launchers, shells, and file managers did appear on the scene to try to bring order to the chaos, but the damage was done. A relatively small but enthusiastic community of adventurous early adopters took the plunge and made the effort to put the pieces together and spread the word on how to make use of the new slot.

After a slow start, we are finally starting to see a critical mass of expansion memory-aware third-party applications, but there is still a ways to go before one can say that expansion card support is universal in the sense that floppy drive support was universal on desktop PCs. It is still a relatively obscure operation for an individual to migrate files and applications from their PC to their PDA's memory card, and the problem grows exponentially if you are an IT manager responsible for the rollout of PDAs to hundreds or thousands of mobile employees. Better tools are appearing to help manage these problems, but at present it requires some research to find third-party solutions that bridge the gap.

Palm OS 4.0 and VFS, the Virtual File System

Coinciding with the availability of Palm's m-series devices was the introduction of a new version of the Palm operating system, version 4.0. Palm OS 4.0 was the first OS release from Palm to specifically include native expansion file support through the introduction of the "Virtual File System," or VFS. Earlier implementations of VFS had been included on pre-OS 4.0 devices from Palm OS licensees Sony and Handera, and in fact those implementations are for the most part binary compatible with the 4.0 VFS subsystem.

Among other things, one of the more significant enhancements was the arrival of expansion card support built in to the operating system, with documentation for third-party developers on how to make use of the new cards. For non-programmers, the only visible signs that things had changed were the addition of a new Card category in the standard application launcher, as well as a new menu command for copying a file to or from a card. Palm referred to the new built-in OS support as the "Virtual File System" or VFS for short.

One of the biggest areas of confusion for customers and developers alike is the assumption that an expansion card is treated like a "second hard drive" on your PDA. In the desktop world, you can add more hard drive storage to your PC in about a half an hour with a screwdriver, and after you are done you can work fairly transparently with your new drive. You can run applications off of the second drive just as easily as you can from the original, and you can even store your application data on a different drive. In general, it doesn't matter; it all just works.

With VFS, Palm significantly changed the way that applications load and store their data, creating a situation where applications need to work with files one way if the files were on the original PDA's memory (which I will refer to as RAM), and work with files another way if the files were on a VFS-mounted expansion card. This really slowed down the rollout of new VFS-aware versions of Palm applications. Further, Palm's VFS implementation did not allow users to run applications directly from card storage. Palm's launcher did allow you to store an application (such as a game or word processor) on the card, but when you tapped on the application icon, Palm OS would work behind the scenes to temporarily copy the application to RAM first, and then run it. Depending on the size of the application, a noticeable but not unreasonable delay would occur while the application was being copied over. And Palm's built-in "core four" (Address Book, Datebook, ToDo, and MemoPad) remained blissfully unaware of the existence of VFS, and in fact could not be moved to card storage or store their data on a card.

A problem with this scenario is that even if you store a large application on a VFS-mounted card, that application will not be able to recognize and use its own databases if you choose to store them on the card as well. For example, if you take a VFS-unaware dictionary application and put it as well as the dictionary database on an expansion card, when the dictionary application is run it will naturally be copied first to RAM, but then will be unable to find its dictionary database. To help solve this problem, Palm introduced the concept of *bundled components*, whereby a developer could tag the various databases and other components of their application, so that Palm OS would be obligated to move all of the application components from the card to RAM upon launching. Still, not every application vendor took this step, and as a result the users must determine on a case-by-case basis whether their applications can be run from an expansion card.

These issues stem from a design decision Palm made when it contemplated how to support expansion cards. One can sympathize somewhat with their dilemma: the core operating system's file system is very minimal, and thus would not scale well to support removable media that is expected to grow to very large sizes. Also, one gets the impression that Palm imagined most of the utility of expansion media being in terms of reference materials, e-books, and other read-only data. In my experience, this is counter to customers' expectations, which usually assume they are expanding their 8MB device to become a 72MB device with the addition of a 64MB card.

Expectations notwithstanding, overall VFS is a good addition to the Palm OS. Even though it requires some commitment on the part of the developer community in order to fully utilize it, the system is built to handle a variety of uses, and appears to scale well to large denominations of expansion cards. To their credit, Palm has also been very successful in getting their OS licensees to migrate to the 4.0 version, so applications that are VFS-aware in general will run on any modern PDA. (A notable exception to this is Handspring. Visors still run OS 3.5, and thus are not VFS-aware, although third-party vendors are working to resolve this with add-on utilities.)

A Brief History of Expansion Support in Palm OS

Prior to the release of VFS in Palm OS 4.0, Handspring was the only Palm OS licensee to include operating system extensions supporting the use of expanded memory, through the Springboard interface that came standard with the Visor line of devices. These extensions were proprietary extensions of Palm OS 3.x, allowing Visors to accept and make use of a wide variety of third-party adapters conforming to the Springboard specification, including cameras, modems, networking cards and other peripherals. Among the Springboard adapters were a series of "flash modules," which allowed Visor customers to augment their device's internal memory with add-on cards containing 8MB or more of memory.

The Visor OS extensions enabled Palm OS applications to work transparently with expansion memory. The Handspring Application Launcher also provided users with easy access to applications on an expansion card. Applications found on an expansion card were listed among internally stored applications, with a simple dot displayed to the left of the application icon indicating that an application is located on the card. Handspring also supplied with its own memory cards a utility called the File Mover, which gave users the ability to copy or move files to and from a card.

This was possible because Handspring's software architects mapped the expansion memory into the standard Palm database system, such that an application using the standard Palm OS Database Manager APIs could refer to an expansion memory card as "Card 1" (Card 0 being mapped to the device's built-in memory). Unless the application developers hard-coded their Database Manager calls to assume Card 0, in theory the application would "automatically" be able to work with the expansion card. This means it could open and close databases, as well as read and write records from and to those databases.

Handspring however did not augment the Palm SDK with APIs to enable third-party developers to copy, move, or delete databases, nor did it provide for a way to store non-Palm files (such as JPG or MP3) on expansion cards. (A Handspring SDK is available as part of Handspring's developer program, and there are APIs in that SDK such as `HsDatabaseCopy` that might appear to be suited for some of these functions, but in fact these functions produce errors when attempting to copy databases to or from expansion cards.)

Enter VFS, the Virtual File System

Palm took a more expansive approach when architecting its own expansion memory subsystem for OS 4.0. As we will see, this approach was at the expense of simplicity and transparency for Palm applications. The Virtual File System would appear to be designed first and foremost to enable access to expansion memory cards that were already being used on other handhelds and electronic devices such as cameras, audio players, and laptops. This includes SD, MMC, Memory Stick, and Compact Flash, and potentially other formats.

Additionally, the OS 4.0 expansion sub-system is capable of supporting multiple file systems through a driver subsystem. Palm chose the familiar FAT file system as the standard system supported by the OS. Although a detailed discussion of the Palm OS file system internal architecture is beyond the scope of this chapter, theoretically a developer could program a VFS-compatible driver that enabled the mounting of drives formatted using NTFS, NFS, or other file systems. Most of these file systems (FAT included) support hierarchical directories, a concept previously foreign to Palm OS. Finally, Palm's system allows you to mount multiple volumes simultaneously on a single device.

From the users' point of view, Palm's new expansion card support appears to be relatively transparent, although not quite as transparent as Handspring's approach. Admittedly disconcerting at first, the built-in Application Launcher represents an expansion card as a "category," listed at the end of all of the normal application categories. To view applications on the card, the users must select the Card category just as if they were selecting the Business category. From the Card category, applications can be "launched" from the card. Applications located on the card thus cannot be associated with categories. Palm also chose not to bundle a file management utility with their expansion-capable devices. Instead, they made the built-in Launcher "expansion-aware" through a new Copy menu item, and some limited capabilities to access files on a card through the Beam, Delete, and Info menu items as in Figure 20.1.

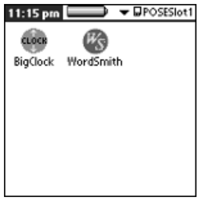


FIGURE 20.1 Launcher in Card category.

VFS and Developers

For developers, things are a bit more complex. Yes, non-VFS aware applications can run from an expansion card. Native Palm files (PDB, PRC, and PQA) can be stored on a card as well. But that's where it stops. None of the Database Manager APIs in the Palm SDK can be used to access databases residing on the card. Developers also soon realized that applications could not be "run in place" from the card—programs literally had to be copied from the card into internal memory prior to being launched.

Even a cursory review of Palm's developer documentation on VFS and expansion makes it obvious that modifying your application to be VFS-aware involves a decent amount of work, and requires you to write separate code specifically to access and work with databases residing on expansion memory cards. Although this revelation caused groans of despair from many in the developer community, in reality it is not too bad. Part of the goal of this chapter is to make it as easy as possible to integrate at least basic VFS compatibility into your programs. Once you are over the initial hump, you will most likely find that many new possibilities will open up for you.

Volumes, Slots, and Cards

Some of the terms you will see in conjunction with expansion memory on Palm devices can be confusing, so it's worth getting a few definitions out of the way:

- A *slot* refers to a physical expansion slot on a device. It is important to note that a slot has no relationship to the concept of a "card number" used in the Database Manager. A slot might not have a card inserted at any given time.
- A *card* refers to the storage media itself; an SD card, an MMC card, or a Memory Stick card. Only one card can be in one slot at any one time.
- A *volume* is a named, formatted area of a physical card that can hold information, typically organized into files and directories. Although clearly some file systems allow multiple volumes to be described on a single storage media, at present Palm OS recognizes only one volume mounted per slot.

- A *file system* refers to a scheme for storing files on media, and typically supports basic file operations such as open, close, read, and write. Palm OS natively supports both FAT and VFAT via system components. The details of the underlying file system are not normally a concern of users or developers.
- *VFS*, the Virtual File System, refers to the software and programming interfaces that allow programs to access and work with cards, volumes, and file systems without needing to understand the type of media or the file system mounted on that media.

Cards and Directories

One of the most misunderstood parts of expansion card support among developers and users alike is the role of directories on those cards. As you will see later in this chapter, it is fairly trivial to write an application that can present a view of a card as a set of files stored in hierarchical directories, similar to how Windows, Macintosh, and other computer platforms are organized. Further, expansion cards already had a history of use with other electronic devices, including cameras, audio players, and card readers.

Often, such devices came bundled with computer software that allowed them to be attached to your PC and mount them as "drives," which could then be browsed from within your computer's normal filing system. Just as with hard drives, upon inspection it was easy to see that the cards had certain pre-defined subdirectories where photos, audio, and other files seemed to be stored. Many users comfortable with this mode of copying and moving files between their PCs and expansion cards even created their own directories on these cards.

But for a number of reasons, it is important for users and developers to recognize that expansion cards should not be treated like blank floppy disks or hard drives. When an expansion card is formatted, depending on the operating system doing the formatting, the card is given a standard set of named directories, and it is expected that applications respect and adhere to these standard directories.

The reason is due to both simplicity and efficiency. Palm understood that it would not be efficient to make its own Launcher look everywhere on an expansion card for applications to list on the screen, nor would it be efficient for third-party applications to have to search for file types such as documents or databases if they were strewn throughout a series of directories on a given card. It would be much more efficient if Palm's launcher would only need to look in one place for programs, a document reader would only need to look in one place for documents, and so on.

Just as important, Palm did not want to require users to "browse" the directory structure in order to locate their programs and files. Although on other platforms we have grown used to doing this, mobile devices are different. Users need to find files in a hurry and have no patience for sifting through an expansion card's contents in search of a lost program or file.

Going with a standard set of directories that are tied to very specific uses simplifies tasks such as running programs, looking for files, and moving files from device to card and vice versa. Thus cards that are formatted or mounted by Palm devices are equipped by default with the standard directory structure shown in Table 20.2.

TABLE 20.2 Default Standard Directory Structure

Directory	Description
`/PALM`	The parent directory for Palm files, other programs, and data
`/PALM/Backup`	A sub-directory reserved for use by Palm for the purposes of backing up files from the device
`/PALM/Programs`	The parent directory for programs and their associated data files
`/PALM/Launcher`	A directory reserved for storing program files that are visible in Palm's Application Launcher screen

The `/PALM/Programs` directory is typically where third-party application data is stored. Most applications adhere to creating a subdirectory with the company or program name off of `/PALM/Programs`, and then storing any card-resident data under there. Thus if I wrote a program called `MyPhotos`, my program would take care to create a directory called `/PALM/Programs/MyPhotos`, and place its files (in this case digital images) in that directory.

When a user asks the Palm Application Launcher to copy a program to the card (through the Copy menu), Palm will in fact copy the program's PRC file to the `/PALM/Launcher` directory.

In addition to these standard directories, Palm OS pre-registers directories that should be used to store common non-Palm file types such as GIF, JPG, WAV, as well as a number of MIME types. Developers are free to ignore these pre-registered directories, but in doing so they might create problems for users who insert cards in other electronic devices that expect images and other files to be located in these standard places.

Predictably, a number of utility applications now allow the handheld user to browse the card's physical directory structure, and even create directories, rename directories, and move files among directories. Although it is educational to view your card in this way, beware that you can get yourself into a bit of trouble if you dive in and rework the way files are stored on your card, similar to what might happen if you decided to open your `C:\Windows` directory and start moving programs and other files around in there. Finding lost files is not easy on a tiny screen, and it also too easy to accidentally delete a required Palm or third-party program component or database, possibly damaging your card. Unless driven by a strong need to do otherwise, it's best to adhere to the Palm guidelines.

The VFS and Expansion Manager APIs

There are, as of this writing, some 50 functions associated with the Expansion Manager and Virtual File System. For most situations, however, you'll use a smaller subset of these functions. The following sections, therefore, discuss the smaller subset (the SDK documentation, of course, contains descriptions for all 50 functions).

Let's start out by imagining we want to make an application VFS-aware. By this I mean it should be capable of storing and accessing native Palm databases residing on an expansion card. This gives users the freedom to move their program data to the card, either for use with other computing devices, or simply to access more storage space.

In order to enable this functionality, as with any other operating system platform, you need to enumerate files on a storage device, open and close files, and read and write data from and to files. You also need to be able to perform basic card functions such as verifying that a card is inserted, that a card is formatted, and the like.

The Expansion APIs are largely concerned with the world of slots and cards, not files and volumes. Thus, unless you are developing a slot driver or need a finer level of control over a specific slot, these APIs are not relevant to most developers. There are two utility APIs that are of interest, however:

- `ExpCardPresent ()` can tell you whether there is a card inserted in a given slot, but it requires that you have a "Slot Reference Number." You can obtain such a value by enumerating all slots on the device with `ExpSlotEnumerate ()`, but given today's single-slot devices (the Handera 330 being the notable exception), this seems like overkill. Because most developers will not care so much about the presence of a card in a slot versus the actual presence of a volume on a card in a slot, it is actually more appropriate and simpler to use the VFS volume APIs to obtain this information.
- `ExpCardInfo ()` can tell you about the characteristics of a card that is inserted in a slot. It also requires a valid Slot Reference Number, and in return can tell you the type of card, a manufacturer description, and capability flags such as whether it supports reading and writing of data. For the purposes of this section we will assume that cards we work with support reading and writing.

The VFS APIs are all about files, directories, and volumes, and thus will look fairly familiar to anyone who has used similar APIs on Windows, DOS, or Unix. Most of the expected open, close, read, write, and enumerate functions are here. Additionally, to ease the migration of code from Data Manager, Palm included a set of "helper" APIs that make it possible to perform a subset of Data Manager functionality on a native Palm database that happens to reside on an expansion card. The rest of this chapter will focus on the VFS APIs.

Enumerating Files and Folders

In order to enumerate files and folders on an expansion card, the Palm SDK provides the workhorse function `VFSDirEntryEnumerate ()`, which can enumerate all files in a given directory. But in order to call `VFSDirEntryEnumerate` on a directory, first you need something called a `FileRef` for the directory. `FileRefs` are obtained by opening a file, using the `VFSFileOpen` API.

`VFSFileOpen` is fairly straightforward to use, except for one thing—it requires something called a `VolRefNum`. How do you get a `VolRefNum`? You need to enumerate volumes first with `VFSVolumeEnumerate`! But wait, didn't all this start with enumerating? Have no fear—if this seems a bit complex right now, the next few paragraphs should clear it all up for you. We've got the right set of functions now; we just have to use them in the right sequence.

You can't do anything with VFS unless you first have what is called a Volume Reference Number, or `VolRefNum`. A `VolRefNum` is an identifier that refers to a single, mounted volume, and it is a required value in order to do just about anything else with the VFS functions. Although enumerating volumes may seem silly given virtually all Palm OS devices have a single slot and Palm currently limits slots to just one volume, it is the only way to obtain a `VolRefNum`. So enumerate we must, and `VFSVolumeEnumerate` is just the function we need.

The following snippet of code shows how to enumerate the volumes on a given device:

```
Err             err;
UInt32      volIterator = vfsIteratorStart;
UInt16      volRefNum = vfsInvalidVolRef;



while (volIterator != vfsIteratorStop)
{
   err = VFSVolumeEnumerate(&volRefNum, &volIterator);
   if (err == errNone)
   {
      // found a volume! The variable volRefNum now contains a valid VolRefNum
      // do something with the VolRefNum
   }
}
```

The basic mechanism here is similar to what you will see later when enumerating files. `VFSVolumeEnumerate` takes a pointer to a volume reference number variable, and a pointer to an "iterator." This iterator is a magic value that is pre-seeded with a value of `vfsIteratorStart`. `VFSVolumeEnumerate` is then called repeatedly until it "iterates" through all known volumes. When no more volumes are found, the iterator variable receives a value of `vfsIteratorStop`, at which point you should break out of your enumeration loop.

If your program will need to later access the volume(s), you need to copy and hold in a structure or variable the `VolRefNum` values returned from `VFSVolumeEnumerate` for each volume found.

Note that `VolRefNum` values are not persistent, which means that you should not count on a volume reference number being the same for a given volume each time your program runs. Although on some devices this might turn out to be the case, it is not a requirement that this be so. If you need to uniquely identify a specific volume across multiple sessions of your program, you will need to store a unique volume label, or use some other custom method of identifying the volume, such as a unique signature file.

Now that we have a `VolRefNum`, we can go ahead and access directories and files. The next function we apply is `VFSFileOpen`. `VFSFileOpen` takes a `VolRefNum`, a path name, and an open mode, and in return fills out a "file reference number," or `FileRefNum`.

If you already know exactly what file you want to open (for example, `/PALM/Programs/MyPhotos/Julianne.pdb`), you can pass this path name to `VFSFileOpen`, and if a file exists in that directory by that name, you will receive a valid file reference number in return.

If you want to browse for files in a given directory, you have a bit more work to do. The following code shows how to do it:

```
Char            dirName[256];
UInt32          fileIterator;
FileInfoType    fileInfo;
FileRef         dirRef, fileRef;

// set up the fileinfo structure
fileInfo.nameP = fileName; // point to local buffer
fileInfo.nameBufLen = 256;
```

```
       Char        tempName[256];

// state what directory we want to get a listing of
StrCopy ( dirName, "/PALM/" );

// try to open the directory and get a valid fileref for the directory
err = VFSFileOpen( volRefNum, dirName, vfsModeRead, &dirRef );

if ( err != errNone )
{
    // handle error
    return err;
}

// we now have an open directory
// now let's iterate through files in the open directory
// initialize the file iterator
fileIterator = vfsIteratorStart;

while ( fileIterator != vfsIteratorStop )
{
    //  call the enumeration function for the currently open directory
    err = VFSDirEntryEnumerate( dirRef, &fileIterator, &fileInfo );

   if ( err != errNone)
   {
      // found a file, the name of the file is in fileInfo.nameP,
      // and the attributes are in fileInfo.attributes
      // do something with the file info
   }
}
// iteration done, no more files or directories
// close the current directory
VFSFileClose (dirRef);
```

First, we open the desired directory with a call to `VFSFileOpen ()`. If that call succeeds, you now how a valid File Reference Number for the open directory. Given this reference number, you can then proceed to perform operations using it. In this case, we want to enumerate all files and subdirectories that are contained within the open directory. Just like we did with volumes, we set up and initialize an iterator, and this time we call `VFSDirEntryEnumerate ()`. As the name implies, it enumerates "directory entries," which can be files or other subdirectories.

If `VFSDirEntryEnumerate` finds an entry, it will fill out a FileInfo structure with the name and attributes of the entry. (Note that it was important that we properly initialized the FileInfo structure with a buffer to receive the filename!) You can tell whether you found a directory or a file by examining the attributes. ORing the attributes with `vfsFileAttributeDirectory` will yield true if the entry is a directory.

At this point you are free to go ahead and open a specific file using the file reference number, or you can do something with the filename itself, such as display it in a list. Just be sure not to change a file, create a new file, delete a file, or do anything else that modifies the directory or its contents while you are in the middle of an enumeration, otherwise your iteration loop will get confused. You should break out of the loop and properly close the directory before performing other operations.

Reading and Writing Files

There are in general two kinds of functions in the VFS family: functions that assume a Palm database and functions that do not. If all you need to do is minimally read records from a Palm database, you simply open your file as described using `VFSFileOpen ()`, and then use `VFSFileDBGetRecord ()` to obtain each record's data handle. This handle can then be used with Data Manager calls. Two other functions may be of use to you for this type of mode: `VFSFileDBGetInfo` gets Data Manager-like attributes from a database, whereas `VFSFileDBGetResource` obtains a resource data handle of a requested type from the database.

The support for native Palm databases on expansion cards is quite limited. If you need to perform more than the minimum functions on a Palm database, such as creating new databases or records, changing records, or sorting, the VFS functions will not be of help; they assume a simple read-only scenario.

In working with native Palm databases, you have two options—you can copy your database to RAM temporarily, modify it, and then copy it back, or you can treat your database as a non-native, non-PDB file and use the remaining VFS APIs to read and write your file as pure bytes rather than records. The second option offers more flexibility, and for larger databases probably a better user experience, but requires much more work. You will need to decide for yourself whether the flexibility required to store and work with your database as a non-PDB file is worth the effort of having to write your own record-access and file-management routines.

Note that if your program needs to deal with non-native Palm files in the first place (for example, perhaps you've written a Palm version of the Windows NotePad utility), there is no reason to not use the regular VFSFile APIs, because your files' structure will likely not be in record format anyway.

Copying Files

The VFS API enables you to copy native Palm files from internal memory to an expansion card volume, and vice versa. This is accomplished by using `VFSExportDatabaseToFileCustom ()` for copying a PDB or PRC file to an expansion card, and `VFSImportDatabaseFromFileCustom ()` for copying a PDB or PRC from an expansion card to internal memory. Given the background we've covered thus far on volume reference numbers and path/filenames, the parameter lists for these functions should be fairly straightforward. The expansion card version of the file is identified by a Volume Reference Number and a path/filename, whereas the internal memory version is represented by the standard database ID and card number (remember that this is a Database Manager card number and has nothing to do with the VFS kind of card!).

The last two parameters in these functions are more interesting: the first is for an application-defined callback function, which can display a progress dialog to the users and allow them to cancel out of the operation. The second is a general-purpose pointer that can pass data to the callback. This would help it do a better job of tracking the progress (for example, you can pass in the name of the file, the size of the file, and so on).

VFSBrowser: A Simple Card Explorer Utility

We can now combine a number of the things we've covered, and create a simple program that uses the VFS APIs. VFSBrowser is an example program that allows you to navigate the sub-directory structure on a card-based volume. It is educational in that it lets you see all of the standard subdirectories provided by Palm, and it also lets you exercise several of the VFS APIs covered in this chapter.

When VFSBrowser runs, if a card is mounted as a volume it presents a list of files and directories that it found on your card in the root directory (`/`). You can tap on a directory name, and VFSBrowser will then drill down one level in the directory tree and display a list of files and sub-directories under the one you tapped. You can go back up one level by tapping the Back button.

As a display mechanism, VFSBrowser uses a slightly modified version of the `Table` control code presented in Chapter 11, "Tables."

VFSBrowser does perform the proper volume enumeration, and even holds an array of volumes it found. However, the user interface only presents the option of browsing the first volume found.

NOTE

Note that the project assumes you have installed and are using Palm OS 4.0 support in your compiler environment.

The source code for the main form and VFS portions of VFSBrowser follows:

```
/*
   TBLS_MAI.CPP
   Main form handling functions.
   Copyright (c) Bachmann Software and Services, 2002
   Author: Glenn Bachmann
*/

// system headers
#include <palmos.h>
#include <SysEvtMgr.h>
#include "expansionmgr.h"
#include "vfsmgr.h"


// application-specific headers
#include "tables.h"
#include "tbls_res.h"

static void    MainFormInit (FormPtr formP);
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);

// local helper functions
void MainListFill (FormPtr pForm);
void MainListDrawFunction (void * pTable, Int16 row, Int16 column, RectanglePtr
bounds);
void MainListCalcTopVisible (FormPtr pForm);

// current record in the table
static Int16 s_wCurrentRecord;
static Int16 s_wTopVisibleRecord;

#define COL_FILE_NAME   0

static UInt16 s_aryVolumes[10];
static UInt16 s_uiNumVolumes = 0;
static char s_szCurrentFolder[256];
static Char * s_aryFileNames[11];

Err EnumVolumes ( void );
Err EnumFolder ( UInt16 volRefNum, Char *pszFolder );
```

```
/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr pEvent)
{
   Boolean  handled = false;

   switch (pEvent->eType)
   {
      case menuEvent:
      {
         FormPtr pForm = FrmGetActiveForm ();
         handled = MainFormMenuHandler (pForm, pEvent);
         break;
      }


      case ctlSelectEvent:
      {
         // A control button was tapped and released
         FormPtr pForm = FrmGetActiveForm ();
         handled = MainFormButtonHandler (pForm, pEvent);
         break;
      }

      case tblSelectEvent:
      {
         // set the current record based on user selection
         FormPtr pForm = FrmGetActiveForm ();
         s_wCurrentRecord = TblGetItemInt ((TableType *)
➥pEvent->data.tblSelect.pTable,
                                       pEvent->data.tblSelect.row,
                                       pEvent->data.tblSelect.column );

       // if it's a folder,
       // switch the current folder and refill the list
       StrCopy (s_szCurrentFolder, s_aryFileNames[pEvent->data.tblSelect.row]);
```

```
  MainListFill (pForm);
  FrmDrawForm ( pForm );

   break;
}

// physical scroll buttons
  case keyDownEvent:
{
 if (pEvent->data.keyDown.chr == pageUpChr)
 {
      FormPtr pForm = FrmGetActiveForm ();
     s_wTopVisibleRecord --;
      MainListFill ( pForm );
      FrmDrawForm ( pForm );
      handled = true;
  }
  else if (pEvent->data.keyDown.chr == pageDownChr)
  {
      FormPtr pForm = FrmGetActiveForm ();
     s_wTopVisibleRecord ++;
      MainListFill ( pForm );
      FrmDrawForm ( pForm );
      handled = true;
   }
   break;
}

case frmOpenEvent:
{
   // main form is opening

   FormPtr pForm = FrmGetActiveForm ();

   MainFormInit (pForm);

   FrmDrawForm (pForm);

   handled = true;
   break;
}
```

```
      default:
      {
         break;
      }
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   pForm - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr pForm)
{
   // set the controls to an initial state
   s_wCurrentRecord = -1;

        // get a list of volumes
        EnumVolumes ();

        for (UInt16 i = 0; i < 11; i++)
        {
                s_aryFileNames[i] = (Char *)MemPtrNew (256);
                StrCopy (s_aryFileNames[i], "");
        }

        if (s_uiNumVolumes > 0)
        {
                StrCopy (s_szCurrentFolder, "/");

        // walk through the list of files in the current directory
                EnumFolder ( s_aryVolumes[0], s_szCurrentFolder);
        }

   // walk through files in current folder and fill list
   MainListFill ( pForm );
}

/*
```

```
   MainFormMenuHandler:
   Handle a command sent to the main form.
   Parms:   pForm    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormMenuHandler (FormPtr /*pForm*/, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.menu.itemID)
   {
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   pForm    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/

Boolean
MainFormButtonHandler (FormPtr pForm, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
          case MainBackButton:
          {
               Char *fNameP;
               if (StrLen ( s_szCurrentFolder ) > 1 )
                {
                 // move pointer to end and move backward looking for first
                 // '/' character
                  fNameP = s_szCurrentFolder + StrLen( s_szCurrentFolder ) - 1;

                  // skip past the trailing /
                  fNameP--;

                  while( *fNameP != '/' )
                      fNameP--;

                  // add a null terminator after that
                  fNameP++;

                  *fNameP = 0;

                  FrmCustomAlert (ErrorAlert, s_szCurrentFolder, NULL, NULL);
                  MainListFill (pForm);
                          FrmDrawForm ( pForm );
               }
               break;
          }
      default:
      {
         handled = false;
         break;
      }
   }
   return handled;
}

/*
   MainListFill:
   Fills the Table
   Parms: pForm - pointer to main form
   Returns: none
*/
void
MainListFill (FormPtr pForm)
{
   // get a ptr to the table
```

```
Int16 wIDTable = FrmGetObjectIndex (pForm, MainFilesTable );
TablePtr pTable = (TablePtr) FrmGetObjectPtr (pForm, wIDTable);

// init the table

// get the number of rows defined in Constructor for the table
// this is the number that will be visible
Int16 wRows = TblGetNumberOfRows (pTable);
Int16 i;

for (i = 0; i < wRows; i++ )
{
   // set the cells to have custom drawing
   TblSetItemStyle  (pTable, i, COL_FILE_NAME, customTableItem );

   // set the row to be unusable to start
   TblSetRowUsable (pTable, i, false);
}

// set the columns to be usable
TblSetColumnUsable  (pTable, COL_FILE_NAME, true );

// set our draw procedure callback for each column
TblSetCustomDrawProcedure (pTable, COL_FILE_NAME, MainListDrawFunction );

// reset the top visible row based on size of table and current record
MainListCalcTopVisible (pForm);

UInt16 uRecordNum = 0;

// load records into the table, starting at top visible
// get the record number of the top visible record
uRecordNum = s_wTopVisibleRecord;

  for (UInt16 i = 0; i < 11; i++)
  {
       s_aryFileNames[i] = (Char *)MemPtrNew (256);
       StrCopy (s_aryFileNames[i], "");
  }

  EnumFolder ( s_aryVolumes[0], s_szCurrentFolder );
```

```
   for (i = 0; i < wRows; i++, uRecordNum++)
   {
      UInt16 uNextRecordNum = 0;

      if (StrCompare (s_aryFileNames[i], "") != 0)
      {
         // remember the associated record number for this row
         TblSetItemInt (pTable, i, COL_FILE_NAME, uRecordNum);
         TblSetRowUsable (pTable, i, true);
      }
      else
      {
         // if we are out of records, mark row as unusable
         TblSetRowUsable (pTable, i, false);
      }
      // force a row redraw
      TblMarkRowInvalid (pTable, i );
   }

   return;
}

/*
   MainListCalcTopVisible:
   Sets the top visible item in the table
   Parms: none
   Returns: none
*/
void
MainListCalcTopVisible (FormPtr pForm)
{
   // get a ptr to the fish table
   Int16 wIDTable = FrmGetObjectIndex (pForm, MainFilesTable );
   TablePtr pTable = (TablePtr) FrmGetObjectPtr (pForm, wIDTable);

   // get the number of rows defined in Constructor for the table
   // this the number that will be visible
   Int16 wRows = TblGetNumberOfRows (pTable);

   // this all sets up what the top visible record
   // should be
   UInt16 uRecordNum = 0;
   if ( s_wCurrentRecord != -1)
   {
      if ( s_wTopVisibleRecord > s_wCurrentRecord )
      {
         // the current record is before the first visible record
         // "scroll up"
         s_wTopVisibleRecord = s_wCurrentRecord;
      }
   }

   // now adjust to make sure that we have a full number of records even
   // if it's the last screen. If there's less than a screenful left,
   // we have to push TopVisible backwards until there is.
   uRecordNum = 0;

   s_wTopVisibleRecord = min (s_wTopVisibleRecord, uRecordNum);
   return;
}

/*
   MainListDrawFunction:
   Draws a single cell in the main table
   Parms: pTable = table ptr
          row = row of item
          column = cell
          bounds = bounding rectangle for cell
   Returns: none
*/
void
MainListDrawFunction (void * pTable, Int16 row, Int16 column,
➥ RectanglePtr bounds)
{
   FormPtr pForm = FrmGetActiveForm ();

     Char *          pText;


     // get the filename
     pText = s_aryFileNames[row];

   // draw it
     if ( pText )
   {
        WinDrawChars ( pText, StrLen (pText), bounds->topLeft.x,
➥bounds->topLeft.y);
   }

   // release it
   FishReleaseFish ();
   return;
}



/*
     Enumerates all volumes on all cards
     Parms: none
     Returns: none
*/

Err
EnumVolumes ( void )
{
     Err             err;
     UInt32          volIterator = vfsIteratorStart;
     UInt16          volRefNum;
     UInt16          uiCardNo;
     UInt16          uiCardsFound = 0;

   for (UInt16 i = 0; i < 10; i++)
   {
        s_aryVolumes[i] = vfsInvalidVolRef;
     }

   volRefNum = vfsInvalidVolRef;

     while (volIterator != vfsIteratorStop && s_uiNumVolumes < 10)
     {
        err = VFSVolumeEnumerate(&volRefNum, &volIterator);
        if (err == errNone)
        {
             s_aryVolumes[s_uiNumVolumes] = volRefNum;
               s_uiNumVolumes++;
        }
     }
   return errNone;
}

/*

*/
Err
EnumFolder ( UInt16 volRefNum, Char *pszFolder )
{
     Err             err;
     Char            dirName[256], fileName[256];
     UInt32          fileIterator;
     FileInfoType    fileInfo;
     FileRef         dirRef, fileRef;

     UInt32          volIterator = vfsIteratorStart;
     if ( volRefNum == vfsInvalidVolRef )
     {
          return vfsErrVolumeBadRef;
     }

     fileInfo.nameP = fileName; // point to local buffer
     fileInfo.nameBufLen = 256;

     Char        tempName[256];

     StrCopy ( dirName, pszFolder );

     err = VFSFileOpen( volRefNum, dirName, vfsModeRead, &dirRef );

     if ( err != errNone )
     {
          Char    errStr[25];
          StrPrintF ( errStr, "Error opening dir" );
          FrmCustomAlert ( ErrorAlert, errStr, NULL, NULL );
          return err;
     }

     fileIterator = vfsIteratorStart;

   UInt16 uiFilePos = 0;
```

```
     while ( fileIterator != vfsIteratorStop )
     {
          err = VFSDirEntryEnumerate( dirRef, &fileIterator, &fileInfo );

          if ( (err != errNone) && (err != expErrEnumerationEmpty) )
          {
              FrmCustomAlert( ErrorAlert, "Error in VFSDirEntryEnumerate!",
➥ NULL, NULL );
              return err;
          }

          if (uiFilePos >= s_wTopVisibleRecord)
          {
             StrCat (s_aryFileNames[uiFilePos - s_wTopVisibleRecord], dirName);
             StrCat (s_aryFileNames[uiFilePos - s_wTopVisibleRecord],
➥fileInfo.nameP);
            if ( (fileInfo.attributes & vfsFileAttrDirectory) && !(
➥fileInfo.attributes & vfsFileAttrVolumeLabel) )
               {
                 StrCat (s_aryFileNames[uiFilePos - s_wTopVisibleRecord], "/");
               }

          }

          uiFilePos++;

     }

     VFSFileClose( dirRef );
     return errNone;

}
```

Testing and Debugging a VFS Application

VFSBrowser can be tested as usual by downloading the PRC file to your Palm device. Another way to test it is by using POSE. Palm has expansion-enabled the Palm emulator through an installable module named HostFS.prc. HostFS allows you to designate one or more folders on your PC's hard drive and mount them as "volumes" in the emulator environment. This is a very convenient way to test and debug VFS-aware programs, although no emulation is perfect—there is no substitute for testing your program on a physical device. HostFS is available at www.palmos.com, and comes with installation instructions. (As of this writing, there are some glitches in `HostFS.prc` during the reset process of the emulator, resulting in an error message that can be safely ignored.)

With a little more work, `VFSBrowser` can be programmed to display some file properties, and perhaps let the users perform some simple operations against a file.

Summary

Through Palm's expansion memory support and the Virtual File System, it is possible to explore an increasingly rich variety of applications for expansion cards and PDAs in general. As a case in point, my current device is a Sony Clie 760C, and it is equipped with a 128MB memory stick. I have about an hour's worth of music in the form of MP3 files on it, which I use when I go for a jog. I have moved 10 or so large third-party applications to the card, freeing up four or five megabytes on my Clie's built-in RAM, and these applications launch just fine from the card. If I wanted to, I could store beautiful color pictures of friends and family on the card, making my PDA almost a virtual wallet. These capabilities were unheard of just a year ago, but now they are all possible, and my PDA has become more indispensable than ever.

Beyond my own personal uses, some of the more exciting possibilities lay in adopting expansion media as an efficient distribution mechanism for company documents, reports, product data sheets, and applications. This scenario brings significant convenience to the mobile workforce, making it cost-effective to distribute large volumes of timely data in a lightweight format—and all without requiring wireless connectivity or expensive laptops. Complete patient medical histories, large reference materials, product catalogs, real estate home listings, telephone books, maps, and other navigation aids—all these provide but a small peek at the possible applications.

In the coming months and years we can look forward to higher and higher storage capacities, to 1GB and beyond, at unprecedented low prices. Creative people in our industry will dream up new applications and tools, and in time, just as with our old friend the PC, it will become hard for people to remember how we ever got by with a PDA that came without one of those curious little expansion slots.

PART IV
Communications

IN THIS PART

21

Serial Communications

IN THIS CHAPTER

- The Serial Port Hardware
- The Serial Communications Software
- Serial Manager Essentials
- PalmTalk: A Palm OS Serial Terminal Application

The Palm has achieved an incredible level of success, proving to be an indispensable personal information management tool as well as a platform for a rapidly growing raft of third-party and vertical market software.

In many application domains, however, to be a truly useful tool, the Palm platform needs to communicate with the “outside world.” Storing and managing information within the device itself is extremely valuable; however, the ability to share that information is a vital component to any platform’s long-term success and market viability. A key component to the Palm OS success today is its ability to communicate and achieve connectivity with other applications, platforms, devices, and networks. Perhaps the most visible expression of that connectivity is the hot-sync facility, which allows the exchange of information between the Palm and a host system.

With a platform that can both manage and transfer information, the users are empowered to change the way they perform their day-to-day tasks and become more productive and efficient. Of course, the other side to this topic is that communications can (and should be!) fun. In addition to the business activities that can be performed with the transmission of data, a new world of personal communications, messaging, gaming, and other applications becomes possible.

I begin the discussion of communications topics with the Serial Manager, Palm’s low-level API, which exposes the Palm device’s built-in serial port. I conclude this chapter with a sample application that will engage a hosting PC in a “terminal” session. To get the most from this chapter, you will need an application such as Windows Terminal or HyperTerminal running on your PC or other development platform.

The Serial Port Hardware

The present-day Palm device has one physical serial port that is similar to the ports found on your PC, although not entirely compatible. The Palm device uses a UART chip (Universal Asynchronous Receiver and Transmitter), which is compliant with the HPSIR/IrDA Physical Communication Protocol. (Page 5 of the tech sheet for Motorola Dragonball processor MC68328 talks about the UART's properties.) This chip differs from your PC's UART in that it does not give you access to all the signals you might be accustomed to, such as ring indicator (RI) and data terminal ready (DTR). Additionally, your PC UART typically has a 16-byte buffer for both the send and receive buffer, whereas the Palm's UART buffers each hold 8 bytes. The Palm supports serial communications at baud rates ranging from 300 to 115,200bps.

There are a number of ways to "connect" your Palm to the outside world and make use of serial communications. The Palm talks via the cradle to the PC or other device. To communicate with a parallel device (such as a laser or line printer), you can purchase (or build, if you are adventurous!) a serial-to-parallel converter cable. If you want to connect the Palm to a modem, your best bet is to pick up a Palm modem cable. This cable is specially wired to make up for some of the "missing" signals I mentioned. Depending on the device you want your Palm to communicate with, you might need a special cable. You can visit Palm's Web site for more information on the signals available from the Palm device itself and the wiring pin-outs of the cradle.

The Serial Communications Software

Many layers of software make up the entire serial communications "stack." Your application's needs and your interest in creating your own "protocols" will determine where your program will interact with the Palm's communications facilities. The lowest layer above the hardware itself is referred to as the Serial Manager. The Serial Manager provides for byte-level input and output and the ability to set port parameters such as baud rate and parity.

Beyond the Serial Manager, there are layers for modem management and a connection management protocol. Additionally, for packet-oriented communications, the Palm OS provides the Serial Link protocol. The Serial Link protocol offers an interface similar to sockets, where multiple "conversations" can occur over a single physical connection. Of course, the partner application (whatever is running on the PC or other connected device) must use the same protocol.

The best way to learn is to do, and you want to get up and running quickly with serial communications. In this chapter, you will build a Palm application to conduct a terminal session with a host PC. Before you can dive into the application, you need to learn some of the core Serial Manager APIs and concepts. I discuss the high points of the Serial Manager, giving you enough ammunition to get started as a Palm OS communications programmer. For the intricate details of the Serial Manager functions, refer to the Serial Manager documentation from Palm Computing as well as review the SerialMgr.h file shipped with the Palm OS SDK.

Serial Manager Essentials

Palm OS exposes its low-level serial port access via the Serial Manager, as I discussed earlier. To gain access to the Serial Manager, you need to load it at runtime. This is done by calling `SysLibFind`.

The Serial Manager is a system library, meaning it is provided by the operating system itself. Once you load the library, you have access to all the functions therein. Each application making use of the serial library needs to call `SysLibFind` to obtain a handle. Each subsequent Serial Manager function call will require this handle.

The following code shows how to use `SysLibFind` to load Serial Manager:

```
Err e;
UInt16 SerialRefNum;

// Load the serial library
e = SysLibFind("Serial Library",&SerialRefNum);

if (e)
{
   FrmCustomAlert(ErrorAlert,"Failed To Load Library!","","");
   return;
}
```

Once the Serial Manager library is loaded, you can do things such as open and close serial ports and change properties such as baud rate and handshaking.

Opening the Port

Before you use the serial port, you have to open it via `SerOpen`. This function takes three parameters, the handle to the Serial Manager library, the port number, and the initial baud rate desired. As mentioned earlier, this library has support for multiple serial ports (sort of). At the moment, the device has but one, so the port value must be `0`. The baud rate should be set to a standard rate in the range of 300–115,200bps. A successful return code for this function will be `0` or `serErrAlreadyOpen`.

It is possible for multiple tasks to have the port open simultaneously. However, if your application expects to have full control of the serial port and the `SerOpen` function returns `serErrAlreadyOpen`, your application needs to call `SerClose` and not perform any further action with the serial port functions:

```
// SerialRefNum is returned from SysLibFind
e = SerOpen(SerialRefNum,0,9600);
if (e)
{
    FrmCustomAlert(ErrorAlert,"Failed To Open Port 0!","","");
    return;
}

// Let's keep track of when the port is open
bConnected = true;
```

Closing the Port

Another key function, `SerClose` shuts down the Palm's serial port. This function should be called only if a previous call to `SerOpen` returned `0` or `serErrAlreadyOpen`. Please note that there is no “port” parameter to this function. All you pass is the handle to the library returned by `SysLibFind`. This function will shut all ports (one at the moment) opened by your application. It is anticipated that as the hardware progresses to include multiple serial ports, the Serial Manager API functions will be modified to let you select the desired port.

It is important to note that you should call `SerClose` prior to exiting your Palm OS application. This can present a challenge in Palm OS because of the way tasks are closed or terminated with limited warning. To close the port, you might want to practice using a variable to indicate the port's status. If you are about to switch applications, as notified by the application's event loop, check that variable. If your port is open, make a call to close it prior to exiting. Failing to do so can cause problems with other applications that need to use the UART, including your own application the next time it is invoked, the IR port, and hot-sync. (You have been warned!)

```
// Close port
if (0 == Connected) return;
e = SerClose(SerialRefNum);
if (e)
{
 FrmCustomAlert(ErrorAlert,"Error during Close!","","");
}
// Keep track of our state
Connected = 0;
```

`SerGetSettings` and `SerSetSettings`

The `SerGetSettings` and `SerSetSettings` functions allow you to modify the serial port's baud rate, handshaking, and other communications properties. It is a good practice to retrieve the serial port's settings prior to changing them. You want to make a call to `SerGetSettings` first, passing in the handle to the Serial Manager library and the address of a variable of type `SerSettingsType`. Once you have this, you can make the appropriate modifications and then call `SerSetSettings` to make the changes take effect:

```
SerSettingsType sstSetup;
SerGetSettings( SerialRefNum, &sstSetup );
// Let's set the baud rate & other serial comm's properties
sstSetup.baudRate = 9600;
sstSetup.flags =    serSettingsFlagBitsPerChar7 |
                    serSettingsFlagParityOnM |
                    serSettingsFlagParityEvenM |
                    serSettingsFlagStopBitsM |
                    serSettingsFlagStopBits1;
sstSetup.ctsTimeout = serSettingsFlagRTSAutoM
 | serSettingsFlagCTSAutoM;

// Ask Serial Manager to update the port settings
SerSetSettings( SerialRefNum, &sstSetup );
```

SerReceiveFlush

The `SerReceiveFlush` function acts to "reset" the serial port. It discards data from Serial Manager's receive queue and clears the saved error status. It takes two parameters, the Serial Manager's library handle and an "interbyte timeout" (in system ticks). What the interbyte timeout means is simply that `SerReceiveFlush` will block until a timeout occurs while waiting on the next byte:

```
// Let's clear the port, just in case some garbage is
// sitting there
SerReceiveFlush (SerialRefNum, 100 );
```

SerReceiveCheck

The `SerReceiveCheck` function will tell the application the number of bytes waiting in the receive queue. This is an effective way of performing nonblocking I/O. The function takes the handle to the Serial Manager library and the address of an unsigned long (`UlongPtr`). Examine the value of this unsigned long argument to determine whether your application needs to service the receive queue:

```
// See if there is anything in the queue ... "peek"
UInt32 ulBytes;
```

```
e = SerReceiveCheck(SerialRefNum,&ulBytes);
if (ulBytes)
{
   // Read from queue
}
```

SerReceive

The `SerReceive` function will read from the receive queue. The parameters are the library handle, the buffer for receiving data, the count of bytes desired, the interbyte timeout in system ticks, and the address of an `Err` variable. This function will return the number of bytes read. Examine the value of the error variable; it should be zero for a successful read. This value might be `serErrTimeOut`, which indicates that the `SerReceive` function is returning due to a timeout during the read process. The application should check the number of bytes actually read:

```
SerReceive(SerialRefNum,&szBuf[index],1,0,&e);
// Check for read error
if (e)
{
   FrmCustomAlert(ErrorAlert, "Error Reading Serial Port!","","");
   SerReceiveFlush(SerialRefNum,100);
   index = 0;
   return;
}
```

SerReceiveWait

`SerReceiveWait` will wait a given number of system ticks for the receive queue to accumulate a certain amount of data. For example, you use this function if you need to read in records of 20 bytes each and your application could not handle partial records. To call this function, the application must pass the Serial Manager library handle, the number of bytes to accumulate, and the number of system ticks to wait:

```
// Wait for 20 bytes to accumulate
e = SerReceiveWait(SerialRefNum, 20, 200);
if (e)
{
   // Process timeout or line error
}
else
{
   // Read in the bytes
}
```

SerSend

To send data via the serial port, use the `SerSend` function. This function takes the Serial Manager library handle, a pointer to the memory buffer holding the data you want to send, the count of bytes to send, and the address of an `Err` variable. The function returns the count of bytes actually transferred:

```
// Let's send our message!
UlBytes = SerSend(SerialRefNum, (unsigned char *) pText,
➥   StrLen(pText),&e);
if (e)
{
FrmCustomAlert(ErrorAlert, "Error Sending Message!","","");
}
```

SerSendWait

At times, an application will send a stream of bytes and then want to close down the serial port, switch to another application, and so on. It is important to wait until all your data has been sent before closing down the port with `SerClose`; otherwise, the trailing portion of your transmission might never reach its destination.

The arguments for `SerSendWait` are the Serial Manager library handle and the number of system ticks to wait. This second parameter is actually not implemented as of Palm OS version 4.0. The application programmer must use a value of `-1` for the timeout parameter:

```
// Let's wait for the port to flush
e = SerSendWait(SerialRefNum,-1);
```

PalmTalk: A Palm OS Serial Terminal Application

Now that you have looked at the components of a Serial Manager application, you can build a sample application. PalmTalk allows a Palm device that is sitting in its cradle to communicate with its host PC via the PC's connected COM port.

One topic that I have not addressed is the topic of receiving data during the normal operation of the program. Open, close, and send activities are initiated by the press of a button or a menu selection, but you need to display information as it is available from the port. To achieve this, you will modify the standard application event loop to check the port for incoming data.

Normally, the `EvtGetEvent()` function's second parameter is `evtWaitForever`. I have replaced this with a timeout value of 100 system ticks. The following code segment services the receive queue during each cycle. If you didn't modify the `EvtGetEvent` function's timeout parameter, the queue would be serviced only when an event such as a pen stroke or button press was processed.

The following shows how to modify the event loop to handle this:

```
void
AppEventLoop (void)
{
   EventType   event;

   do
   {
      EvtGetEvent (&event, 100);
      MainFormReadSerial();
      // Ask system to handle event.
      if (false == SysHandleEvent (&event))
      {
         // System did not handle event.

         UInt16        error;
         // Ask Menu to handle event.
         if (false == MenuHandleEvent (0, &event, &error))
         {
            // Menu did not handle event.
            // Ask App (that is, this) to handle event.
            if (false == AppEventHandler (&event))
            {
               // App did not handle event.
               // Send event to appropriate form.
               FrmDispatchEvent (&event);
            }
         }
      }
   }
   while (event.eType != appStopEvent);

   // Just in case we still have the port open!
   MainFormCloseSerial();
}
```

Note that at the end of this function, you have to make a call to close the serial port to free it for other applications.

To test this application, you need to configure your terminal emulation software for a direct connection over a COM port. The settings should be 9,600 baud, 7 data bits, and even parity with 1 stop bit. When you've built PalmTalk successfully, you should be able to get a session going between PalmTalk and HyperTerminal, as shown in Figure 21.1.

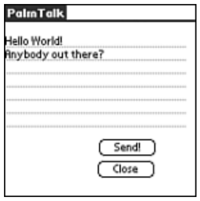


FIGURE 21.1 PalmTalk speaks!

The relevant source code for PalmTalk follows in Listing 21.1.

LISTING 21.1 PalmTalk's Relevant Source Code

```
/*
   PalmTalkmain.cpp
   Serial Manager Example Program
   Copyright (c) Bachmann Software and Services, 1999-2002
   Author: W.F. Ableson
*/

#include <PalmOS.h>
#include <SysEvtMgr.h>
#include <SerialMgrOld.h>
#include "PalmTalkmain.h"         // MainFormEventHandler ()
#include "PalmTalk_res.h"


static void    MainFormInit (FormPtr formP);
static Boolean MainFormButtonHandler (FormPtr formP, EventPtr eventP);
static Boolean MainFormMenuHandler (FormPtr formP, EventPtr eventP);
void    MainFormOpenSerial (FormPtr formP);
void    MainFormWriteSerial (FormPtr formP);
static void    MainFormVersion (FormPtr formP);


static UInt16 SerialRefNum;
static int Connected = 0;
```

LISTING 21.1 Continued

```
/*
   MainFormEnableControl:
   Parms:   formP form pointer,
            object id of button to enable/disable,
            true for enable | false to disable
   Return:  none
*/
static void
MainFormEnableControl(FormPtr formP, UInt16 objectId, Boolean enabled)
{
  // Get object index from form
  UInt16 objectIndex = FrmGetObjectIndex(formP,objectId);

  // get ptr to control
  ControlPtr controlP = (ControlPtr) FrmGetObjectPtr (formP, objectIndex);

  // set enabled
  CtlSetEnabled(controlP, enabled);

  if (enabled)
  {
    CtlShowControl (controlP);
  }
  else
  {
    CtlHideControl (controlP);
  }
}



/*
   MainFormEventHandler:
   Parms:   pEvent   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormEventHandler (EventPtr eventP)
{
   Boolean  handled = false;
```

LISTING 21.1 Continued

```
   switch (eventP->eType)
   {
      case menuEvent:
      {
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormMenuHandler (formP, eventP);
         break;
      }

      case ctlSelectEvent:
      {
      // A control button was pressed.
         FormPtr formP = FrmGetActiveForm ();
         handled = MainFormButtonHandler (formP, eventP);
         break;
      }

      case frmOpenEvent:
      {
         FormPtr formP = FrmGetActiveForm ();
         MainFormInit (formP);
         FrmDrawForm (formP);
         handled = true;
         break;
      }

      default:
      {
         break;
      }
   }
   return handled;
}

/*
   MainFormInit:
   Initialize the main form.
   Parms:   formP - pointer to main form.
   Return:  none
*/
void
MainFormInit (FormPtr /*formP*/)
```

LISTING 21.1 Continued

```
{
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            command  - command to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormMenuHandler (FormPtr formP, EventPtr eventP)
{
   Boolean handled = false;
/*
   switch (eventP->data.menu.itemID)
   {

   }
*/
   return handled;
}

/*
   MainFormButtonHandler:
   Handle a command sent to the main form.
   Parms:   formP    - form handling event.
            eventP   - event to be handled.
   Return:  true  - handled (successfully or not)
            false - not handled
*/
Boolean
MainFormButtonHandler (FormPtr formP, EventPtr eventP)
{
   Boolean handled = false;

   switch (eventP->data.ctlEnter.controlID)
   {
      case MainBtnOpenButton:   // User selected Open Serial button.
      {
         MainFormOpenSerial (formP);
```

LISTING 21.1 Continued

```
         handled = true;
         break;
      }

      case MainBtnSendButton:   // User selected Write Serial button.
      {
         MainFormWriteSerial (formP);
         handled = true;
         break;
      }

      case MainBtnCloseButton:   // User selected Close Serial button.
      {
         MainFormCloseSerial();
         // modify user interface
         MainFormEnableControl(formP,MainBtnCloseButton,false);
         MainFormEnableControl(formP,MainBtnSendButton,false);
         MainFormEnableControl(formP,MainBtnOpenButton,true);

         handled = true;
         break;
      }


  }
   return handled;
}


/*
   MainFormOpenSerial:
   Handle the main form's OpenSerial command.
   Parms:   formP    - form handling event.
   Return:  none
*/
void
MainFormOpenSerial (FormPtr formP)
{
   Err e;
   SerSettingsType sstSetup;

   // load the serial library
```

LISTING 21.1 Continued

```
   e = SysLibFind("Serial Library",&SerialRefNum);
   if (e)
   {
      FrmCustomAlert(ErrorAlert,"Failed To Load Library!","","");
      return;
   }

   // now, let's open the port itself.  We will request an initial
   // baud rate of 9600.
   e = SerOpen(SerialRefNum,0,9600);
   if (e)
   {
      FrmCustomAlert(ErrorAlert,"Failed To Open Port 0!","","");
      return;
   }

   // let's keep track of when the port is open
   Connected = 1;

   // before we can set the parameters on the port, it is a good
   // practice to retrieve them first.
   SerGetSettings( SerialRefNum, &sstSetup );

   // let's set the baud rate and other serial comms properties
   sstSetup.baudRate = 9600;
   sstSetup.flags =    serSettingsFlagBitsPerChar7 |
                        serSettingsFlagParityOnM |
                        serSettingsFlagParityEvenM |
                        serSettingsFlagStopBitsM |
                        serSettingsFlagStopBits1;
   sstSetup.ctsTimeout = serSettingsFlagRTSAutoM | serSettingsFlagCTSAutoM;

   // Ask Serial Manager to update the port settings
   SerSetSettings( SerialRefNum, &sstSetup );

   // let's clear the port, just in case some garbage is sitting there
   SerReceiveFlush (SerialRefNum, 100 );

   // change the state of some buttons on our main form
   MainFormEnableControl(formP,MainBtnCloseButton,true);
   MainFormEnableControl(formP,MainBtnSendButton,true);
   MainFormEnableControl(formP,MainBtnOpenButton,false);
```

LISTING 21.1 Continued

```
}


/*
   MainFormWriteSerial:
   Handle the main form's WriteSerial command.
   Parms:   formP    - form handling event.
   Return:  none
*/
void
MainFormWriteSerial (FormPtr formP)
{
   unsigned short usRegister;
   int i;
   Err e;
   UInt16 wIDField;
   FieldPtr pCtlField;
   MemHandle hText;
   char * pText;


   // check connected .. just in case
   if (Connected == 0) return;

   // get message from our field
   wIDField = FrmGetObjectIndex(formP,MainMessageField);
   pCtlField = (FieldPtr) FrmGetObjectPtr(formP,wIDField);
   hText = FldGetTextHandle (pCtlField);
   pText = (char *) MemHandleLock(hText);

   // let's send our message!
   SerSend(SerialRefNum,(unsigned char *) pText,StrLen(pText),&e);
   if (e)
   {
      FrmCustomAlert(ErrorAlert,"Error Sending Message!","","");
   }
   MemHandleUnlock(hText);
   // let's send along a newline, so if we can use PalmTalk on two devices
   // connected together with a null modem cable!
   SerSend(SerialRefNum,(unsigned char *) "\n",1,&e);
   if (e)
   {
```

LISTING 21.1 Continued

```
      FrmCustomAlert(ErrorAlert,"Error Sending Message!","","");
   }


}



/*
   MainFormReadSerial:
   Handle the main form's ReadSerial command.
   Parms:   none
   Return:  none
*/
void
MainFormReadSerial ()
{
   static unsigned char szBuf[1024];
   static int index = 0;
   Err e;
   UInt32 ulBytes;


   if (Connected == 0) return;
   // see if there is anything in the queue ... "peek"
   e = SerReceiveCheck(SerialRefNum,&ulBytes);

   // ensure there were no errors
   if (e)
   {
      FrmCustomAlert(ErrorAlert, "Error Checking Serial Port!","","");
      return;
   }

   // is there something waiting for us ?
   if (ulBytes)
   {
     // let's retrieve the info!

     // first, make sure the amount to read is not too large
     if ((ulBytes + index) > sizeof(szBuf))
     {
        ulBytes = sizeof(szBuf) - index - 1;
```

LISTING 21.1 Continued

```
     }

     // retrieve one byte at a time, looking for our end of message indicator
     while (ulBytes)
     {
        SerReceive(SerialRefNum,&szBuf[index],1,0,&e);
        // check for read error
        if (e)
        {
           FrmCustomAlert(ErrorAlert, "Error Reading Serial Port!","","");
           SerReceiveFlush(SerialRefNum,100);
           index = 0;
           return;
        }
        switch (szBuf[index])
        {
            case 0x0a:
              szBuf[index] = 0x00;
              // we have our new line indicator .. that will mark the end
              // of this "message" so let's display it!
              FrmCustomAlert(MessageAlert,(char *) szBuf,"","");
              index = 0;
              break;
            default:
              index++;
              break;
        }
        ulBytes--;
      } // while
   }

}

/*
   MainFormCloseSerial:
   Handle the main form's CloseSerial command.
   Parms:   none
   Return:  none
*/
void
MainFormCloseSerial()
{
```

LISTING 21.1 Continued

```
   Err e;

   // close port
   e = SerClose(SerialRefNum);
   if (e)
   {
      FrmCustomAlert(ErrorAlert,"Error during Close!","","");
   }

   // keep track of our state
   Connected = 0;

}



/*
   MainFormVersion:
   Handle the main form's version command.
   Parms:   formP    - form handling event (not used).
   Return:  true     - handled (successfully or not)
            false    - not handled
*/
void
MainFormVersion (FormPtr /*formP*/)
{
}
```

Summary

In this chapter, you learned how to make your first use of one of the Palm's means of communicating with the world around it. You should now be able to develop applications that use Serial Manager to communicate with other devices, host PCs, and even modems. In the next several chapters, you will continue exploring the Palm's communication capabilities, covering infrared, networking, and wireless communications.

22 Infrared Communications: OBEX and Exchange Manager

IN THIS CHAPTER

- What Is Exchange Manager?
- Client and Server Roles
- Using Exchange Manager

The Palm OS supports the exchange of information in and out of the device through a variety of methods and techniques. The most common mechanism for moving data on a Palm OS device is the traditional hot-sync process. Hot-sync can be thought of as a high-level protocol that is ideal for synchronizing complete databases of information between the device and the desktop computer (or server). However, there are many cases where a PDA user wants to exchange information directly with another PDA user, without going through the hot-sync process and a host computer.

For this type of peer-to-peer data exchange, a more flexible approach is required. The task of ad-hoc transferring of data and "objects" is accomplished through the activity known as *beaming***.** Beaming allows PDA users to share information easily and quickly, without the need for hot-sync software or specialized cables. Although we mostly think of a PDA-to-PDA infrared connection when we discuss beaming, in fact data can be exchanged among PDAs, cell phones, personal computers, or even embedded devices. It is not limited to only Palm OS-capable devices. This kind of interoperability is increasingly crucial in a mobile, technology-dependent society.

The object-transfer functionality described previously is offered in the Palm OS via a software component known as Exchange Manager. As a Palm user, you have no doubt already used Exchange Manager without even knowing it, because every Palm application that supports the Beam command (including the built-in applications) makes use of Exchange Manager.

This chapter, along with its companion Chapter 23, "Using the Infrared Library," provides an in-depth exploration of how to take advantage of the infrared communications capability that is inherent in every Palm device sold today.

Specifically, you'll learn about

- The Object Exchange protocol
- Adapting Object Exchange to a client/server model
- The program components required to implement beaming

In addition, this chapter presents a full working example project, which you can quickly adapt for use in your own programs.

What Is Exchange Manager?

Exchange Manager is a Palm OS-supplied shared library used by both built-in applications such as the Address Book and by custom applications alike.

Exchange Manager implements a protocol known as Object Exchange, or *OBEX*. OBEX transfers objects as streams of octets; the content of a transfer might contain an Address contact, a collection of custom transaction data, or even a complete database or shareware application.

The OBEX protocol calls for the exchange of "typed" information. This means that the receiving device must understand how to handle the "type" of the incoming data. For example, when a Palm OS user beams a business card to another user, the receiving device understands the "contact" record and properly saves it to the local Address Book's contact database. Sending an arbitrary object that is not recognized by the receiver results in an error message explaining that there is no application prepared to process the received object. For a successful object exchange to occur, there must be a *registered* application for each type of data exchanged.

The Object Exchange protocol defines the bits and bytes of the communication to exchange byte streams, but it does not require a specific transport mechanism. Although mostly associated with infrared, the OBEX protocol can run over infrared, Bluetooth, TCP/IP, or even a serial connection.

In the Palm OS environment, OBEX over infrared has been available since OS 3.0 was introduced on the Palm III device. Exchange Manager utilizes the IR Library shared library for provision of the infrared transport. Palm OS version 4 introduces the concept of *sending*, which extends the concept of object exchange to SMS and Bluetooth. This chapter's sample focuses only on utilizing Exchange Manager and does not concern itself with discussions of non-infrared exchanges. A section at the end of this chapter is dedicated to the latest options available for Exchange Manager.

Client and Server Roles

A Palm OS device can act as one of two roles in an OBEX data exchange: a *client* or a *server*. When beaming is enabled in the Palm OS Preferences applet, the Palm OS device is capable of receiving information via Exchange Manager, and is said to be in the "server" mode. Another computer or device (Palm OS or otherwise) can initiate an OBEX transfer by "pushing" data to the server-mode Palm OS device. Alternatively, the Palm OS device can initiate a transfer, thereby participating as the client in an OBEX session.

It is important to understand that there is more to beaming than just calling API functions; the operating system itself participates in these transfer sessions. This participation comes in the form of various applications launching with different "launch codes." As you saw way back in the beginning of the book, launch codes are special messages that are sent by either another application or Palm OS itself to a target Palm application's `PilotMain` routine. In the case of OBEX, Palm OS acts as a kind of traffic cop that makes sure that OBEX requests and responses are routed to the correct application.

There are key milestones in the life cycle of an OBEX session. Figure 22.1 demonstrates the process of receiving beamed data from the perspective of the server role.

Essentially, if an application on the receiving device is prepared to handle the "type" of the inbound object, that application is started by Palm OS, with a special launch code, requesting permission to carry out the transfer. If permission is granted, the application is then re-launched, this time with a different launch code, instructing the application to initiate the receiving process.

Once the object has been completely received, an application is finally launched to present the newly received object to the users; in most circumstances, the same application fulfills all roles of notification, receiving, and presentation. The section later in this chapter covering launch codes describes this process in further detail.

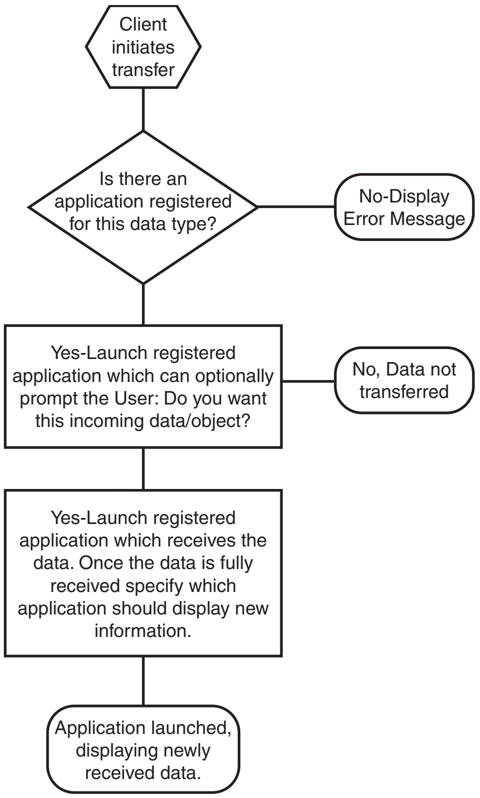

Client initiates transfer
Is there an application registered for this data type?
No-Display Error Message
Yes-Launch registered application which can optionally prompt the User: Do you want this incoming data/object?
No, Data not transferred
Yes-Launch registered application which receives the data. Once the data is fully received specify which application should display new information.
Application launched, displaying newly received data.

FIGURE 22.1 A flowchart of a typical OBEX session, from the perspective of the server (receiver).

Using Exchange Manager

This section examines Exchange Manager more closely, covering the handling of file and data types, object registration, launch codes, and sending/receiving of data.

A Sample Application Overview

The sample application presented in this chapter demonstrates exchanging information via Exchange Manager facilities. Both the client and server roles are explored. The application takes the steps necessary to receive objects and store each into a database of files.

For the purposes of this simple example, the application manages what I am calling "arbitrary files," or to be specific, files with an "*.arbitrary" extension in the name.

In the real world, simply replace "arbitrary" with whatever file or data type you want to exchange—for example a document, memo, or custom data. The sample application is designed to be an example of how Exchange Manager applications operate; the actual presentation of the arbitrary files is of little consequence for the purposes of this exercise.

The user interface for this example application is simple. The list of locally stored "arbitrary" files (objects in OBEX parlance) is presented in a simple pop-up list. When an object is selected in the list, it can be deleted or "sent" via Exchange Manager facilities. When the Send option is selected, the application takes on an OBEX client role by *PUTting* the object to a remote device.

The next sections discuss Exchange Manager API calls and launch code handling. A complete listing of our sample application's source code is also presented.

The Object Registration Model

The first step in enabling interaction with Exchange Manager requires the registration of an object "type." This registration associates an object type with a specific Palm OS application. The OBEX protocol calls for the passing of this object-type information during the transfer, however, in practice, the object type is determined by examination of the object's filename extension. This type-handling mechanism makes the OBEX protocol very similar to MIME data type handling in the HTTP environment or class registration in the Windows Explorer model.

In the MIME scenario, you might have noticed that when a Zip file is downloaded from the Internet, a Zip utility program is launched to handle the file. This is because the utility program is registered to process all Zip files. Common types of data exchanged in the Palm OS environment include

- Txt/doc—These files represent word processing documents or plain text documents.
- VCF—Virtual Business Cards or contacts.
- PRC/PDB—It is quite common for Palm OS users to beam applications and databases to one another. Some game vendors distribute "levels" to a game in the form of Palm OS databases, or *.pdb files. If these files are not protected from beaming, they can be easily exchanged between users.

There exists a function in the Exchange Manager API dedicated to the task of object-type registration: `ExgRegisterData`. Here is the function prototype:

```
Err ExgRegisterData(const UInt32 creatorID,const UInt16 id,
                    const Char * pDataTypes);
```

The first parameter specifies the creator identifier for the application associated with this object type. The sample application's `CreatorId` is `'EMSP'` so this is the first parameter passed to the function.

The second parameter represents the kind of registration taking place. The options are as follows:

- `exgRegTypeID`—Use this value when registering a MIME type, such as `text/html`.
- `exgRegExtensionID`—This option is useful when describing the data to be transferred in terms of a filename. For example, specifying `"html"` with the `exgRegExtensionID` is equivalent to specifying `"text/html"` with the `exgRegTypeID`.

Here is the usage to assign the "arbitrary" data/object type to the sample application.

```
err =  ExgRegisterData('EMSP', exgRegExtensionID, "arbitrary");
```

Data Structures

The most important data structure associated with Exchange Manager is the Exchange Socket, or `ExgSocketType`. This structure holds all of the control information during an OBEX session. Here is the structure as it is found in the Palm OS SDK and a discussion of some of the more commonly used members.

```
typedef struct ExgSocketType {
     UInt16   libraryRef;  // identifies the Exg library in use
     UInt32    socketRef;  // used by Exg library to identify this
                           // connection
     UInt32    target;    // Creator ID of application this is sent to
     UInt32   count;      // # of objects in this connection (usually 1)
     UInt32   length;    // # total byte count for all objects being
                         // sent (optional)
     UInt32   time;      // last modified time of object (optional)
     UInt32   appData;   // application specific info
     UInt32    goToCreator; // creator ID of app to launch with goto
                            // after receive
     ExgGoToType goToParams;  // If launchCreator, this contains goto
                              // find info
     UInt16   localMode:1; // Exchange with local machine only mode
     UInt16   packetMode:1;// Use connectionless packet mode (Ultra)
     UInt16   noGoTo:1;    // Do not go to app (local mode only)
     UInt16    noStatus:1; // Do not display status dialogs
     UInt16    preview:1;  // Preview in progress: don't throw away data
                           // as it's read
     UInt16   reserved:11;// reserved system flags
     Char *description;   // text description of object (for user)
     Char *type;      // Mime type of object (optional)
     Char *name;      // name of object, generally a file name (optional)
} ExgSocketType;
typedef ExgSocketType *ExgSocketPtr;
```

The `name` field is by far the most commonly used and important member of this structure. It contains the filename of the incoming object. The extension of this filename governs which application is to be notified upon receipt of the object. As depicted in Figure 22.1, if an application is registered for the data type described in the `name` member, it will be launched in order to receive the incoming object.

A program's interaction with this structure is dependent upon the role it is playing in an OBEX session. When the application is in the server role and receiving data, it receives a pointer to this structure from `PilotMain`, as seen here.

```
static UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
ExgSocketPtr s;
…
      case sysAppLaunchCmdExgReceiveData:
      s = (ExgSocketPtr) cmdPBP;
   …
   break;
…
}
```

This pointer is required in all subsequent calls to Exchange Manager functions such as `ExgAccept()`, `ExgReceive()`, `ExgDisconnect()`, and so on. While operating in the server mode, all of `ExgSocket`'s fields are pre-populated; nothing is required other than receiving the data. Once the data has been received, the application can choose to launch another application (including itself) to display the newly received data. The `goToCreator` and the `goToParams` members are available for this purpose.

When sending data in the role of an Exchange Manager client, the `name` member is used to inform the recipient of the data type and the object's name. The `description` field is used for the user-progress dialogs.

The `length` field is optional and should not be relied upon in server mode applications; however, in client mode it is a good practice to provide it. In this way the recipient can choose to ignore the inbound object if the size is too large.

The `localMode` member is a useful mechanism for testing Exchange Manager applications. By setting `localMode` to 1 when sending objects, Exchange Manager actually "loops back" to the local device rather than attempting to send to another device via infrared. In this configuration, an application may "beam" data to itself to test both the sending and the receiving functionalities.

When an application receives notification that an inbound object is available for receipt, it receives a pointer to the `ExgAskParamType`. This structure allows the application a means to accept the object, reject the object, or prompt the user to decide. Here are the relevant snippets from the SDK header file `ExgMgr.h`. Note that this structure also contains a pointer to a `ExgSocketType`.

```
typedef enum { exgAskDialog,exgAskOk,exgAskCancel } ExgAskResultType;

typedef struct {
   ExgSocketType *socketP;
   ExgAskResultType result;   // what to do with dialog
   UInt8 reserved;
} ExgAskParamType;
typedef ExgAskParamType *ExgAskParamPtr;


// Here is a common usage of the structure:
static UInt32 EMSamplePalmMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
ExgAskResultType * p;
…

      case sysAppLaunchCmdExgAskUser:
      p = (ExgAskParamType *) cmdPBP;
      p->result = exgAskDialog;
      …
      break;
…
}
```

The action taken depends upon the value assigned to the result member. This example prompts the users, asking if the object should be received and stored locally. Providing a value of `exgAskOk` causes the application to accept the object. Once the application chooses to accept the object, the receive process commences. When the receive process is complete, the record may be displayed with the assistance of the `ExgGoToType` structure.

```
typedef struct {
UInt16      dbCardNo;         // card number of the database
LocalID      dbID;            // LocalID of the database
UInt16       recordNum;       // index of record that contains a match
UInt32      uniqueID;         // position in record of the match
UInt32      matchCustom;      // application specific info
} ExgGoToType;
```

This operates in the same manner as the result of a Find operation. The members of this structure combine to uniquely identify any record on the device.

The next section builds upon this examination of the data structures by providing a narrative of each launch code and relating the data structures to the overall process of receiving objects.

Launch Codes

One of the pillars of Exchange Manager programming is the handling and familiarity with launch codes. Each of the following sub-sections discusses a particular launch code—the data structures associated with it and the role the launch code plays in an Exchange Manager session.

`sysAppLaunchCmdExgAskUser`

This launch code is invoked when an object transfer commences. The purpose of the launch code is to inform the receiver of an impending transfer, which is the first indication to an application that an object exchange is underway. This gives the receiving application the option to accept, reject, or conditionally accept the incoming object.

The option/choice mechanism is conveyed through the `ExgAskParamType` data structure presented earlier. This data structure is passed into the application via `PilotMain`'s second parameter, a `MemPtr` named `cmdPBP`. In order to be used, the passed pointer must be cast to an `ExgAskParamType` pointer.

The `ExgAskParamType` structure contains two functional members. The first is an `ExgSocketPtr` containing information relevant to the incoming object and the ensuing transfer. This information is available to assist the application in deciding on whether to accept the object. For example, if the `length` parameter is populated the application can determine whether there is sufficient storage space available for the object. The second member is an enumerated type used as a flag. It informs Exchange Manager what step to take next. When the application ends, the value in this flag is examined. If the value is `exgAskOk`, the operating system launches the application with the `sysAppLaunchCmdExgReceiveData` launch code, informing the application that it must receive this object.

In the scenario where the user is to be prompted for acceptance of an object, the default dialog may be overridden using the `ExgDoDialog()` function. However, if the application does not handle the `sysAppLaunchCmdExgAskUser` launch code, the default behavior is for the operating system to display a simple “Do you want this Object” dialog to the user. If the user chooses to accept the incoming object via either the default dialog or the custom dialog, the operating system takes the next step of re-launching the application with the next launch code, `sysAppLaunchCmdExgReceiveData`.

`sysAppLaunchCmdExgReceiveData`

Once the application chooses to receive an object, it is invoked with the `sysAppLaunchCmdExgReceiveData` launch code. For this launch code, the `cmdPBP` parameter from `PilotMain` contains a pointer to an `ExgSocketType` structure. This structure is required in all subsequent Exchange Manager API calls.

The usual activity that takes place during this type of launch is the receipt of data via the Exchange Manager API and subsequent storage of the data locally into a database. It is important to understand that the application’s global variables are likely not available at this time. For example, the receiving Palm OS device may be editing a memo at the time the transfer occurs, so any global variables associated with database IDs, database references, or memory buffers will not be initialized. It is possible to check the launch flags to determine whether globals are available, but the better course is to assume that they are not and construct the receiving routines in a self-contained manner.

Once the data has been received, it can be displayed with the assistance of the `sysAppLaunchCmdGoto` launch code.

`SysAppLaunchCmdGoto`

This launch code is available for the receiver to optionally cause the device to display the newly received object. When the transfer is complete but prior to the `sysAppLaunchCmdExgReceiveData` invocation completion, record-specific “key” information about the newly received object can be stored in the `ExgSocketPtr`’s `goToCreator` and `goToParams` members. These fields become available to the application in a subsequent launch with the `sysAppLaunchCmdGoto` launch code.

When invoked, the application uses the information provided in the `cmdPBP` parameter cast to a `GoToParamsType` pointer to retrieve and display the record.

Unlike the previous two launch codes discussed, when a `sysAppLaunchCmdGoto` launch code is received, the application does not terminate, but rather continues to run upon this launch.

Receiving Objects

Receiving data with the Exchange Manager APIs takes place during the `sysAppLaunchCmdExgReceiveData` invocation. Using the `ExgSocketPtr` received from the `cmdPBP` parameter of `PilotMain`, the object is in turn accepted with `ExgAccept()`, received with successive calls to `ExgReceive()`, and then finally concluded with a call to `ExgDisconnect()`.

Ideally, the size of the object is available in the length member of the `ExgSocketPtr`; however it is an optional field that cannot be relied upon. As such, the prescribed manner for receiving objects is to read continually until the number of bytes read is zero. The sample application receives a single buffer's worth of data for demonstration purposes. Custom applications should decide on a storage mechanism and technique around the `DmResizeRecord()` to accommodate large objects.

Here are the function prototypes:

```
Err ExgAccept(ExgSocketType *socketP);

UInt32 ExgReceive(ExgSocketType *socketP, void *bufP,
                  UInt32 bufLen, Err *err);

Err ExgDisconnect(ExgSocketType *socketP, Err error);
```

And a snippet from the sample application showing the calls in action:

```
s = (ExgSocketPtr) cmdPBP;
switch (ExgAccept(s))
{
   case errNone:
      // begin to receive data
      if (s->name) StrPrintF(buf,"name is [%s]",s->name);
      MsgBox(buf);
      MemSet(buf,sizeof(buf),0x00);
      bytes = ExgReceive(s,buf,sizeof(buf) - 1,&error);
      if (bytes)
      {
         StoreData(s->name,buf,bytes);
         MsgBox(buf);
      }
      else
      {
         StrPrintF(buf,"Failed to receive data!Error [%d]",error);
         ErrBox(buf);
      }
      ExgDisconnect(s,0);
      break;
   case exgErrBadLibrary :
      ErrBox("Bad Library on accept?");
      break;
   case exgErrStackInit:
      ErrBox("Stack Init errors");
      break;
   default:
      ErrBox("Unrecognized return from ExgAccept");
      break;
}
```

The sample application calls a locally defined function named `StoreData()`, which takes care of storing the newly received object into an application-specific database.

Sending Objects

Most of the discussions up to this point have revolved around the launch codes and process flow of receiving data. This section examines the process of sending data, as a client of the Exchange Manager API. Unlike receiving, sending is not initiated as a result of a Palm OS launch code. Sending can be initiating from the common button select event or from a menu-selection event.

Sending data mirrors the receive operation. The first task is to initialize an `ExgSocketType` structure. The best practice is to zero out the entire structure and then simply populate the members necessary for the transfer. The commonly used members are as follows:

- `name`—Contains the filename or type of the object.
- `description`—Contains a description used for displaying to the user on both the sending and receiving side.
- `length`—An optional parameter that can contain the object's size. This can be used to assist the recipient in deciding on whether to receive the object.
- `count`—This field is normally set to 1, but can be a larger value if multiple objects are transferred in a single session.

A call to `ExgPut()` initiates the sending process. If this succeeds, repeated calls to `ExgSend()` effectively transfer the object. Once all of the data has been transferred, a call to `ExgDisconnect()` informs the application of any errors encountered during the transfer.

Here are the function prototypes:

```
Err ExgPut(ExgSocketType *socketP);

UInt32 ExgSend(ExgSocketType *socketP, const void *bufP,
               UInt32 bufLen, Err *err);

Err ExgDisconnect(ExgSocketType *socketP, Err error);
```

Note that all three functions take the `ExgSocketType *` as the first argument to the function. Any errors encountered by the `ExgPut` or the `ExgSend` functions must be communicated to the `ExgDisconnect` function as the second parameter. Here is a code snippet from the sample application:

```
ExgSocketType sock;
Char namebuf[50];
Char buf[50];
Char *p;

   MemSet(&sock,sizeof(ExgSocketType),0x00);
   StrPrintF(namebuf,"%s%s",name,EXTENSION);
   sock.name = namebuf;
   sock.description = namebuf;
   sock.length = StrLen(data);
   sock.count = 1;

   err = ExgPut(&sock);
   if (err)
   {
      StrPrintF(buf,"Failed to initiate OBEX Put. Error [%d]",err);
      ErrBox(buf);
      ExgDisconnect(&sock,err);
   }
   else
   {
      // let's send this data across ....
      bytes = sock.length;

      p = (Char *) data;
      err = 0;
      while (err == 0 && bytes > 0)
      {
         bytessent = ExgSend(&sock,p,bytes,&err);
         bytes -= bytessent;
         p += bytessent;
      }
      if (err || bytes > 0)
      {
         ErrBox("Error, data not completely exchanged!");
      }
      ExgDisconnect(&sock,err);
   }
```

Note that the call to `ExgSend` is embedded in a loop for successive calls as necessary. It is not guaranteed that a call will transfer all of the requested bytes. The loop repeatedly calls `ExgSend` until all of the data has been sent or until an error occurs.

Exchanging Databases

Beyond transferring individual records, it is also common to exchange entire Palm OS databases and applications via Exchange Manager. The `ExgDBRead()` and `ExgDBWrite()` functions allow an application to read and write a database in its internal format for the purposes of transfer to another device. This can be used to move supplemental databases such as fonts or game levels from one device to another.

Targeting a Specific Application

The example used in this chapter relied upon the `name` field to convey the data type; however, it is also possible to send a piece of data directly to a specific application on the receiving device. This is accomplished by populating the target member with the creator ID of the intended recipient. This approach is useful when communicating from one instance of an application to another instance running on another Palm OS device. However, when the goal is to exchange data with a variety of devices in a more heterogeneous environment, the `name` parameter is the recommended approach.

The Send Paradigm—Beyond Infrared

With the advent of additional communications technologies such as Short Messaging Service (SMS) and Bluetooth, the Palm OS has extended Exchange Manager beyond the confines of infrared communication. Object Exchange, or OBEX, has come to exist as a high-level protocol over each of these transport technologies.

In order to further leverage the OBEX protocol, the Exchange Manager APIs have been expanded. Starting with version 4 of the Palm OS, additional functions expose these capabilities. Note that some of these capabilities are not downward-compatible with the infrared transport.

Infrared Exchange Manager operations continue to be limited to the `PUT` operation; however, the new APIs accommodate the `GET` operation with the `ExgGet` and `ExgRetrieve` functions. There are also additional convenience and management functions useful for manipulating the relationships between registered applications and the data types they handle.

There is now also a preview capability. In prior releases of Exchange Manager, an application deciding on accepting or rejecting an incoming object was confined to the few elements available in the `ExgSocket` structure. The `ExgNotifyPreview` function adds the ability to display a portion of the data to the users from the `ExgDoDialog()` function when deciding whether to accept the object.

Perhaps the most significant enhancement to Exchange Manager beyond the additional transport choices is the mechanism of URL schemes used to specify which transport should be used in a particular exchange. A URL scheme may be `beam`, `send`, or `local`. If a URL scheme is prefixed with a question mark, `?`, Exchange Manager provides a dialog asking the user to decide which transport technology to use for the exchange. The capabilities and flexibility of the `ExgRegisterData` mechanism have been expanded with the addition of the `ExgRegisterDataType` function.

All of these enhanced capabilities are available in Palm OS 4.0 version and later.

Summary

This chapter has presented the fundamentals of Object Exchange with Exchange Manager for Palm OS. This content equips the developer to add basic beaming support to an application, thereby extending the reach and capability of the application's features and function. The sample covers the basics, relying only on version 3 of the Palm OS functionality in order to reach the widest possible audience and to focus on the core activity of exchanging objects.

For applications requiring additional functionality, either in terms of object exchange design patterns such as query response or flexibility of transport, you should consider basing development around Palm OS version 4.0 or higher to leverage the latest in Exchange Manager features.

23

Using the Infrared Library

IN THIS CHAPTER

- The IrDA Standard
- Implementing Infrared Connectivity in a Palm Application
- IrDemo: Building a Palm OS IR Application

In Chapter 21, “Serial Communications,” you learned how serial communications can further improve productivity by providing the opportunity to print and share all the useful information. Unfortunately, to make it all work, you have to carry along a multitude of cables, wires, adapters, and power supplies. We live in an increasingly wireless world and to maximize the usefulness of our Palm devices (and to send us off the scale in the “cool” department), we need to *beam*. That’s right, Star Trek meets Silicon Valley! With infrared communications at our command, we become truly portable (if not the most advanced gadgeteer in the office, which is a prize in itself).

Although communications options are exploding through a rich assortment of add-on modems, cards, and adapters, every Palm compatible device comes standard with something all applications can rely on: an infrared port. This is a comforting constant, because the built-in applications universally support a “beam” menu option. Despite the fact that infrared is a relatively short-range communications option, there are many worthy applications available to the Palm. For instance, take a quick look at your digital cell phone. Many new phones are equipped with built-in infrared capability. (Just don’t send us the phone bill!)

There are a few types of infrared (IR) communications available to the Palm developer. The specific needs and resources available for a particular application will dictate the approach you take for IR communications on the Palm. In Chapter 22, “Infrared Communications: OBEX and Exchange Manager,” we reviewed the OBEX (Object Exchange) services available on Palm OS, and found that

OBEX was particularly suited for dealing with highly structured data needing to be transferred between two devices that support the OBEX protocol. Although arguably the easiest IR development option for the beginner, a significant downside of OBEX is that not every IrDA device supports it. Thus for more universal device compatibility, Palm provides a lower-level interface called the Infrared Library.

In this chapter, you'll learn

- The infrared capabilities of Palm OS devices
- What functions are available in the Palm OS IR Library
- How to use IR Library to implement infrared communications in your application

In this chapter I continue the discussion of communications topics with the Infrared Library, Palm OS's implementation of the IrDA standard. IrDA (the Infrared Data Association, `http://www.irda.org`) is the standards organization setting the direction of infrared communications for devices and appliances. Major manufacturers and software houses are members of IrDA today, and this effort will continue to grow in the years ahead. Learning to program to the IrDA specification will put you in the fast lane for Palm OS development. Where else would you want to be? I conclude this chapter by looking at the included sample application, IrDemo. IrDemo provides a framework for writing head-to-head, Palm-to-Palm games (err, I mean applications).

The IrDA Standard

The IrDA specifications call for two types of infrared communications, namely SIR (Standard IR) and FIR (Fast IR). SIR effectively provides serial communications, thus replacing the copper wires in your modem cable with light waves in the infrared spectrum. The transmission speeds of SIR match those of traditional serial communications in the 300–115,000 bps range. Fast IR is capable of delivering substantially higher transmission speeds. FIR is being used on some platforms to implement local area networking connectivity because it is capable of throughput of up to 4Mbits per second. To put this in perspective, your normal Ethernet network is capable of either 10 or 100Mbits per second.

Like many communications specifications, the IrDA standard defines many protocol layers that form the IR stack. Each layer offers distinct services upon which additional services are built. At the bottom of the stack is SIR/FIR, which is strictly hardware, involving an IR device that emits light waves and a controller. The controller takes the form of a UART or other chip (for FIR). Riding atop the physical layer is the Infrared Link Access Protocol (IrLAP). This layer provides the actual data path for

IrDA communications. There is one IrLAP "connection" per IR device. The Infrared Link Management Protocol (IrLMP) handles one or more sessions over the single IrLAP connection. Higher levels of the stack include TinyTP, IrCOMM, IrLAN, IrLPT, and OBEX. Additional layers coming into play include protocols for IR keyboards. This is a 10,000-foot view of the stack. You are encouraged to visit the IrDA Web site for more detailed information.

Not all the IrDA standards are implemented by Palm OS. However, because the required layers are supported in the IR Library, you can roll your own implementations of the missing layers. The Palm's Exchange Manager implements OBEX, but you will need to develop the code to interact with other devices such as a printer using either IrCOMM or IrLPT. The sample program demonstrates my own IrDemo implementation.

Palm OS IR Capabilities

The first Palm OS release to support infrared communications was Palm OS 3.0. Therefore, IR applications require the 3.0 SDK. The addition in Palm OS 3.0 of primary concern is `irlib.h` (and the IR Library itself, of course). `irlib.h` lays out all the data types needed for IR programming.

The Palm device's single UART chip provides services for both serial and infrared communications. As I touched upon briefly in Chapter 21, it is important to reiterate that the Palm III device's sole UART controls both serial and IR communications. This means that you cannot operate the IR port and the serial port simultaneously. The mutual exclusivity between IR and serial communications also has implications for the debugging environment. In short, you cannot use the Metrowerks debugging environment to trace IR calls. In addition, POSE does not currently emulate IR functionality. Included in the IrDemo application is an example of debugging on the Palm, using a home-grown `printf` function.

IR Library Essentials

Before I jump right into the IR Library's API, it is important to have a brief discussion on the architecture of an IR application.

When your Palm application wants to communicate with another device, such as another Palm device, a printer, or cell phone, the first task is to "discover" the other device. This is essentially the IR device looking around for other devices to talk to. Once this process is complete, the application will have the address of the remote device and can initiate a connection. Once you have a low-level connection (IrLAP) to the other device, you need to look up the service you want to communicate with. The Information Access Service (IAS) provides a database of information for the services available for a particular device. When you have received the appropriate service attribute information from IAS, the application can establish a session with the appropriate service. Let's look at a real-world analogy.

Suppose you need an electrician. You open the Yellow Pages and look under Electricians. This is akin to the IR device performing discovery. It is looking for devices that "speak IR."

Now that you are on the correct page, you review each of the advertisements, selecting the most appealing electrician service and recording the phone number. In the same way, a discovery process on the Palm might find many devices but needs to select one with which to communicate. This will yield an address to use in the connection process.

Okay, so you call the electrician's office and ask for someone with the expertise to install a refrigerator. On the Palm, you initiate an IAS query looking for, say, IrLPT for printing services.

The receptionist replies, "Dial extension 123." The IAS query responds with something called an "LSAP selector." In the world of IrDA, LSAP stands for Logical Service Access Point and it is a reference to a service that is made available within the IrDA subsystem. In this example, the LSAP selector would provide you with a reference to where IrLPT is found.

You call extension 123 and speak with the electrician who can help you install the refrigerator. The Palm IR application connects to the IrLPT service and can now print!

The IR Library relies upon two callback functions for event notification. You must provide both of the callback functions in the application because there is no default event handler provided by Palm OS. The first callback function, named `IrHandler` in IrDemo, shoulders the majority of the workload in the application. `IrHandler` receives all notifications for `IrStack`-related events, such as IrLAP connect and disconnect, IrLMP session requests and confirmation, data receipt, discovery completion, and so on. This function is installed during the `IrBind` function call. The other callback function I named `IASHandler`. `IASHandler` receives notification when IAS queries have completed.

Because callback events can occur at any time in the Palm application, it is important to avoid the use of alerts or other potentially time-consuming functions. In an effort to provide detailed information during the execution and avoid these problems, IrDemo uses a simple `printf` function for displaying information. The messages appear in an area of the screen toward the bottom of the Palm's display area.

Without further delay, let's work our way through the IR functions and build our application!

Implementing Infrared Connectivity in a Palm Application

Because it is a shared library (see Chapter 29, “Shared Libraries: Extending the Palm OS”), to use the IR Library, you must first load it with the following code:

```
Err e;            // For error result
UInt refNum;         // 'Handle' to library

e = SysLibFind( irLibName, &refNum );
```

(Note: `irLibName` is defined in `irlib.h`.)

After this call, `e` should be `0`, indicating that the IR library was successfully found. If the library is not found, a non-zero error code is returned.

Once the library is loaded, you must open it for the application’s use:

```
e = IrOpen( refNum, irOpenOptSpeed9600 );
```

`irOpenOptSpeed9600` is defined in `irlib.h` along with other constants indicating the initial speed for the port. This is similar to the way in which the Serial Manager works, as described in Chapter 21.

Again, this call should return `0` for success; otherwise it will return a non-zero error.

Once the `IrOpen` call has been made successfully by an application, it must call `IrClose` prior to application termination. Due to the way in which Palm applications terminate when the task is switched, it is important to detect the application switching to clean up properly.

Now that the library is opened by the application, you need to initialize it with a function named `IrBind`. Binding will associate an `IrConnect` structure, defined in the application, with a callback function that the IR Library uses to notify you of completion of certain IR-related events:

```
e = IrBind( refNum, &irCon, IrHandler );
```

See `irlib.h` for a description of the `irCon` parameter. It is actually a structure of type `IrConnect`.

`IrBind` will return `0` if the binding is successful; otherwise it will return a non-zero error code.

Now that you have bound an `IrConnect` structure and the `IrHandler`, you go on to advising the `IrStack` who you are. This is done with the `IrSetDeviceInfo` function. The return value of this function is not the generic Palm OS error type, `Err`, but rather the type `IrStatus`. See `irlib.h` for a description of this type and the possible values it can hold:

```
IrStatus irStat;
static Byte OurDeviceInfo[] = {IR_HINT_PDA, IR_CHAR_ASCII,
➥ 'P','A','L','M','D','E','M','O'};

if (IrSetDeviceInfo( refNum, OurDeviceInfo ,OurDeviceInfoLen ) !=
                         IR_STATUS_SUCCESS)
{
    IrUnbind( refNum, &IrCon );
    IrClose( refNum );
    printf("IrSetDeviceInfo Failed!" );
    return;
}
```

`OurDeviceInfo` is a byte array that cannot exceed the size defined in `irlib.h` of `IR_MAX_DEVICE_INFO`. This array contains hint bytes. The hint bytes are bit masks to indicate the type of device in this application. If there is more than one hint byte to be used, you can use `IR_HINT_EXT`, which indicates that the device info contains an additional byte of hint information. Here is an example:

```
// Device info for standard irComm device
static Byte irCommDeviceInfo[] = {
    IR_HINT_PRINTER|IR_HINT_EXT, IR_HINT_IRCOMM, IR_CHAR_ASCII,
    'I','r','C','O','M','M'};
```

This function should result in `IR_STATUS_SUCCESS`.

At this point, you have successfully loaded and opened the IR Library. You have bound it for use and told the `IrStack` who you are. If you want to advertise a service for other devices to connect to, you use the IAS database. This is a generic database. Each IrDA-compliant device maintains this repository to hold information regarding which services the device offers. This is similar to the way TCP service maps names to ports (such as `FTP -> 21` or `WWW -> 80`):

```
static Byte OurDeviceName[] = { IAS_ATTRIB_USER_STRING,
  IR_CHAR_ASCII,
  8,'P','A','L','M','D','E','M','O'};
static Byte OurDeviceNameLen = sizeof(OurDeviceName);
/* "Standard" class name for our demo is IrDemo with attribute of
   IrDA:IrLMP:LsapSel
*/
static Byte irdemoQuery[] =  { 6,'I','r','D','E','M','O',
18,
'I','r','D','A',':','I','r','L','M','P',
':','L','s','a','p','S','e','l'};
```

```
const irdemoQuerySize = sizeof(irdemoQuery);
/* Result for IrDemo */
Byte irdemoResult[] = {
    0x01,                    /* Type for Integer is 1 */
    0x00,0x00,0x00,0x02      /* Assumed Lsap */
};
/* IrDemo attribute */
const IrIasAttribute irdemoAttribs = {
(BytePtr) "IrDA:IrLMP:LsapSel",18,
(BytePtr)irdemoResult, sizeof(irdemoResult)};
static IrIasObject irdemoObject ={
  (BytePtr)"IrDemo",6,1,
  (IrIasAttribute*)&irdemoAttribs};


IrIAS_SetDeviceName( refNum, OurDeviceName,OurDeviceNameLen);
IrIAS_Add( refNum, &irdemoObject);
```

To connect to another device, you must find the device's address. To obtain this, you use the function `IrDiscoverReq`:

```
// Initiate a discovery and IrLAP connection
while ( ++lCounter <= lTimeout)
{
   irStat = IrDiscoverReq( refNum, &IrCon );
   switch (irStat)
   {
      case IR_STATUS_MEDIA_BUSY:
         printf("Media Busy");
         continue;
      case IR_STATUS_FAILED:
         printf("Failed in Discovery");
         IrUnbind( refNum, &IrCon );
         IrClose( refNum );
   FrmCustomAlert(ErrorAlert,
"Failed to Discover.  Ending Application","","");
         MemSet(&evtExit, sizeof(EventType), 0);
         evtExit.eType = appStopEvent;
         EvtAddEventToQueue(&evtExit);
         return;
      case IR_STATUS_PENDING:
         // This is the one we want!
         // At this point we need to wait for the discovery process to
         // complete ...
         printf("Discovery Pending!!!");
         return;
   }
} // while
```

irStat can come back with one of the following values:

> IR_STATUS_MEDIA_BUSY indicates that the media is busy and you should retry the function.
>
> IR_STATUS_FAILED indicates an error in the stack.
>
> IR_STATUS_PENDING is the one you want; it indicates a successful start of the discovery process.

Because it is possible for the IrDiscoverReq function to come back busy a few times and then become pending, you wrap this call into a while loop with a timeout on the iterations. This gives the application a healthy chance of finding another device, able to withstand a couple of media busy responses without a disappointing failure in the connection process.

The completion of the discovery process is notified via the callback function registered during the IrBind call, namely IrHandler.

Once discovery has completed successfully, you will have the address for a remote device. Actually, you might have many devices in range, and you will need to sift through them all to select the one you want. You can sort through any available IR devices by examining the hint bytes and nickname that is returned by the discovery process. Once you have selected a device, you want to establish an IrLAP connection. You do this with the function IrConnectIrLap:

```
// Check for a valid device
if (pCBParms->deviceList->nItems == 0)
{
   printf("No Devices Found!");
   return;
}

// At least one device has been found, we will
// assume that the first (and probably only!) device
// found is the one we want.
g_irDevice = pCBParms->deviceList->dev[0].hDevice;
printf("Found %d.%d.%d.%d",g_irDevice.u8[0],g_irDevice.u8[1],
         g_irDevice.u8[2],g_irDevice.u8[3]);
```

```
// Let's make an IrLAP connection to this address
while ( ++lCounter <= lTimeout)
{
   irStat = IrConnectIrLap( refNum, g_irDevice );
   switch (irStat)
   {
      case IR_STATUS_MEDIA_BUSY:
         printf("IrLap Media Busy");
         continue;
      case IR_STATUS_FAILED:
         printf("Failed in IrConnectIrLap" );
         return;
      case IR_STATUS_PENDING:
         // This is the one we want!
         // At this point we need to wait for the connect process to
         // complete ...
         printf("Connect Lap Pending!!!");
         return;
   }
} // while
```

You are looking for IrLAP to return `IR_STATUS_PENDING`.

As in the discovery process, you wrap this `IrConnectIrLap` function in a `while` loop with a timeout to give it a chance to connect without a single `IR_STATUS_MEDIA_BUSY` pushing you off course.

When the IrLAP connection has been established, you next want to find out what services the remote device is offering. Here is where the IAS database comes in. You will query the device for a specific service that you are interested in. You will actually execute two IAS queries. The first will demonstrate obtaining the device name. The device name is a required field to be maintained in the IAS, as defined by IrDA:

```
// Initiate query for remote service we are interested in
// This first query will provide the remote device name
// as defined by the device's IAS

IrIAS_StartResult(&clientQuery);
clientQuery.result = queryResult;
clientQuery.resultBufSize = sizeof(queryResult);
clientQuery.callBack = IASHandler;
clientQuery.queryBuf = irGetQuery;
clientQuery.queryLen = irGetQuerySize;
```

```
IrIAS_Query(refNum, &clientQuery);
// Now that we have the device we want
// we need to determine how to connect to it, ie. what lsap?
IrIAS_StartResult(&clientQuery);
clientQuery.result = queryResult;
clientQuery.resultBufSize = sizeof(queryResult);
clientQuery.callBack = IASHandler;
clientQuery.queryBuf = irdemoQuery;
clientQuery.queryLen = irdemoQuerySize;
IrIAS_Query(refNum, &clientQuery);
```

The last area to look at is the `IASHandler` callback function. This function is invoked when the results of an IAS query are ready. Because the IAS database stores information in an unstructured format, each attribute must be stored with a data type identifier. When processing the results of an IAS query, you first look at the data type, and then process the value:

```
switch ( IrIAS_GetType(&clientQuery) )
{
   case IAS_ATTRIB_MISSING:
      printf("Attribute is Missing?!");
      break;
   case IAS_ATTRIB_INTEGER:
      printf("Get Integer Value");
      IrCon.rLsap = IrIAS_GetIntLsap(&clientQuery);
      irPack.len = 0;
      // We have the address for the service we want to connect to
      // Let's establish the LMP session

    ...
}
```

The connection process ”propels” itself along via the callbacks. As a review, the steps to connect to a device via the IR Library are as follows:

1. Load the IR Library using `SysLibFind()`.
2. Open the IR port using `IrOpen()`.
3. Initialize the port using `IrBind()`.
4. Use `IrDiscoveryReq()` to obtain the device’s address.

5. From the `IrHandler` callback function, when discovery finishes, you request the `IrLapConnection()` with `IrLapConnectReq()`.
6. When this completes, you query the IAS for the service you want.
7. When the IAS is complete and `IASHandler` is called, you make a request for an LMP connection with `IrConnectReq()`.
8. When this is complete (signaled, of course, by the `IrHandler` callback), you have an up-and-running connection to the other device!

IrDemo: Building a Palm OS IR Application

The IrDemo application is designed to be informative and provide you with a launching pad for your own IR projects. You should understand that it is not intended to be a production-ready application. In a number of areas, I have left comments for to-do's, such as handling the case when a connection request is unsuccessful.

To run the application, you need two Palm devices. (If you have read this far, you might even have three Pilots!) The interface has four buttons: Start, Connect, Send, and Finish.

Place the devices head to head so the IR ports can see one another. Select Start on both devices. You should see some status information scrolling at the bottom of the display. On one (and only one) device, select the Connect button. The applications will display status information, indicating the connection activities. Note that the messages will differ on each device. At this point, you will see connection confirmation on both devices, and you can select the Send button on either device. The data will be received and displayed. When complete, select the Finish button on each device, and the application will terminate.

An interesting thing to try is to move the devices apart and note the messages that are displayed. Move the devices back together (so IR transmission can continue), and notice the display. Move the devices during the discovery process.

Summary

The IrDemo application is designed to demonstrate the fundamentals of IrDA. I hope this will help you develop your own applications, whatever they are used for. Our own Bachmann Print Manager product uses infrared connectivity to enable graphics and text printing on popular laser printers. With so many devices supporting the IrDA standard, there is certainly a world of possibilities for creating special capabilities on the Palm device.

For further reading, check out the following resources:

- Infrared Data Association (IrDA): `http://www.irda.org`
- Linux IR Project: `http://www.cs.uit.no/linux-irda/`
- The Palm OS SDK documentation

24 A Survey of Wireless Options for Palm Developers

IN THIS CHAPTER

- Options for Wireless Communication
- The Developer's Perspective

Arguably there is no aspect of mobile computing that fires the public's imagination more than wireless communications. Indeed, when you contemplate the ability to access the vast wealth of the Internet from the Palm of your hand and communicate with friends and co-workers, anytime, anywhere, it is easy to see why there is so much pent-up excitement surrounding this topic.

It seems like we have been waiting for this marvelous future to happen for quite some time now. I remember just a few years ago attending a wireless and mobile computing tradeshow, and one of the speakers joked that, although many were calling that year the "Year of Wireless," the same tag had been applied to the previous ten years! The truth is that components of the wireless dream are in fact here, but depending on who you are, what your expectations are, and what you want to do with wireless, you might find that the realities of today's wireless capabilities fall short of the promise.

Part of the problem is that, like so many new technologies, wireless has been hyped to the point where it will take a very long time to live up to all the expectations involving pervasive anytime/anywhere fast connectivity. I will say that wireless is one of those few technologies in which the hype is well deserved: Wireless will almost certainly fulfill its promise, and change the way we work and live. The problem is not whether it will or won't happen, but when. I think if someone stood up in 1980 and proclaimed that each of us would ultimately have a personal computer on our desk that had a 10GB hard drive and a 2GHz processor, he would clearly have seen his predictions come true, and given a long enough time span for it to occur, it might not have even seemed outlandish to predict. Timing and expectations are so important as we try to forecast the future of computing.

This chapter provides a brief introduction to the landscape of available wireless communications options for software developers building mobile computing applications.

Specifically, you will learn

- Which wireless communication options are available
- Hardware and software choices for each option
- How to choose the right option for your application
- The Palm SDK APIs that cover wireless

In subsequent chapters, I build on this information, and provide more in depth, low-level treatments of some of these wireless technologies from the perspective of the Palm developer.

Options for Wireless Communication

There are by now a large number of ways for PDAs to connect wirelessly to other computers. Each method has its pros and cons, and for people new to the industry, these methods can be confusing to understand and choose from. It does not help that there are so many components to the whole solution, including hardware, software, adapters, protocols, carriers, and networks, many of which do not work with one another.

This section lays out the relevant connectivity options available today for Palm OS handhelds. Although it is beyond the scope of this book to explain the entirety of wireless communications (that's a book in and of itself!), you should still be able to get a good starting point for understanding the options.

I (somewhat arbitrarily) divide the options into two categories: short-range wireless communications and wide-area communications. As you'll see, there is some overlap in terms of the capabilities delivered to the users, but it's as good a categorization as any.

Short-Range Options

This section provides an overview of the short-range wireless communications options available to the Palm programmer. Among them are infrared communications, Bluetooth, and 802.11 (also known as Wi-Fi).

Infrared

Infrared communications have been bundled with every Palm device manufactured going back to the Palm III. As you learned in Chapters 22, "Infrared Communications: OBEX and Exchange Manager," and 23, "Using the Infrared Library," infrared communications can be used to create one-way and two-way information exchange between two PDAs, or between a PDA and an IrDA-compatible device such as a printer.

Infrared on Palm devices is the shortest and most limited of the short-range communications options. Connections must be "line of sight" (no obstacles blocking the beam), and must be within a range of approximately three feet. (Note that some Sony Clie devices come with an enhanced IR port that can communicate over slightly longer distances such as the width of a living room.)

On the handheld side of things, every PDA has all it needs to support infrared communications. An IR port comes with every device, an IrDA compatible stack is bundled with Palm OS, and the Palm built-in applications are all enabled to beam data to and from other PDAs.

For short-duration tasks, things such as beaming an address book entry from one PDA to another, or printing to an IR-enabled printer, infrared is very reliable. The line-of-sight requirement and the limited range of the beam combine to make this a poor choice for longer duration activities such as Internet communications. Furthermore, on the Palm OS there is no native support for TCP/IP communications over IR.

Infrared can be used for hot-sync, but obviously requires the host computer to have an infrared port.

Bluetooth

Bluetooth, a newer wireless communications protocol that has won industry support, addresses some of the limitations of infrared described previously. In contrast to infrared, Bluetooth is not limited to line-of-sight connections; it can communicate around and through walls. It also has a wider range than IR, communicating with other Bluetooth-enabled devices up to 30 feet away.

Bluetooth also can serve as a base platform for supporting additional higher-level protocols, such as serial communications and TCP/IP. Thus it becomes possible to perform networking over a Bluetooth connection. Because it supports higher communication speeds than IrDA, networking of larger amounts of data becomes more feasible.

Bluetooth has potential for many of the same uses as infrared, including wider-range beaming of data among PDAs, and printing to Bluetooth-enabled printers. It can also be used for "personal area networks," providing communication among multiple devices in your personal space such as cell phones and pagers. Finally, with a Bluetooth network access point in range, it can be used as a gateway to accessing network resources including email and Internet content.

The main problems with Bluetooth are availability and end-user experience. Unlike infrared, there are not any Palm OS handheld devices that come with a built-in Bluetooth radio, meaning that users will need to obtain a Bluetooth adapter from either Palm or a third party. Although prices are coming down for these adapters, they are still expensive enough to make one pause.

The end user experience is also less than seamless in today's Palm OS implementation. Because Bluetooth ports don't "point" at one another, Bluetooth must go through a bootstrap process of "discovering" other Bluetooth devices in the 30-foot range it is operating in. This can take a few seconds to perform, and when it is complete the users are presented with a list of Bluetooth devices to choose from.

If there is only one other Bluetooth device in range, this represents a simple tap. But in an office environment with many Bluetooth-enabled PDAs, printers, cell phones, access points, and other devices, the users must wade through screens of available devices to choose from. This can be frustrating and confusing to the end users who may already know the target device they wish to connect to.

Wi-Fi

Much excitement has arisen recently over the growth in a wireless technology known alternately as 802.11b (along with variations 802.11a and 802.11g) and Wi-Fi (a slang term commonly used in place of 802.11). Wi-Fi is a way to wirelessly network computers over short-to-medium-range distances (up to several hundred feet, depending on an indoor or outdoor location). Like Bluetooth, Wi-Fi is non-directional, and does not require a line-of-sight to make a connection.

A Wi-Fi connection is typically made from either a PDA or a laptop to a wireless LAN access point. The access point connects to your network router/hub, and forms what can be thought of as an invisible Ethernet connection between the hub and any Wi-Fi-enabled computer in range. Speeds go up into the multi-megabyte range, making it suitable for larger amounts of data.

Wi-Fi is implemented with networking in mind, and thus is highly suited for file sharing, networked printing, office applications, email, and shared Internet access. With Wi-Fi you can more or less do the same types of things you can do with a laptop and an Ethernet card. Wi-Fi access points are springing up in offices, homes, and public places like airports and coffee shops, providing laptop and PDA users with easy access to internal and public networks without requiring expensive cabling. It is even possible to "daisy-chain" access points, providing an expanded interlocking web of network coverage over a greater distance.

Because of the networking-centric nature of the Wi-Fi solution, it tends not to be viable for interpersonal communication such as beaming a contact to the person standing next to you. But as a method of connecting your computer to networks without wires, Wi-Fi is an excellent option.

As with Bluetooth, a current downside to Wi-Fi is availability. Although it is becoming common to see laptops with built-in Wi-Fi radios, Wi-Fi is still not widely available as either a built-in or even add-on adapter for Palm handhelds. Symbol has for several years made an 802.11-enabled device for its target markets, and also has produced a wonderful CompactFlash card implementation. An issue that manufacturers are wrestling with is power consumption, which can be higher than that of infrared and Bluetooth. Another is form factor, as the requirements of Wi-Fi have not matched well with the peripheral interfaces available thus far on Palm devices such as SDIO.

Despite the current lack of PDA support, it is apparent that Wi-Fi is spreading fast, and one can imagine that it will soon become rather commonplace in office environments and public places. With that kind of availability, it is hard to believe that we will not see Palm devices with built-in Wi-Fi support in the near future (in fact as a harbinger, there are already PocketPC devices that are sold with Wi-Fi as an option).

Wide Area Wireless Options

This section covers some of the wireless communication options that provide PDA connectivity over very large distances, freeing the users from the need to be located in proximity to an access point or other source.

Wireless Data Networks

Over the past several years, several competing wireless data networks have sprung up, offering the equivalent of a dial-up networking account in ISP-like fashion, only without the need for a telephone jack.

A major downside to wireless data networks is coverage, which is very much dependent on location. Depending on your location, you may have excellent coverage, spotty coverage, or no coverage at all. Whether this matters will depend on how mobile you need to be. If your application will run primarily in high-density population areas and major cities, you will likely not have a problem.

If you cannot predict where your application will run, either because it will be used by people regardless of location, or because it will be used by people who are inherently traveling from place to place (here think of truck drivers or outside salespeople), the uncertainty of an available wireless connection can be a major barrier.

Coverage seems to be slowly improving, but it's easy to see that it will be a long time (if indeed ever) before wireless networks of this type will cover the entirety of the world's populated areas.

Another downside is the subscriber cost. Wireless networks incur a monthly subscriber account and a monthly charge, typically priced according to how much data is transmitted. Depending on how much data needs to be transmitted this cost can become significant, especially if you are equipping a number of PDAs with your application.

There is also the cost of a wireless PDA modem to consider. There is a wealth of modems to choose from, and that has driven down costs. Some providers even offer deals on modems if you subscribe to their wireless service. It's worthwhile to check around before buying.

Palm made a bold move several years ago by producing the first Palm OS-based handheld with a built-in wireless modem. The Palm VII, together with its successors the VIIx and i705, represent a good value for users who require fast and relatively inexpensive access to email and Internet services. Palm's Palm.net service is reliable, and when combined with Palm's proprietary WCA, or Web Clipping Architecture, can deliver Internet data to the palm of your hand quickly and reliably. Palm's device is also easy to connect with, it is said to be "always on." Thus there is no need to constantly connect and disconnect, which is very inconvenient if you've ever needed to connect in a hurry.

A final aspect of wireless data networks to consider is the communications speed. Although much has been made of the impending rollout of high-speed wireless data networks, the harsh reality is that today's network speed ranges from poor to abysmal. If you are lucky enough to be in a location with good coverage, you will still suffer speeds approximately an eighth of the typical landline dialup connection. For tiny bursts of data, or very carefully architected software, this can be sufficient. It does not however make for a pleasant experience downloading email or surfing the Web.

Cellular Phones

"Converged" devices, which combine a PDA with a cellular phones in single handheld device, are being rolled out in increasing numbers by both PDA manufacturers and traditional handset manufactures. Thus far, Palm OS "smartphone" devices have earned an early lead because of their form factor, usability, and availability. After a rough start with the original Qualcomm entry into this device category, several appealing next-generation devices have been produced from Kyocera, Handspring, and Samsung. They each represent different takes on the balancing act between a cell phone with PDA and a PDA with a cell phone.

Along with providing traditional voice capabilities, these cell phone hybrids also offer data connection capabilities through the various cellular networks. They are typically bundled with messaging functions, email, and a Web browser.

Unfortunately in my experience, although I like the idea of not carrying two devices (the cell phone and the PDA), the data networking functionality in these devices is lacking. The models available today are not always on, meaning an annoying dial-up connection that takes a minute or more is required each time you get online. Although that may not sound horrible, and indeed we have put up with that on our home PCs when we dial up to our ISP, in mobile situations where fast access to information is required, it can be frustrating.

As with wireless data networks, once you are connected, the transmission speed is extremely slow, and connections are frequently dropped. Thus wireless data through a PDA phone is more suited for messaging and short bursts of information (such as printing or fax) than it is for longer duration activities such as email or hot-sync.

As with all of these technologies, despite some of the glitches in the present implementation, the user experience will surely continue to improve over time.

The Developer's Perspective

The developer faces two challenges when faced with a requirement for wireless functionality. The first is choosing an appropriate target platform, including a device, peripherals, access point, and carrier (depending on the type). Note that if your application can function across multiple platforms (for example, an email client), you might need to account for a rather large list of compatible equipment.

One of the purposes of the previous section was to give you a sense of these targets, and help you evaluate the pros and cons of each as they relate to your requirements.

The second challenge is in choosing and learning the appropriate developer tools and information required to work on your chosen platform(s). This section matches the platforms with the available Palm OS APIs.

TCP/IP

TCP/IP is the most pervasive development option for Palm programmers in the wireless world. With the exception of infrared, all of the aforementioned communications scenarios can support the use of TCP/IP through the Net Library services offered by the Palm SDK (Net.lib is thoroughly covered in Chapter 26, "Using TCP/IP and Net.lib").

Using Net Library is advantageous because it does offer the largest range of target environments, and also because it is so familiar across platforms. Socket-based programming is a worldwide standard, and networking code can be ported rather easily from UNIX to Windows to Palm.

Use TCP/IP and Net Library where you need to provide access to the Internet, and to shared network resources such as file servers, application servers, and print servers. A downside to using TCP/IP can be the complexity of setting up a connection. Most users do not understand (nor do they want to learn) the complexities of DNS naming and IP addresses. They simply want to connect.

Web Clipping

As you'll see in Chapter 25, "Palm .NET and Web Clipping," Web clipping represents a unique way of optimizing Internet access across a slow wireless connection. It is proprietary to Palm, and arose as part of the Palm VII device platform. However with the installation of the appropriate software components, just about any wireless device can be set up to support Web clipping.

For the developer, Web clipping comes in two levels. At the browser level, Web clipping defines a standard for a flavor of HTML that is tuned for PDAs and slow connections. Thus you can create Web applications that are efficient to use on Web-clipping-enabled devices. A good example of this is the MapQuest clipping application that comes with the Palm.Net service. It is designed to provide driving directions quickly based upon minimal user input, and it works well.

Web clipping can be used at a lower level as a C programming library in the form of the Palm SDK's Internet Library, often referred to as INetLib. INetLib offers a high-level HTTP-like interface for developers to tightly integrate Web clipping functionality in their applications. INetLib has unfortunately had a checkered history, beginning life as an available but unsupported library from Palm, to legitimate status in the SDK. Along the way rumors in the developer community have persistently speculated that INetLib support will be dropped at some future point. If you are specifically targeting Palm.Net capability in your program, it's worth looking into, but otherwise it may be safer to avoid using it until its future is more certain.

Exchange Manager

Oriented more toward the short range of the spectrum, Exchange Manager is most familiar to developers in the form of OBEX, which drives the beaming of data over IR. Exchange Manager is of course capable of much richer functionality than simple beaming, and is also extensible via plug-ins and helpers. Exchange Manager is covered in Chapter 22.

Infrared Library

Specifically targeted at implementing the IrDA specification in the form of a developer's library, Infrared Library (IR Library) lets programmers code directly to the IR port that is standard on all Palm OS handhelds. If your applications need tighter control over infrared communications than what's offered in Exchange Manager, IR Library is worth looking into. IR Library is covered in depth in Chapter 23.

SMS and Telephony

Messaging and telephony are specific libraries that can help you incorporate voice and data exchange into your Palm application. With the telephony APIs, you can build a full-fledged phonebook and dialer application for use on the new wave of PDA smartphones. SMS and Telephony are rich topics in and of themselves and are not covered with their own chapters in this book. However the Palm SDK documentation contains discussions, function listings, and example programs that show you the way if these APIs are what you are looking for.

Summary

This completes the introduction to wireless communications options on Palm OS. I hope you will use this information to help you decide on a set of target devices, peripherals, access, and developer tools that are best suited to your needs.

Once you have decided which technology (or technologies) to explore further, I invite you to turn to the more detailed information offered in the next few chapters. Communications programming is not for the faint of heart, and I recommend you obtain all the resources and examples you can to understand the scope of effort and knowledge required to successfully add wireless communications support to your program.

Palm's Web site (`http://www.palmsource.com`) is a good additional resource and starting point, as is the Palm SDK documentation. You might also want to visit Web sites associated with the specific technologies you choose to target, such as `www.irda.org` or `www.bluetooth.com`.

25

Palm.Net and Web Clipping

IN THIS CHAPTER

The Palm VII wirelessly enabled handheld device was first demonstrated in 1998 and became commercially available the following year, just after the publication of the first edition of this book. Since that time, Palm has introduced two successors to the original model. The first was the Palm VIIx, which added more memory and some other upgraded capabilities. The most recent successor is the i705, which merges the original Palm VII capabilities into a more well-rounded wireless device and also adds expansion card support and other modern conveniences associated with PDAs.

Figure 25.1 shows the main application view for the Palm VII.

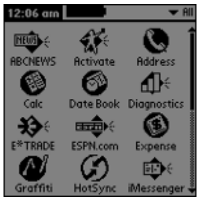


FIGURE 25.1 The main application list (icon view) for the Palm VII.

As noted in Chapter 24, “A Survey of Wireless Options for Palm Developers,” there are also now a wealth of alternative choices, from both Palm OS licensees and non-Palm devices, for the users who want to combine a PDA with wireless connectivity. Palm’s VII, VIIx, and i705, which collectively can be thought of as representing the “Web clipping model,” represent a unique spin on the idea of

PDA-based wireless connectivity, and in my opinion they remain among the most well-thought out solutions for both the customer and the developer.

This chapter introduces the developer to Web clipping and describes the design philosophy behind Palm's wireless product line. We investigate the infrastructure that enables the device to work in concert with the Internet to create a productive and enjoyable user experience. Finally, because this is a book about programming, we review the SDK changes and new developer-oriented features in store for you, the Palm programmer.

The Palm Difference

The feature that differentiates the Palm VII, the Palm VIIx, and the i705 from other Palm units, and other handheld computing devices in general, is the design philosophy embodied in the Palm OS that is most visibly expressed by the bundled application software. The Palm VII was not envisioned as a Palm III that is capable of browsing the Internet. Rather, the architects of the Palm VII acknowledged and studied the unique problems those before them in the mobile/wireless computing industry tried to solve. The Palm VII delivers a promising solution to those problems and creates a new platform for distributed, Internet-aware applications for the mobile professional.

To understand why Palm's wireless solution is so attractive, you need to understand why it has taken so long for wireless connectivity to become viable on a handheld device, and why it remains a challenging problem despite all the technological advances.

The Problem with Wireless Internet Connectivity

Although the Internet as a pervasive technology in our daily lives is a fairly recent phenomenon, most of you access the Internet via a relatively high-speed connection. The absolute minimum connection these days tends to be a 28.8Kbps dial-up, although most dial-up users enjoy up to 56Kbps speeds. Although on occasion frustratingly slow, this speed still provides a more-or-less acceptable Internet experience. Many people achieve faster connections via DSL, cable modems, ISDN, and other technologies that push the speed of the ISP connection to 64Kbps or higher. Still faster, the luckiest among us connect at T-1 speeds.

With such connection speeds the norm, Web content designers make assumptions accordingly about the amount of data that can be tolerably downloaded by users visiting their sites. As a result, today's Web pages are increasingly heavily formatted and are laden with large images, complex graphics, animations, Java applets, and more. All these fancy page elements need a big enough pipe to travel through in order to get to your Web browser before you become tired of waiting and move on to the next Web site.

What if tomorrow, someone walked up to your computer and replaced your Internet connection with a connection that was 8Kbps (less than one third as fast as the most awful dial-up connection we consider acceptable)? When you logged on to the Internet, how unpleasant would your Web-surfing experience become? How long before you logged off in frustration, tired of waiting 5, 10, or more minutes for a single page to download? Unfortunately, despite a great amount of hype about future wireless infrastructure, most wireless networks today run at speeds little faster than 8Kbps.

Of course, most Web sites are designed with another bias in mind. They assume that your browser display resolution is at minimum 640×480 pixels, if not higher. What if someone replaced that big, high-resolution monitor on your desk with a little miniature screen with less than 25 percent as much viewable area? How many of the Internet's beautifully crafted Web pages would make any kind of sense when viewed under these conditions?

If you are now thinking that surfing today's Web under these conditions wouldn't be much fun, you are beginning to understand the problem of wireless Internet connectivity on mobile computers. Many computer users (and even several prominent companies in the computer industry) have persisted in a PC-centric view of mobile computing, assuming that because people worked a certain way on their desktop or laptop computer, they would insist on being able to work the same way on a handheld device.

As it turns out, at least with the wireless technology generally available today, surfing the Web in the way we are used to is simply not possible on a handheld device using a wireless connection. Although several general-purpose Web browsers have been developed for the various handheld computing platforms, and no doubt more will be, the truth is that few people want to surf the Web from their cell phones or PDAs under the conditions described here.

Finally, aside from the technical issues, mobile computer users as a group are not likely to aimlessly spend time browsing the Web. They are on the go, looking to perform a few targeted actions with their Palms. They have a specific need for a piece of information, and they want to get at that information with a minimum of fuss. Navigating through pages of graphics and text to get what they need is inefficient and unacceptable.

Palm looked at the history of attempts to develop and deliver a handheld wireless solution, considered the problems of bandwidth and screen display, and came up with a solution. They call it *Web clipping*.

Enter Web Clipping and Palm.Net

The expression ”Web clipping” refers to the act of capturing only the information you need from the content available on the Web, just as you would clip a single article or advertisement from a magazine or newspaper. Palm wanted to apply this concept to the Web so that a Palm unit could access useful data from the Web efficiently, in terms of the quantity of data transferred.

As implemented by Palm, the Web clipping feature has a few key components, described in the following sections.

Palm.Net

Palm.Net is a broad description for a special network that acts as a “proxy server” for all communications between a Palm VII or i705 and the “real” Internet. Palm.Net provides security via encryption and also provides special compression and protocol translation—both inbound and outbound—to achieve the low-traffic levels required by the wireless data network.

It is important to understand that all traffic to and from a Web clipping device must run through Palm.Net, which automatically re-routes requests to the proper Internet host after the appropriate compression/decompression and security checks. If you want to use Web clipping, it is not possible for the unit to use wireless connectivity to directly communicate with hosts on or outside of the Internet.

In addition to the these services, Palm.Net also automatically strips bandwidth-unfriendly content from requested Web pages, including graphics, pictures, frames, Java, and scripting commands.

Content Providers

As noted, most Internet content is not created with the Palm’s miniature display in mind. Palm has done an admirable job of working with key content providers to help them design and create specially formatted content that is appropriate for display on the Palm. This content is similar to what’s normally provided but is targeted at the Palm user who wants quick and convenient information without wading through extraneous data.

For those content providers, Palm has provided a style guide that gives helpful suggestions and guidelines for designing and formatting content intended for the Palm user. The mantra is “less is more,” meaning that users will have a better experience (not to mention a lower subscriber cost) if you strip unnecessary information from your content.

Figure 25.2 illustrates this point via a screen from the sports content provider ESPN.

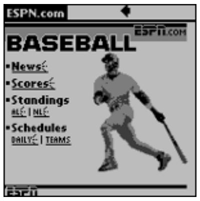


FIGURE 25.2 A sports junkie's dream: catching up on the latest baseball news—live as it happens!

Palm Query Applications

To prevent unnecessary re-transmission of content, Palm concluded that it was necessary to partition Web sites and applications so that the initial page was created, installed, and stored locally on the Palm device itself.

Essentially compressed HTML with special tag support, these locally stored pages are called Web Clipping Applications, or WCAs for short. (At the start, these were called Palm Query Applications, or PQAs, and in fact the "extension" for WCAs remains .PQA to this day.) Although almost any type of HTML page can be turned into a WCA, the most common example is that of a form for querying Web content.

WCAs are created via a Palm OS SDK tool, the Web Clipping Application Builder, which loads an HTML file and converts it to a .PQA file. This file is then loaded into the Palm just as a .PRC file is, using the Palm Installer tool.

Later in this chapter I will actually walk you through the process of creating a WCA, installing it, and running it on a Palm device.

The Developer Perspective

Either pre-installed on a wireless Palm device, or in some cases added to an existing device, are the components that enable a device to support Web clipping. Along with enabling the entire Web clipping architecture, these components expose new APIs as part of a library called "InetLib," which lets application developers take direct advantage of the wireless connectivity features.

Integration with Existing Applications

Before we get into the creation of a Web Clipping Application, I want to note that there are also ways to integrate Web content with "normal" Palm applications. The following capabilities are available to your application:

- Launch either a local Clipping Application or a remote URL from an application via a special launch code.
- Launch a Palm application from a tag in a downloaded HTML file. (This is conceptually similar to the tags that support local execution of Java or ActiveX applications.)
- Call a specific function within a Palm application from a tag.

iMessenger Integration

iMessenger is the e-mail client that is bundled with the Palm.net service. This built-in wireless messaging service also lets you add email-based messaging to your application. You can do this by embedding a special `mailto` tag in your HTML that will launch the iMessenger application. Code written in C can also launch iMessenger and directly set message attributes such as the subject, to, and cc fields.

What Is a Web Clipping Application?

A Web Clipping Application, or WCA, is not really an "application," at least not of the same type that you've been learning how to build throughout this book. A WCA is really an HTML (Hypertext Markup Language) file, with HTML being the *lingua franca* of the Internet. A WCA file is a set of HTML pages, links, and graphics stored locally on the Palm in compressed format as a special type of database. Just like any other type of database, a WCA database contains records. In the case of a Web Clipping Application, each record represents an HTML page.

You know that HTML by itself is not an executable program. On other systems, you rely on a Web browser to load and display HTML stored either locally or on a host system. On the Palm, you don't do general-purpose browsing; rather, you launch special-purpose WCAs to retrieve very specific bits of information.

When you launch a WCA, the Palm OS launches a hidden program called *Clipper*. It is Clipper's job to provide the interaction with the Palm.Net Web service. Clipper can be considered the Palm's "browser," and it is Clipper's job to load, decompress, and render your WCA as HTML. In case you were wondering, you can use Clipper from your own application to load either local or remote content by passing it a URL as a launch code.

A WCA is only one half of the picture, however. The rest of the content to be supported by your WCA actually resides on your Web site. This content is returned to the Palm as *clippings*. The Web server houses the remainder of the logic, returning results, or clippings, in the same manner as is done on millions of Web sites.

Effectively, your Palm application is “partitioned” into a client/server architecture. Of late, it has become fashionable to talk of thin clients, one expression of which is a browser-based application that presents a program’s user interface as HTML. A WCA is not that “thin” because at least the main page for your program must be stored on the Palm itself. Through Palm’s HTML extensions, it is possible for your client to be “fat” by calling into and thus relying on other code provided by you (or another vendor) in a Palm .PRC.

Typically, a WCA consists of an HTML form that uses standard CGI to submit the form’s contents to the Web server and receive the results as HTML. However, you are free to do anything you want with your local HTML, as long as it falls within the subset of HTML 3.2 supported by Palm. (See the Palm SDK documentation for a complete list of supported HTML.)

One more thing: It might appear that the quickest way to get your existing Web content onto the Palm VII is to create a simple front-end PQA that links to your existing Web site. This is a mistake. If you don’t recall why this is a bad idea, go back to the beginning of this chapter and review the reasons why general Web browsing is not well supported on handheld wireless platforms. You will need to re-evaluate the content returned by your Web site and possibly reconsider your Web site’s design.

Design Guidelines for PQAs

It is important to understand what works with PQAs and what either is not supported or will not work well.

Size of Content

The first restriction you should consider is that the entire viewable area for your content will be 153 pixels wide by 144 pixels high. You’ve obviously encountered this limitation throughout the book, but until now, it’s been a case of simply fitting everything you want into a small display area. With the Palm VII comes two additional factors: speed of retrieval and cost to the users. The more data you send over the wireless connection, the longer the users have to wait for your content. Perhaps more importantly, the users (at least with the subscriber plan currently in effect) are paying by the byte.

As a rule of thumb, you should not return more than 500 bytes on any individual page. (Palm recommends even fewer.) Also, keep in mind that if you use the default Palm font, you have about 11 lines of text to work with before you force the users to scroll.

HTML

A subset of HTML version 3.2 is supported by Clipper. In general, you want to stick with the simplest HTML you can possibly use to create an acceptable page on the Palm. This means limiting text effects such as bold, underline, font variations, and so on.

Beyond the> recommendations, some elements and extensions of HTML are just not supported. Palm.net will either strip out these elements or return an error to your users. Some of these are frames, nested tables, large or color-based images (see the next section), scripting extensions such as JavaScript, and dynamically loaded applets such as those created using Java or ActiveX. A full list of supported HTML elements is included in the Palm OS developer reference.

Images

Graphical images (both JPEG and GIF) are permitted on both local and remote content pages; however, they are limited to the same 153-pixel wide limitation. (Your users will receive an ugly error message if the Palm VII receives an image that is too big.) They also are limited to a maximum two pixels of color depth (in other words, simple grayscale).

Palm recommends that you strive to reduce the number of images returned by your Web site because they dramatically increase the number of transmitted bytes.

You can embed images in your local PQA by defining a `meta` tag in the `<head>` section of your HTML as follows:

```
<head>
<meta name="localicon" content="myimage.jpg">
</head>
```

In the body of your HTML, you then refer to the image in the same way you normally do:

```
<img src="myimage.jpg">
```

To get the Palm to understand and accept remote images, you must identify your HTML as “Palm-friendly” via the `meta` tag:

```
<meta name="PalmComputingPlatform" content="true">
```

If this tag is not present, remote images will not be downloaded and rendered. Also, your text will be truncated after 1,024 bytes. If you are in control of your content, however, there is no reason not to include this tag.

Hyperlinks

You define hyperlinks just as you do in "normal" HTML, using the `<A HREF="`*`linkname`*`">` tag. Things are slightly more complex in that you are permitted to bundle multiple HTML pages and images in your local PQA, so Palm provided a way to distinguish between remote and local links, as follows.

Following is a remote link:

```
<A HREF="http://www.bachmannsoftware.com/index.html">
     Click Here for Bachmann Software</A>
```

This is a local link:

```
<A HREF="file:myapp.pqa/index.html">Click Here for MyApp</A>
```

The `file:` designation clues Clipper into the fact that it must look in your .PQA file for the desired HTML.

When partitioning your application into local and remote content, you should carefully consider which portions of the content might change over time. Web-based content can obviously be changed as needed with no action from the users. If you need to change the content on a local PQA, however, you will have to tell the users to obtain a current version of your PQA and install it on their Palms.

If you have a significant amount of information that must be returned, you should structure your query so that it does not return all the information on the first query. Instead, allow the users to progressively ask for more levels of detail via a More link.

The Clipper History List

Mainstream Web browsers support a "history" list that offers a convenient way for users to quickly access recently visited sites.

The Palm VII and i705 have a somewhat limited history-tracking capability. The recent pages accessed are tracked, but on a per-PQA basis. The overall Palm VII environment does not track this; each PQA is responsible for designating what should be shown in the history list, and only the history for the currently running PQA is available to the users.

To enable history tracking, you add another `meta` tag to your HTML:

```
<meta name="HistoryListText" content="My Page">
```

`"My Page"` is replaced by the text you want to see in the history list to represent that page.

Customizing the User's Experience

As you might know, HTTP is a "stateless" protocol and as such does not offer the opportunity for either the host or the Web site to "remember" who the other person is from page request to page request. It is a challenge to customize Web content based on who you are from page to page and session to session. For example, a bookstore might find it useful to remember that you prefer mysteries so it can display ads and special offers that reflect that preference.

Some sites employ *cookies* to get around this problem. Cookies are small bits of information stored on the user's local computer. This information helps the Web site track the identification of the users and their preferences from session to session. The specific cookie mechanism is not supported, but Palm does support a special variable named `%deviceid`, which you can pass to the host via CGI. This is the Palm unit's physical device ID, and it can be used by the Web site to uniquely identify the users and match them to sets of preferences.

The Palm offers another way to customize the content returned to the users by supporting another variable called `%zipcode`. When passed via CGI, this variable contains the ZIP code of the nearest wireless base station to the Palm's location. PQAs that provide weather information, traffic reports, and such can support special features that "know" who and where you are.

Launching Other Applications

You can employ special tags to launch and interact with other applications resident on the Palm. For example, the following line launches another PQA-named helper:

```
File:helper.pqa
```

The following line launches the Address Book application:

```
Palm:addr.appl
```

The following line invokes the iMessenger application with the address specified:

```
mailto:glenn@bachmannsoftware.com
```

Steps Involved in Creating a PQA

Creating your first PQA is actually simple. You need only a couple of tools, including:

- **HTML editor**—A plain-text editor is best. You definitely do not want to litter your HTML with vendor-specific tags. You also want to have tight control over the HTML so that the minimum number of bytes is transferred. Keep in mind that you will also need to manually insert Palm `meta` tags.
- **Query Application Builder**—This is provided in the Palm SDK and converts one or more HTML files along with linked images to a PQA.

As I said before, the Builder will convert not only your “main” HTML file, but will also convert “local” HTML files referred to by the main HTML page’s links. All the pages, images, and links are stored in the final PQA.

A Sample PQA: *Palm Programming* Information Kiosk

The current documentation from Palm includes some basic examples of how to create a “hello world” PQA. I take that one step further and create something that might be considered more useful.

I will show you how to create a PQA that is an information resource for this book. The local portion of the application will be a single main page that provides links to information on *Palm Programming*, including an overview, a table of contents, and (most importantly) information about me! Theoretically, the remote portion could be hosted on Sams Publishing’s Web site and provide a dynamically updated portal into *Palm Programming*’s table of contents, corrections and revisions, other sources of information, and so on. You can easily imagine extensions to this kiosk to let visitors purchase the book and even submit comments, questions, and reviews to the publisher.

The first step in creating this PQA is to develop the HTML. Look at the HTML for the main (local) page in Listing 25.1.

LISTING 25.1 HTML Syntax for the Main Page

```
<html>
<head>
<meta name="PalmComputingPlatform" content="true">
<meta name="localicon" content="sams_sign_left.gif">
<title>Palm Programming</title>
</head>
<body>
<img src="sams_sign_left.gif">
<BR>
Presenting <B>Palm Programming</B> by Glenn Bachmann
<BR>
```

LISTING 25.1 Continued

```
<a href="http://www.bachmannsoftware.com/pp_overview.htm">Overview</a>
<BR>
<a href="http://www.bachmannsoftware.com/pp_toc.htm">Table of
 Contents</a>
<BR>
<a href="http://www.bachmannsoftware.com/pp_me.htm">About
 Glenn Bachmann</a>
<BR>
<BR>
Glenn Bachmann's Palm Programming is a straightforward tutorial that
teaches developers how to create fully functioning Palm applications
using the CodeWarrior development environment.
<BR>

</body>
</html>
```

If you are familiar with HTML, you know this looks like the source for a normal page. Only two elements here are new. As I described earlier in the chapter, the following line is a `meta` tag that must be present for the Palm to process images correctly:

```
<meta name="PalmComputingPlatform" content="true">
```

The following line was also covered earlier, and you will recall that it identifies an image that is embedded in the PQA itself:

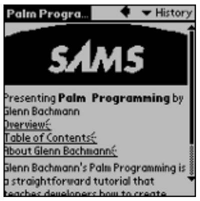
```
<meta name="localicon" content="sun.gif">
```

Without this tag, Clipper would not be able to properly handle the main page's icon (which is a standard Sams Publishing image).

There are three remote links in the HTML, and all three reference HTML files on the public Internet.

How do I create a PQA from this file? Enter the Query Application Builder! Launch QAB.EXE, and you will be greeted by a simple utility, shown in Figure 25.3.

From the File menu, choose Open Index, and locate your main HTML file (often called index.html). The Builder parses your HTML, identifies any local links (not remote links), and automatically adds them to the Builder main window. In this case, it located and loaded the Sams .GIF image.



Presenting **Palm Programming** by Glenn Bachmann
Overview
Table of Contents
About Glenn Bachmann
Glenn Bachmann's Palm Programming is a straightforward tutorial that

FIGURE 25.3 The Query Application Builder main screen with the HTML and image loaded.

Choose Build PQA from the File menu, give your PQA a filename (such as Palm Programming.pqa), and click the Build button. You now have a Palm Query Application ready to load onto the Palm VII.

The other three HTML files are all similar, so look at Listing 25.2 to examine the only one that has a small twist: the About Glenn Bachmann page.

LISTING 25.2 HTML Syntax for the About the Author Page

```
<html>
<head>
<meta name="PalmComputingPlatform" content="true">
<meta name="HistoryListText" content="About Glenn Bachmann">
<title>Palm Programming - About Glenn Bachmann</title>
</head>
<body>
<BR>
<B>Glenn Bachmann</B> is president of <B>Bachmann Software and
Services, LLC</B>.
<BR><BR>
Founded in 1993, Bachmann Software and Services provides
software products and professional software development
services for the mobile computing and wireless communications industries.
<BR>Bachmann Software has developed products for the industry's
leading corporations and is the creator of <B>PrintBoy</B>
and <B>FilePoint</B> for the Palm Computing Platform.
<BR><BR>
<A HREF=mailto:glenn@bachmannsoftware.com>Send email to the author</A>
</body>
</html>
```

Note two lines of interest. The following line is a `meta` tag that identifies this page as one that belongs in the History list and gives it the proper text to display:

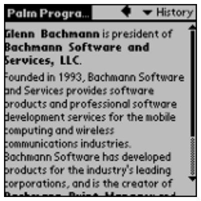
```
<meta name="HistoryListText" content="About Glenn Bachmann">
```

The following is a link to the built-in `mailto` tag, and when clicked, it will launch the iMessenger application on the Palm VII, telling it to open a new message with my e-mail address in the `to` field:

```
<A HREF=mailto:glenn@bachmannsoftware.com>Send email to the author</A>
```

Installing the *Palm Programming* kiosk is a two-step process. First, use the Palm Install tool to load your PQA during the next hot-sync operation, just as you do with a .PRC file. Second, store the remote HTML files in the location referenced by the PQA's remote hyperlinks.

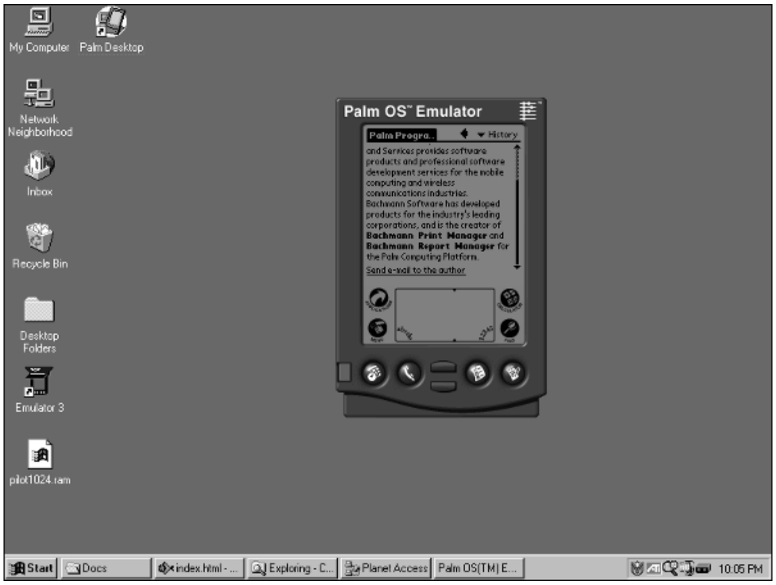
Figures 25.4 and 25.5 show the main *Palm Programming* kiosk page as well as the About Glenn Bachmann page.



Glenn Bachmann is president of **Bachmann Software and Services, LLC.**

Founded in 1993, Bachmann Software and Services provides software products and professional software development services for the mobile computing and wireless communications industries. Bachmann Software has developed products for the industry's leading corporations, and is the creator of

FIGURE 25.4 The *Palm Programming* main PQA page.



and Services provides software products and professional software development services for the mobile computing and wireless communications industries. Bachmann Software has developed products for the industry's leading corporations, and is the creator of **Bachmann Print Manager** and **Bachmann Report Manager** for the Palm Computing Platform.

Send e-mail to the author

FIGURE 25.5 That's me!

Summary

With remarkably little effort, this chapter shows you how you can create a useful application that uses the built-in wireless Internet capabilities of the Palm VII. Although much more sophisticated applications are possible, of course, this chapter demonstrated what you can do with only a little knowledge of HTML and some guidelines on PQA design.

26

Using TCP/IP and Net.lib

IN THIS CHAPTER

There is perhaps no more popular a computing topic than networking. Networking computers and users together has been the focus of the industry for years, and indeed today it is difficult to find a computer that is not connected in some way to other computers through a network. Now that computing has gone mobile, the quest for connectivity rages on.

Although a basic Palm device that you buy at your local store shows no obvious signs of being capable of networking, the Palm OS platform is by no means limited to working on a single device as an island. In fact, the Palm platform bridges the gap between standalone personal digital assistant functionality and Internet citizenship. When equipped with the necessary physical hardware and connections, a Palm-powered device is capable of interacting on both the public Internet and private networks alike, supporting applications such as email, inventory control, and general Web browsing.

This chapter examines the networking capabilities of the Palm OS via the Network Library programming interface. The content of this chapter covers network programming techniques, Palm OS– specific APIs, and connectivity basics on the platform; it assumes that you are familiar with basic inter-networking concepts.

The Palm OS Network Stack

Before diving into network-enabled applications and the nuances of the Palm OS's networking functions, let's take a high-level look at the Palm OS network stack. Figure 26.1 outlines the multiple layers contributing to a networked Palm OS application.

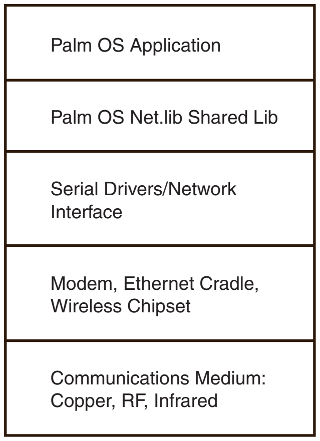

Palm OS Application
Palm OS Net.lib Shared Lib
Serial Drivers/Network Interface
Modem, Ethernet Cradle, Wireless Chipset
Communications Medium: Copper, RF, Infrared

FIGURE 26.1 The Palm OS network stack.

In Figure 26.1, the top layer marked Palm OS Application represents user applications. These are the programs written by Palm OS developers, the target audience of this book. It is in this layer that programs such as Web browsers, email programs, FTP applications, and others exist.

These user applications rely upon a set of application programming interfaces provided in the next layer, known as Net.lib. Net.lib first appeared in Palm OS 3.0, and is a Palm OS shared library explicitly loaded by an application for the purpose of interacting with the "network."

Moving down the stack, the serial drivers and network interfaces are software components implementing protocols such as Serial Line Internet Protocol (SLIP) and Point-to-Point Protocol (PPP). These protocols carry network messages from one place to another. This layer allows the abstracted network functions found in Net.lib and physical devices such as modems to communicate. Physical devices themselves communicate by a variety of means, including wired connections, CDMA cell service, infrared, and radio-frequency (RF) technologies such as 802.11x and Bluetooth.

The layered approach to networking has been leveraged on most computing platforms. The advantage to layering services and functions is the ability to swap out various pieces without needing to re-write the entire stack.

For example, an e-mail program needs only to effectively utilize the resources available in the Net.lib library—it is not concerned with which underlying protocol or connecting device is employed. The program works just the same (speed notwithstanding) whether the underlying physical connection is a landline modem or a CDMA-equipped cell phone.

Connectivity Options and the Preferences Panel

Establishing a physical network connection on a Palm OS device typically starts with the System Preferences application.

The Preferences application provides an interface for the users to configure various system level settings, which are grouped and arranged in a series of panels. The Network Preference panel provides a mechanism to define and configure network connectivity. A given device may utilize any number of network connections, known as *services*. Each service has a set of configuration data providing the necessary information for the device to establish a network connection.

Here is an example of the information managed for a given service:

- Service Name
- User Name
- Password
- Connection Type (PPP, SLIP, and so on)
- IP Address and DNS Information
- Login Script commands
- Phone Number (version 3.0)
- Connection (version 3.5 and later)

The Connection Manager, introduced in Palm OS 3.5, provides a more flexible mechanism for defining and re-using physical device settings. A service uses a connection, whereas a connection uses a particular technology for connectivity.

Figure 26.2 shows a simplified representation of the relationship between Network Preference Service entries and connections.

Services may be manipulated and managed through the Preferences panel or via special Net.lib API functions. Connections are similarly controlled via the Preferences panel or via the Connection Manager functions.

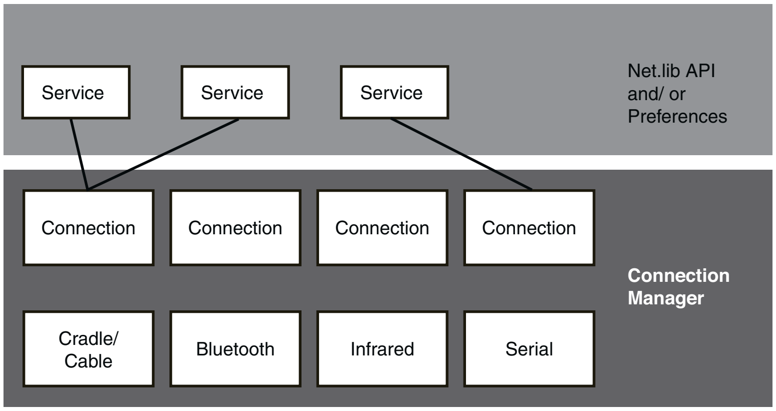


FIGURE 26.2 Net.lib and the Connection Manager.

The Net.lib Development Environment

A complete network application development environment consists of several components.

In addition to the core elements such as a Palm OS device and a compiler, a network application needs a peer application with which to communicate. For example, an HTTP client (coincidentally the first sample application discussed in this chapter) requires a Web server in order to test proper functionality. The second example application—a daytime protocol UDP-based client—requires a daytime server to query.

Testing and debugging a networking Palm application can be a challenge. Although some testing on a physical connected device is possible (and necessary!), the Palm OS Emulator (POSE) is an invaluable tool when developing communications applications. The emulator has an option for directing TCP/IP calls through the host computer. This means that networking calls made by your application inside of POSE are automatically forward to the TCP/IP stack on the host computer (your PC).

This feature permits an accelerated edit, compile, and run cycle because it's much quicker to re-load an application into the emulator than to hot-sync the application to the handheld device and then re-establish a network connection directly from the device. Additionally, using the emulator provides a means to rapidly and conveniently test the application against different versions of the Palm OS by simply loading a different ROM into the emulator (for a full discussion of the Palm OS Emulator, see Chapter 3, "Rapid Development Using POSE, the Palm OS Emulator").

Although the emulator-testing approach is sufficient for early development cycles, it is imperative that an application be tested on real hardware with real network connectivity. Admittedly this can be painfully slow at times, but it is a necessary step. For one, your PC typically will have a dramatically faster connection than a Palm device will, so you need to be sure not to overlook the impact of a slow wireless connection when downloading data as compared to running the application on the emulator at LAN speeds. Always keep the target environment in focus during the development cycle.

Using the Network Library

This section covers how to use the functions in Net.lib, including how to load and open the library, data structures, and APIs used in network programming. It also discusses establishing and using sockets, and sending and receiving data.

Loading and Opening Net.lib

Networking support in the Palm OS is optional; not every application requires networking, so the library must be explicitly loaded by an application before making a network connection.

The network library, hereafter simply referred to as Net.lib, is a shared library initialized in the same manner as other shared libraries. Here is a code snippet from one of the sample applications demonstrating the initialization of Net.lib:

```
Err err;
UInt16 refNum,neterr;

// attempt to find the Network Library
err = SysLibFind ("Net.lib", &refNum);

// was the library found?
if (!err)
{
   // attempt to open the library
   err = NetLibOpen(refNum,&neterr);
   if (err)
   {
      StrPrintF(buf,"Failed to open Network library. Error Code is [%d]",
               neterr);
      ErrBox(buf);
      return;
   }
}
else
{
   ErrBox("Failed to find Net.lib, cannot continue");
   return;
}
```

The `SysLibFind()` function populates the `refNum` variable with the library's identifier. All subsequent calls to the Net.lib functions require this parameter. Once the library has been found, it must be opened with the `NetLibOpen()` function prior to taking any other network actions.

Closing the Library

When an application has concluded with network interactions, the `NetLibClose()` is invoked, releasing resources. The function takes two parameters. The first parameter is the library reference, stored in the `refNum` variable. The second parameter governs whether the library should shut down immediately or permit connections to linger. Setting the parameter to `true` causes the library to shut down immediately and drop the physical network connection.

If another `NetLibOpen()` call is anticipated within your application's session, the *immediate* parameter should be set to `false`, allowing the underlying physical connection to remain intact. This improves performance. Good programming practice dictates that `NetLibClose()` must be called once for each `NetLibOpen()` call.

Cost and performance are often at odds. It may be desirable to persist a network connection to improve latency for subsequent operations; however, wireless connections are not typically free and airtime charges accumulate quickly. Balancing performance and cost is an important factor in wireless application design.

Data Structures and API Conventions

Applications utilizing Net.lib include the Palm OS SDK header file, NetMgr.h. This file contains data type definitions, constants, and function prototypes required for network programming. The following section introduces some of the more commonly encountered data types.

A `NetSocketRef` represents a network connection, equivalent to a *socket* on other computing platforms.

```
// Socket refnum
typedef    Int16             NetSocketRef;
```

Network addresses have many representations. An IP address is stored in a NetIPAddr.

```
// Type used to hold Internet addresses
typedef    UInt32        NetIPAddr;        // a 32-bit IP address.
```

The NetSocketAddrINType represents a complete Internet socket address, including address family, port number, and 32-bit IP address.

```
typedef struct NetSocketAddrINType {
   Int16      family;      // Address family in HBO (Host UInt8 Order)
   UInt16      port;       // the UDP port in NBO (Network UInt8 Order)
   NetIPAddr   addr;       // IP address in NBO (Network UInt8 Order)
   } NetSocketAddrINType;
```

The NetSocketAddrType represents a generic socket address.

```
typedef struct  NetSocketAddrType {
   Int16         family;               // Address family
   UInt8         data[14];            // 14 bytes of address
   } NetSocketAddrType;
```

At times, it is necessary to perform a lookup operation, transforming an IP address into an Internet hostname, or conversely, finding an IP address from an Internet hostname. The structure used in these operations is the NetHostInfoType.

```
typedef struct  {
   Char *      nameP;         // official name of host
   Char **     nameAliasesP;  // array of alias's for the name
   UInt16      addrType;      // address type of return addresses
   UInt16      addrLen;      // the length, in bytes, of the addresse
                   // Note this denotes length of an address
                  // not # of addresses.
   UInt8 **   addrListP;      // array of ptrs to addresses
} NetHostInfoType, *NetHostInfoPtr;
```

Socket multiplexing is accomplished with the assistance of NetFDSetType, which holds descriptors for multiple sockets.

```
typedef UInt32      NetFDSetType;
```

Net.lib provides a comprehensive offering of networking functionality; however, there are some platform considerations important to note:

- Net.lib is limited to four socket connections.
- Most Net.lib functions include two parameters not found on other platforms:

  A timeout value in system ticks

  An address of the variable to receive the error code
- The majority of Palm OS networking applications are client oriented. Palm OS devices often operate behind some sort of gateway service providing Network Address Translation, or NAT. Publicly accessible IP addresses required for server application binding are hard to obtain.

TCP Sockets—Streams

The most common network connection type is the TCP or stream socket. TCP stands for Transmission Control Protocol. The TCP protocol is layered over the Internet Protocol, or IP, providing guaranteed, sequenced delivery of information. As data is sent across the network, it is broken into relatively small chunks known as *packets*.

Packets may arrive at their destination in any order; however, they contain a sequence number, allowing the receiving TCP layer to re-assemble the data into its original form. Common protocols such as HTTP, SMTP, POP3, and Telnet rely upon TCP stream sockets. TCP/stream sockets are characterized by the fact that they require a *connection* in order to exchange data. No data can be sent or received without a connection.

The sample application `http_palm` included with this chapter is the basis for the TCP examples. The next few sections discuss select portions of the sample application; a full description of the application is presented at the end of the chapter.

The first step is to open a socket:

```
NetSocketRef sock;
UInt16 neterr;

sock = NetLibSocketOpen(refNum,netSocketAddrINET,
                        netSocketTypeStream,
                        netSocketProtoIPTCP,
                        10 * SysTicksPerSecond(),
                        &neterr);
if (sock == -1)
{
     // handle error contained in neterr
}
```

The parameters to the `NetLibSocketOpen` call inform Net.lib of the type of socket requested. In this example, the address protocol is of type `NetSocketAddrINET`, the socket type is `Stream`, and the protocol is `TCP`. The function has 10 seconds to execute and any errors are reported in the `neterr` parameter.

Domain Lookup and Connection

Once a socket is opened, it must be connected to a host. The following code demonstrates looking up an address for an Internet host and establishing a connection to the host once it is located:

```
static NetSocketAddrINType sa;
static NetHostInfoBufType   hostbuf;
static NetHostInfoType *   pHost = NULL;
UInt16 neterr;
static Char buf[200];
Int 16 port = 80;

sa.family = netSocketAddrINET;
pHost = NetLibGetHostByName(refNum,"www.palm-communications.com",
                            &hostbuf,SysTicksPerSecond() * 3,&neterr);
if (pHost == NULL)
{
   StrPrintF(buf,"Failed to find host [%s]",host);
   ErrBox(buf);
}
else
{
   sa.addr = hostbuf.address[0];
   sa.port = NetHToNS(port);
   retval = NetLibSocketConnect(refNum, sock,
                               (NetSocketAddrType *) &sa,sizeof(sa),
                                3 * SysTicksPerSecond(),&neterr);
   if (retval == -1)
   {
      StrPrintF(buf,"Unabled to Connect, ErrorCode [%d]",neterr);
      ErrBox(buf);
   }
   else
   {
      // good connection, let's use it
   }
}
```

Points to note in this code:

- The target host is `www.palm-communications.com`. Substitute any domain you like.
- The `NetLibGetHostByName` function returns a pointer to a `NetHostInfoBufType`, containing the network address of the host. Comparing this pointer to `NULL` indicates the success or failure of the call. If successful, the return value points to the `NetHostInfoBufType` passed to the function. In this example, `pHost` points to the address of the `host` variable if the Internet hostname is found. If the host is not found, `pHost` equates to `NULL`.
- The address in the returned host buffer structure is utilized in the subsequent `NetLibSocketConnect` call.
- Both the host lookup and the socket connection calls have three seconds to complete.
- The `SysTicksPerSecond` function is employed for timing purposes. Supplying the timeout parameter in this fashion facilitates portable code across multiple Palm OS devices where the clock speed and processor type may vary.
- Any errors encountered in these functions are passed via the `neterr` parameter.
- The `NetHToNS` is actually a macro. Note that it does not require the typical `refNum` parameter.
- The port number is set to 80; this is the well-known port for HTTP servers. Net.lib provides a convenience function named `NetLibGetServByName`, useful for looking up port numbers for common applications such as Telnet, SMTP, and so on. It is good practice to handle port numbers as configurable parameters, rather than hard-coded values. It is not uncommon for a service to be available on a non-standard port. For example, some Web sites make their services available at ports other than 80. The http_palm sample application allows connections to Web servers at any port number.

Sending and Receiving Over a Socket Connection

When the socket has established a connection to the remote host, a virtual circuit is in place between the two end-points. Bi-directional communication may take place. From the application's point of view, the connection is *full-duplex,* meaning that sending and receiving can occur simultaneously. The underlying transport may support only half-duplex communications, but Net.lib's buffering shields the application from this complexity.

Sending data over a socket is achieved with the `NetLibSend()` function. The function is essentially a request to the Net.lib networking layer to transfer data on behalf of the application. Here are the parameters required for the proper usage of the function:

```
Int16 NetLibSend (
UInt16 libRefNum,    // refNum passed to all Net.lib functions
NetSocketRef socket, // socket identifier
void* bufP,          // data to send
UInt16 bufLen,       // # of bytes to send
UInt16 flags,        // special i/o flags
void* toAddrP,       // target address (NULL for TCP sockets)
UInt16 toLen,       // length of address (0 for TCP sockets)
Int32 timeout,       // time in system ticks before timeout
Err* errP           // holds any error encountered
)
```

Here is an example of sending an HTTP request to a Web server:

```
Char lbuf[100];

StrPrintF(lbuf,"GET %s HTTP/1.1\r\n",uri);
rc = NetLibSend(refNum,sock,lbuf,StrLen(lbuf),0,NULL,0,
                SysTicksPerSecond() * 1,&neterr);
if (rc == -1)
{
   StrPrintF(buf,"Error Sending [%d]",neterr);
   ErrBox(buf);
   return -1;
}
StrPrintF(lbuf,"Host: %s\r\n\r\n",host);
rc = NetLibSend(refNum,sock,lbuf,StrLen(lbuf),0,NULL,0,
                SysTicksPerSecond() * 1,&neterr);
if (rc == -1)
{
   StrPrintF(buf,"Error Sending [%d]",neterr);
   ErrBox(buf);
   return -1;
}
```

This code sends a valid HTTP request header, requesting the value in the `uri` variable. Note the implementation of the HTTP/1.1 protocol; it mandates the provision of the `Host` header variable.

If any errors are encountered, the function returns –1 and the `neterr` variable contains the error code. If the send is successful, the return value is the number of bytes actually transmitted.

It is important to note that the call may be successful, yet not transfer all of the data requested. Therefore, it may be necessary to call `NetLibSend` in a loop, checking that all of the data has been sent. For the sake of simplicity and clarity, the sample assumes all data is sent successfully in the first request and the checking is omitted.

Receiving data is performed via the `NetLibReceive` function. The function reads any data available for the socket and then returns a value. Like the `NetLibSend` function, a return value of –1 indicates an error. A return value of 0 (zero) is an indicator that the remote side of the virtual circuit has closed the connection. Here is the list of parameters required for the `NetLibReceive` function:

```
Int16 NetLibReceive (
UInt16 libRefNum,      // refNum passed to all Net.lib functions
NetSocketRef socket,   // socket identifier
void* bufP,            // buffer to hold received data
UInt16 bufLen,         // size of buffer
UInt16 flags,          // special i/o flags
void* fromAddrP,       // address of sender (UDP only)
UInt16 * fromLenP,     // size of address
Int32 timeout,         // time in system ticks before timeout
Err* errP             // holds any error encountered
);
```

The sample application reads data one byte at a time, parsing the data as it reads. Here is the code required to read a single byte from a TCP stream socket:

```
Char c;
NetLibReceive(refNum,sock,&c,1,0,NULL,NULL,SysTicksPerSecond() * 1,&neterr);
```

The ability to parse incoming data is a necessary skill in TCP programming. Data may arrive to the application in one large burst, or slowly over an extended period of time. Applications should make no assumptions about how quickly data will be available on a socket. Similarly, an application needs to be mindful of the fact that it may be impossible to send data, as the underlying network buffers may be full. The next section (which discusses the `NetLibSelect` function) addresses these timing and control considerations.

Servicing Multiple Connections—`NetLibSelect`

The core resource available to the network programmer for providing flexible, responsive applications is the `NetLibSelect` function. This function permits an application to monitor activity on multiple sockets. Its basic role is to detect activity or availability of socket resources. It can monitor the following conditions of sockets:

- *Ready to receive,* which means there is data available to be read by the application.
- *Ready to send,* which means there is sufficient buffer space to write data to the socket.
- *Exception handling.* If an error occurs on a connected socket, the `NetLibSelect` function can detect the problem, and thereby inform the application to take action.

`NetLibSelect` operates on `NetFDSetType` structures. The function compares the values contained in each of three `NetFDSetType` parameters to internal flags found deep in the core of Net.lib. The return value of the function indicates the number of sockets requiring attention.

There is a `NetFDSetType` parameter for reading-, writing-, and exception-processing. It is possible to monitor only one of the three conditions; however, a minimum of one condition must be monitored in each call to the function. Special macros assist the network programmer in manipulating these structures.

```
NetFDSetType readFDs,writeFDs;

netFDZero(&readFDs);      // initialize the variables
netFDZero(&writeFDs);
netFDSet(sock,&readFDs);   // Indicate an interest in read availability
netFDSet(sock,&writeFDs);  // Indicate an interest in write availability
```

Macros also exist to clear and test these structures:

```
NetFDClr(sock,&readFDs);   // clear socket from test for reading
NetFDIsSet(sock,&readFDs); // is this socket ready for reading?
```

Here is the function prototype:

```
Int16 NetLibSelect (
UInt16 libRefnum      // library identifier
UInt16 width         // # of sockets to test in each descriptor set
NetFDSetType* readFDs      // set of sockets to test for reading
NetFDSetType* writeFDs     // set of sockets to test for writing
NetFDSetType* exceptFDs   // set of sockets to test for
                          // exceptions (error conditions)
Int32 timeout         // how long to wait for a socket to become signaled
Err* errP             // error code
) ;
```

`NetLibSelect` returns the following:

Return Code	Description
0	Timeout
-1	Error condition; examine `errP` for specific error code
X	Number of sockets in signaled condition

A network application must accommodate a variety of *participants* and activities during the course of the running a program. In addition to servicing established network connections, the program must also service the user interface. This means that the application needs to *multiplex* between any available network connections and the user interface. The user interface is simply treated as a special socket value, `sysFileDescStdIn`.

```
NetFDSet(sysFileDescStdIn,&readFDs);
```

Understanding and implementing the `NetLibSelect` function is a key ingredient to a well-behaving network application. Here is a snippet from `http_palm` when attempting to read a line from the Web server:

```
Int16 GetHTTPHeaderLine(NetSocketRef s, Char * buf)
{
static NetFDSetType readFDs;
Int16 n;
UInt16 timeoutcount=0;
UInt32 rc=0;
Char c;
Int16 index=-1;


   while (1)
   {
      netFDZero(&readFDs);
      netFDSet(s,&readFDs);
      switch (n = NetLibSelect(refNum,netFDSetSize,&readFDs,NULL,NULL,
                               SysTicksPerSecond() * 1,&neterr))
      {
         case -1:
            // handle error!
            return -1;
         case 0:
            // timeout
            MsgBox("timeout");
            return -1;
            break;
         default:
            if (netFDIsSet(s,&readFDs))
            {
               switch  (NetLibReceive(refNum,s,&c,1,0,NULL,NULL,
                                       SysTicksPerSecond() * 1,&neterr))
               {
                  case 0:
                  case -1:
                     // done....
                     return 999;
                  case 1:
                     // check for newline
                     switch (c)
                     {
                        case 0x0d:  // eat this
                           break;
                        case 0x0a:
                           // term this line
                           buf[++index] = 0x00;
                           MsgBox(buf);
                           return 0;
                        default:
                           buf[++index] = c;
                           break;
                     }
                     break;
               } //switch
            }
            else
               MsgBox("???");
            break;
      } // switch
   } // while (forever)
   return 999;
}
```

This discussion of `NetLibSelect` has thus far been dominated by examples of connected sockets and client-oriented connections. The next section briefly discusses server sockets.

Server Mode and `NetLibSelect`

`NetLibSelect` is also useful before a socket connection is established. Here is the recipe for an application operating in server mode—where it is waiting for connections initiated by other clients:

1. Application calls `NetLibSocketBind()` to associate itself with the local network address and a port number. For example, a Web server binds to port 80 by default.
2. Application calls `NetLibSocketListen()`, informing Net.lib that it is interested in handling any inbound connections to the bound socket.
3. Application calls `NetLibSelect()` with the bound socket set to monitor reading.
4. When `NetLibSelect` indicates that the socket is ready to read, the application invokes the `NetLibSocketAccept()` function to accept the inbound connection.
5. The application resumes calls to `NetLibSelect`, awaiting additional connections.

UDP Sockets—Datagrams

Not all network communications require a "connection." In fact, the computational and bandwidth expense of establishing TCP connections is not always the most efficient, because many applications send very short, single-packet messages.

An alternative approach to TCP is the User Datagram Protocol, or UDP. UDP is a connectionless, best-effort protocol. UDP does not offer the same level of assurance that data will arrive at its intended destination; however, it has less overhead than TCP connection-based communications. Common examples of the UDP protocol include domain name lookups and the Simple Network Management Protocol (SNMP).

The ability to parse an incoming stream was an essential element to TCP programming; however, the UDP datagram is a chunk of data requiring no special parsing. Once the packet arrives, all of the data in the packet is available to the receiver. Some UDP-based applications send data across multiple packets and must do their own stitching together and sequencing, but for simple applications, UDP is a great alternative to the stream-oriented programming required with TCP.

Creating UDP Sockets

Again, the NetLibSocketOpen function is used in the creation of a UDP socket; however, the parameters vary from the TCP sample discussed earlier. Here is the code necessary to create a UDP socket:

```
sock = NetLibSocketOpen(refNum,netSocketAddrINET,
                        netSocketTypeDatagram,netSocketProtoIPUDP,
                        SysTicksPerSecond() * 10,&neterr);
```

The parameters to the NetLibSocketOpen call inform Net.lib of the type of socket requested. In this example, the address protocol is of type NetSocketAddrINET, the socket type is Datagram, and the protocol is UDP. The function has 10 seconds to execute and any errors are reported in the neterr parameter.

Sending and Receiving

Hostname lookups take place in the same manner as described earlier in the TCP example. The key difference is that there is no connect step. The address is used directly in the NetLibSend function call:

```
MemSet(&sa,sizeof(sa),0x00);
sa.family = netSocketAddrINET;
pHost = NetLibGetHostByName(refNum,host,&hostbuf,
                             SysTicksPerSecond() * 3,&neterr);
if (pHost == NULL)
{
   StrPrintF(buf,"Failed to find host [%s]",host);
   ErrBox(buf);
   NetLibSocketClose(refNum,sock,SysTicksPerSecond() * 2,&neterr);
   NetLibClose(refNum,true);
   return;
}
else
{
   sa.addr = hostbuf.address[0];
     sa.port = NetHToNS(13);
   // send request
   rc = NetLibSend(refNum,sock,lbuf,0,0,&sa,sizeof(sa),
                   SysTicksPerSecond() * 1,&neterr);
   if (rc == -1)
   {
      StrPrintF(buf,"Error Sending [%d]",neterr);
      ErrBox(buf);
      NetLibSocketClose(refNum,sock,SysTicksPerSecond() * 2,&neterr);
      NetLibClose(refNum,true);
      return;
   }
}
```

Note that this call to NetLibSend includes the address of the target host. Recall that the TCP sample provided a NULL value for this parameter. This example actually sends a zero length packet to port 13 of the remote host. This is the step required for querying a daytime server. The next step is to read the response from the server:

```
// read response
addrlen = sizeof(sa);
rc = NetLibReceive(refNum,sock,lbuf,sizeof(lbuf),0,&sa,&addrlen,
                   SysTicksPerSecond() * 1,&neterr);
if (rc)
{
   // display response
   MsgBox(lbuf);
}
else
{
   ErrBox("Failed to retrieve DayTime information");
}
```

Again, the call to NetLibReceive for UDP sockets requires the fromAddress parameter to be populated.

Berkeley Sockets Compatibility

Because the majority of networked applications are available in UNIX/Berkeley sockets form, Palm OS provides a compatibility mechanism for easing the job of porting networking applications to the Palm OS platform. The compatibility layer is a collection of constants, enumerations, macros, and global variables facilitating the mapping between Berkeley APIs and Net.lib APIs. For example, the send function maps to NetLibSend. The gethostbyname function is actually a macro mapping to the NetLibGetHostByName function. Recall that a characteristic of Net.lib functions is the presence of timeout and error parameters not found in Berkeley functions. These parameters are satisfied by the use of globally defined variables, forcing a single timeout value for all operations.

There are a few Palm-specific requirements that make the port distinct to the Palm platform. The task of finding Net.lib, opening, and closing the library remain even with the Berkeley compatibility layer. Additionally, the NetSocketRef data type is used, despite the fact that Berkeley simply uses integers as socket identifiers. Porting a large body of networking code can benefit from this approach, although it for simple applications, a clean port to Net.lib might be preferable. One of the sample applications included in this chapter, Daytime Berkeley, demonstrates using the compatibility.

Here are the basic steps to using the Berkeley sockets compatibility:

- Add NetSocket.c to the application. This file is part of the SDK.
- Include the sys_socket.h header file in the source files. This provides key macros and `#defines`.
- The global variable `AppNetRefNum` should be used in any Net.lib functions, such as `NetLibOpen` and `NetLibClose`.
- The global variable `AppNetTimeout` is used by every function requiring a `timeout` parameter. This value should be adjusted accordingly.

The goal of this chapter is to introduce the topic of networking programming on the Palm OS platform, providing enough guidance to get started. To that end, there are three sample applications demonstrating the activities presented throughout the chapter. Note that the topic of network programming is very large, and in fact there are entire books that cover the topic in depth.

Roadmap to Sample Applications

`http_palm`

This application is a basic HTTP client, written in native Net.lib sockets form. The application prompts the user for the target URL. Support is provided for fully qualified domains and port numbers other than 80. Authentication is not supported and would be an excellent extension of the application. Once the URL is gathered, the application executes a simple `GET` operation. The receiving process is broken into a series of functions to handle the response in an orderly fashion. The `CONTENT-LENGTH HTTP` header is relied upon for receiving the body of the response. Chunked encoding is not supported.

As more and more software products expose services via *Web Services* and *HTTP*, a working knowledge and understanding of the HTTP protocol is essential for navigating the future computing landscape. An example of this type of service is an online search mechanism available from Google, which exposes their comprehensive search engine to client applications.

`daytime_palm`

This application demonstrates a simple UDP application. It prompts the user for a target hostname and then queries that host for the date and time. The daytime protocol is available in both TCP and UDP forms. In the UDP form, which this application exercises, the client application sends an empty datagram, or packet, to the server. The server then responds with a single packet containing the date and time.

`daytime_berkeley`

This application is a port of the daytime_palm application to use the Berkeley socket functions. A side-by-side comparison of this project and the daytime_palm application shows an additional source file NetSocket.c. This file is normally found in the SDK; however, it has been copied to the local Src folder for convenience.

Summary

This chapter has introduced the basics of Palm OS network programming, including both TCP and UDP concepts. The sample applications present an excellent starting point for the beginning Palm OS network programmer. Additionally, porting considerations have been addressed, opening up the Palm OS platform to benefit from the multitude of networking applications available from other platforms. Finally, adding network support to new and existing applications enhances the platform's usability and function.

PART V
Advanced Topics

IN THIS PART

27 Overview of Conduits

IN THIS CHAPTER

- What Is a Conduit?
- How Does Hot-Sync Interact with Conduits?
- How to Obtain the Conduit SDK
- Development Tools and Programming Environment
- Other Conduit Development Options

Although when you think of Palm programming, you probably think of developing code that actually runs on a handheld device, an important part of the appeal of the Palm product is the ease with which it interacts with a desktop computer. Indeed, the first thing most Palm handheld customers do after they purchase their PDA is connect it to their desktop computer and synchronize their PDA with their PC.

The piece of software that makes this connection possible is called the HotSync Manager. The HotSync Manager is a kind of overseer that routes all data communications that occur between the PDA and the host computer. One of the HotSync Manager's main jobs is to track which application is responsible for each type of data on the handheld, and make sure that the appropriate application is given the opportunity to transfer, or "synchronize" that data.

In order for an application on the Palm device to synchronize its data with a host computer, a piece of software must be written that handles the synchronization from the desktop computer's perspective. This piece of software is called a *conduit* because it manages the flow of data through a pipe between a desktop application and a handheld application.

More generally, conduits provide a facility for transferring data and applications back and forth between Palm Computing devices and host platforms such as a desktop PC, corporate server, or Internet location.

Conduits are a bit of a paradox in that they lie both at the heart of the Palm computing platform and on the periphery of the world of Palm programming. Without conduits and Palm's hot-sync technology, data could not flow to and from the Palm, applications could not get installed on the device, and the synchronization of data between desktop-based applications and their Palm equivalents could not occur. Yet the Conduit Software Development Kit is a completely separate set of developer tools from the Palm OS SDK, in terms of how it is packaged and distributed by Palm Computing.

Further, conduits are not programmed using the same development tools you use for normal Palm development, such as CodeWarrior or PRC-Tools. In fact, although it is possible to use Java or other tools and languages to program a conduit, most conduits are created using C++ in Microsoft's Visual Studio.

Many—perhaps most—applications will never require the development of a custom conduit. Case in point: I've managed to get through all of the chapters in this book, with nary a mention of the topic of conduits. In addition, a quick scan of the commercial software, shareware, and freeware available for the Palm indicates that most of them do not include a custom conduit either.

Nevertheless, the rise of the Palm as a legitimate business tool for use in deploying corporate applications and providing access to corporate data is driving the need to deliver non-PIM data to the Palm. Many of these applications will require conduit components to help transfer or even synchronize that data. Third-party tools are available that minimize the need for you to resort to the Conduit SDK for these applications, but there will still be situations where a custom conduit will be preferable.

Conduit development, somewhat deservedly, has a reputation as being a mysterious black art mastered by a very few gurus. To the fledgling Palm programmer, I'll grant that conduits can seem somewhat off the beaten path due to the programming paradigm and the different toolset, but conduit development is not as scary as some would make it out to be.

This chapter covers

- What a conduit is
- How the hot-sync facility interacts with conduits
- How to obtain the Conduit SDK and what it contains
- The development tools and environment

The next chapter actually presents an example conduit that you can use as the basis for your own conduits.

What Is a Conduit?

Unless your Palm is equipped with a modem (wired or wireless), conduits are Palm's primary framework for communicating with the outside world, including data and applications residing on other computer systems. Conduits work in concert with the HotSync Manager to perform the following tasks:

- Mirroring and synchronizing with data on other systems
- Backing up data residing on the Palm
- Downloading and installing Palm OS applications
- Importing and exporting data

Conduits provide a way for you to partition your application so that one piece runs on the Palm device while more processor- and memory-intensive tasks run on more capable host systems. You can then deploy a conduit to exchange database records and other types of information between the two pieces of the application. (You can consider this arrangement a strange kind of form of client/server application!)

Depending on the application requirements, you can design a conduit to take one of four forms:

- A data mirror
- A one-directional conduit
- A transaction-based conduit
- A backup

Data Mirroring

Data mirroring is the "classic" synchronization activity, and it is performed by the default conduits for the address book, date book, and so on. The conduit intervenes and takes appropriate action when the desktop or the handheld application modifies data.

One Directional

The one-directional conduit assumes that either the desktop or the handheld will be modifying data, but not both. The synchronization activity is limited to copying the data from one of the platforms to the other. If the Palm is to be used as a read-only front end to corporate data, choosing this type of conduit might be a good strategy. Conversely, if remote data entry is to be performed on the Palm, you can simply transfer the data collected back to the host for further processing.

Transaction Based

Transaction-based conduits perform additional processing on the host computer for each synchronized record. If an order record was added from the Palm, for example, and the host system required that a customer order number be generated for each new record, the number might be generated at synchronization time. This type of activity can significantly impact how long the conduit will take to run to completion, so it is recommended that you use it with caution. There are usually other ways to accomplish the same goals using multiple passes.

Backup

The backup is perhaps the simplest kind of conduit in that it only needs to transfer the data from the Palm to the host computer. Most situations that call for a backup conduit will not require a custom-developed conduit but can instead rely on the backup conduit supplied by Palm.

How Does Hot-Sync Interact with Conduits?

A conduit is not a Palm application at all, but rather a Windows dynamic link library (DLL) that plugs into a special interface in the HotSync Manager so that it is invoked when the users perform a hot-sync. This DLL uses Conduit SDK interfaces to access data managed by the desktop application and transfers data to and from the target Palm application.

You can perform a hot-sync via cable, modem, or network connection. Third-party vendors have also provided infrared synchronization utilities. The HotSync Manager handles all these connection types, hiding the details behind a standard communication interface.

The HotSync Manager runs as a background task on your PC. (You've no doubt seen the icon appear in your Taskbar system tray.) It monitors a designated serial communications port, waiting for a notification that the Palm device is requesting a hot-sync.

When a hot-sync is initiated, the HotSync Manager receives the creator ID for each database installed on the Palm and checks whether there is a conduit registered to handle that creator ID. If there is, the HotSync Manager then calls the conduit and asks it to perform whatever tasks it was designed for.

This level of coordination with the HotSync Manager is possible because there are standard programming entry points that your conduit must support. These entry points are called as functions by the HotSync Manager when a hot-sync is being performed. The primary entry point is `OpenConduit`, in which all the synchronization activity takes place. Inside your DLL, you call functions in the HotSync Manager API in order to communicate with the data on the Palm device.

With multiple versions of the Palm in the field, there are also multiple versions of the HotSync Manager and other Palm desktop software. You are responsible for coordinating versions of your conduit with versions of the HotSync Manager and responding correctly when the HotSync Manager asks your conduit for its version number.

Once created, conduits must be installed on your desktop so that the HotSync Manager can recognize them. This is done by inserting a number of Registry keys and values in the Windows Registry, one of which is the target creator ID for your conduit (which must match the creator ID of the database to be synchronized). Current versions of the Conduit SDK provide a sample `InstallShield` script that you can use as a basis for your own installation program.

How to Obtain the Conduit SDK

The Conduit SDK is available for downloading from the Palm OS Web site at `www.palmsource.com`. A copy of the SDK is included with CodeWarrior's toolset, but it is a good idea to frequently check the Palm Computing Web site because the SDK has undergone frequent revisions in recent times.

The Conduit SDK comes with several useful components:

- *Programmers' Guide*: An overview and introduction to conduit concepts and the hot-sync mechanism.
- *Programmers' Reference*: A detailed reference to the HotSync Manager and other conduit-related APIs.
- Sample conduit code: The code for the built-in conduits as well as a skeleton conduit.
- Condmgr.exe: A utility for quickly adding a conduit to the Registry without having to write an install script.
- Headers and libraries: These are required in order to build a conduit in Visual Studio.

Development Tools and Programming Environment

The Conduit SDK requires the Microsoft Visual C++/Visual Studio environment as well as the Microsoft Foundation Classes in order to create Windows-based conduits.

To make it easier to get up and running with conduit development, Palm has provided a conduit wizard that plugs into the Visual Studio Project Wizard. Creating a basic conduit shell is now easily done by choosing File, New from the Visual Studio menu, choosing Conduit from the Project tab, and answering a series of questions regarding how your conduit will be used. Visual Studio then proceeds to create an entire project complete with source code, ready for building.

Other Conduit Development Options

If you want to create a conduit suitable for users who perform hot-sync to a Mac OS host, you'll find support in the Mac OS version of Metrowerks.

Also, a Java version of the Conduit SDK works with tools such as Symantec's Café development tool, which allows you to create Windows-based conduits.

Summary

In this chapter, I introduced Palm conduits. I explained what you can use conduits for, and you reviewed the hot-sync process in detail in order to understand where the built-in conduits (as well as the conduits you develop) fit in with the hot-sync mechanism. I also described the contents of the Conduit SDK and the development environment.

The next chapter deals with the task of actually creating a conduit, and presents a full working example. There is a Palm-support discussion forum dedicated to those brave souls who dare to develop conduits. Visit the Palm Computing Developer Zone Web site to subscribe to this list, or browse past messages at `www.egroups.com`, which hosts the discussion group `conduit-dev-forum`.

28 Programming Palm OS Conduits

IN THIS CHAPTER

- Framework Choices
- Structure of a Basic Conduit
- Optional Interfaces
- The Generic Conduit Framework
- Using the HotSync Manager APIs
- The HotSync Log API
- Putting It All Together: A Simple Conduit

In the last chapter I introduced the general topic of the Palm OS conduit. As you saw, conduits are generally responsible for exchanging databases and synchronizing information between a PDA and a host computer. I covered in broad strokes which development tools were used, and the types of conduits.

Even just learning how conduits work is a complex undertaking. Learning how to program a conduit is extraordinarily complex, and is not something to be taken lightly. Of all of the Palm OS programming topics covered in this book, conduit programming is easily the most difficult one. In the Palm developers toolkit, conduits are in their own developers kit (called the CDK, or Conduit Developers Kit), have their own documentation, and use an entirely different set of development tools and APIs than do programs that run on the device itself.

None of this is meant to alarm you—there is a wealth of information now available about conduit programming in the CDK, and Palm has worked hard to make conduit programming simpler than ever. But synchronizing data between two computers running two different platforms is a complex task no matter how you slice it, so it is important to have a healthy respect for the scope of the problem before diving into code.

This chapter covers issues related to programming conduits, including

- Selecting a development model for your project
- Design guidelines

- Differences between the generic framework and the low-level APIs
- Required interfaces for your conduit
- HotSync Manager APIs used in conduit programming

Also included with this chapter is a very simple conduit example using the HotSync Manager APIs, which illustrates some of the concepts introduced here.

Conduit programming is a vast topic, arguably one that deserves an entire book all to itself. This chapter will help you get your feet wet with the basics, and give you a good head start by using a basic conduit template as a foundation for your own conduit projects.

Framework Choices

In the previous chapter I mentioned that conduits can be developed in either C++ or Java. Although accurate at a high level, the choices are actually far more complex than that.

Presumably based on developer demand for more familiar development options, Palm has added support for a COM interface, which not only adds more options for the C++ developer, but also enables integration with tools such as Visual Basic.

Furthermore, within the C++ development kit (now called the C++ Sync Suite) there are two options, the Generic Conduit Framework, and the HotSync Manager. Palm presents these two options as if the Framework is the “easy” option and the HotSync Manager is the “hard” option. I’m not so sure that’s true, and it still does not address the inherent complexity of synchronization that remains, but admittedly there are fewer APIs to deal with and more classes to inherit from in the Framework.

Although I will touch briefly on the structure of a Generic Conduit Framework-based conduit, this chapter focuses more on the C++ HotSync Manager APIs.

Structure of a Basic Conduit

Because this discussion is focused on a C++ Windows conduit, it involves creating a standard Windows DLL. As a Windows DLL, there is nothing magical about the conduit, and indeed it can call any Windows API that is legally callable from within a DLL.

What makes a DLL a legitimate conduit is simple: It registers itself with the HotSync Manager, and it implements very few required interfaces that the HotSync Manager calls during a hot-sync operation.

The required interfaces include

`OpenConduit`

`GetConduitVersion`

`GetConduitInfo`

OpenConduit

`OpenConduit` is pretty much where everything happens inside your conduit. When the HotSync Manager decides it is your conduit's turn to "do its thing," it calls `OpenConduit`.

Inside `OpenConduit` you must do everything you need in order to synchronize your application data between the handheld and the host computer. This includes opening both the handheld and host databases, reading all necessary data, comparing records, adding, deleting or modifying data on both sides, and cleaning up.

GetConduitVersion

`GetConduitVersion` is a simple function that simply reports the version of your conduit back to the HotSync Manager.

GetConduitInfo

`GetConduitInfo` is responsible for providing several pieces of information to the HotSync Manager, including the version of MFC (if you used MFC in your conduit), the default sync action (do nothing, handheld overwrites PC, PC overwrites handheld, and so on), and the name of your conduit.

Optional Interfaces

There are additional interfaces you can implement in your conduit, but they are technically not required. The most important of these is `ConfigureConduit` (formerly known as `CfgConduit`). This function is called when a user chooses `Custom` from the HotSync Manager and wants to change the behavior of your conduit after selecting it from the HotSync Manager's list. Although it is possible for a conduit to be installed and working without providing a configuration screen, it is highly recommended that you give your users the option.

The Generic Conduit Framework

The idea behind the Generic Conduit Framework is a noble one: Conduit development is very complex, so why not hide a lot of the dirty work in a class framework and simplify life for the programmers? Combined with a Visual Studio compatible Conduit Wizard, on the surface using the Framework does appear to be cleaner.

The sample conduits provided with the CDK (under C++/Samples) are written using the Framework. You'll notice that in the `OpenConduit` implementation, there an instantiation of something called a Synchronizer. This is a class that is responsible for managing all sync activity in your conduit. Your responsibilities are to tell your Synchronizer object everything it needs to know about your database structure, and implement the required Synchronizer interfaces in your derived object.

For this reason, the `OpenConduit` implementation in a Framework conduit can appear extremely clean. However, in some ways this is more a matter of shifting complexity from `OpenConduit` to your Synchronizer class, so beyond the most basic of conduits, it's not clear how much effort is saved. I am sure there are strong proponents of the Framework out there, I just don't happen to be one of them.

For more information on using the Generic Conduit Framework, please see the CDK developer documentation as well as the samples that accompany the C++ Sync Suite.

Using the HotSync Manager APIs

The HotSync Manager APIs are a set of functions that allow you to open, close, read, write and modify databases that reside on your handheld. With this capability you can perform any type of sync activity you need, including backup, install, and one-way or two-way syncs.

As I noted earlier, this activity will happen inside of the required `OpenConduit` function. The following sections break out each category of HotSync Manager APIs and provide examples of their usage.

Conduit Registration

In order for your conduit to participate in a synchronization session, you first need to call `SyncRegisterConduit`. `SyncRegisterConduit` must be successfully called before making any other HotSync Manager API calls.

After your conduit is finished working, you should call `SyncUnRegisterConduit`.

Opening and Closing Databases

Your conduit can only perform its functionality by virtue of having the privilege through the HotSync Manager of opening, reading, and writing handheld databases directly on the device, all from the vantage point of a Windows DLL. The nice thing about this is that your conduit DLL is insulated from any communications awareness: Your code should work the same whether synchronization is happening over a serial, USB, infrared, modem, or network connection.

Although the APIs are somewhat different (and more limited) than those you work with on desktop databases, the net effect achieved is one where you can open your handheld database with almost as much ease as a desktop database.

To open an existing handheld database, you need to call `SyncOpenDB`. `SyncOpenDB` requires a database name, a memory card number (be careful, this is not an expansion card in the VFS sense), and mode flags that dictate how you will use the open database. Note that you can open only one database at a time; you must close the database prior to opening an additional one.

If successful, a handle to the open database is returned to you. This handle must be used in subsequent calls that interact with the open database.

Closing a database is simple: Just call `SyncCloseDB` with the handle you received from `SyncOpenDB`.

Creating and Deleting Databases

If you want to create a new database on the handheld, you can use `SyncCreateDB`. `SyncCreateDB` has a similar signature as its Palm sibling, `DmCreateDatabase`, in that it requires a creator ID, type, card number, and other attributes associated with Palm databases.

Deleting a handheld database is also similar to its Palm counterpart `DmDeleteDatabase`. You simply specify a database name and card location.

Iterating, Reading, and Writing Records

Once you have an open database, you can retrieve and manipulate records in the database, as well as add and delete records.

The HotSync Manager API provides several functions that allow you to navigate through a handheld database. The record itself in all these APIs is represented as a structure called a `CRawRecordInfo`. A `CRawRecordInfo` structure is used to either set record information or retrieve it from a record.

If you know the record you want to read, either by ID or by index, you can use direct access functions like `SyncReadRecordByID` or `SyncReadRecordByIndex`. If you need to iterate through a number of records, you can use `SyncReadNextRecInCategory`, `SyncReadNextModifiedRec`, or `SyncReadNextModifiedRecInCategory`. The advantage of iterating only modified records is that in many cases you will want to sync only records that have changed. For the iterator-type functions, you can move to the first record in the iteration by calling `SyncResetRecordIndex`.

Once you have a record in your hand (so to speak), you can review its field values and attributes to determine how to deal with it in light of anything that might have happened on the desktop side of things. In some cases you will want to replicate the record on the desktop, whereas in others you may need to surgically update individual fields, or even delete the record either on the desktop or the handheld.

If you need to update the handheld record, you can use `SyncWriteRec`. This function can also be used to add a new record to the handheld database.

Finally, you can delete handheld records by calling `SyncDeleteRec`.

Record Attributes

Central to the idea of synchronization is performing the correct action based on events that have occurred on either the host or the handheld since the last sync. This is possible through a series of record attributes supplied with each record in the `CRawRecordInfo` structure.

A handy decision matrix is included in the CDK documentation that helps you determine the correct action based on the attributes found on a handheld record combined with the status of the corresponding desktop record.

The HotSync Log API

It is important to realize that you are restricted in terms of the normal ways of communicating errors and other information to your users when your conduit is running. Although there is nothing to stop you from calling Windows functions such as `MessageBox`, the Palm conduit design guidelines prohibit you from relying on user interface actions to communicate with the users.

This guideline is there for good reasons. The first is that users expect hot-sync to be a simple, magical process that happens quickly and silently when they press the hot-sync button on their PDA or cradle. Users are not interested in monitoring the progress of hot-sync, nor are they likely to make an informed decision about what to do if your conduit reports a problem.

The other reason for avoiding user interface during hot-sync is that in the real possibility of your conduit running over a modem or network connection, your users will not even be anywhere near the PC that your conduit is running on.

Nonetheless, it is important to provide a way to communicate hot-sync results to the users. In cases where there are hot-sync conflicts, or other unforeseen errors, the appropriate thing to do is add an entry to the hot-sync log. Although arguably few users are even aware of its existence, the log is accessible from the HotSync Manager via the pop-up menu. Every conduit that is called during a hot-sync has an opportunity to provide feedback to the users via the log.

If nothing else, the log can sometimes be a last resort method of debugging a conduit problem with a remote user of your program. If your conduit is verbose about what is happening during its operation (perhaps in a special debug mode), you can ask your users to send you the contents of the log after running your conduit.

To add an entry to the log, simply call `LogAddEntry`. Note that the log must first be initialized by calling `LogInit`.

Putting It All Together: A Simple Conduit

Thus far, I've covered conduit programming at a conceptual level, but there is nothing like seeing a concrete example that ties the concepts together into a working model of how conduits operate in a real-world scenario. Above all other topics in Palm programming, conduit development does not really start to make sense until you can look at a real example.

The goal for SyncEx, the conduit example, is to take the *Fish Database* project that you've been working on through several other chapters, and enhance it so that it can be synchronized with a database that exists on a PC. As you may recall, Fish is a simple program that can maintain a database of fish species on a Palm device. It lets users list the fish on the Palm display as well as add new species.

Synchronizing the Fish Database

For this chapter, you are going to create a conduit that will synchronize the records on a Palm device with records that reside on a "database" that exists on the PC. To keep things simple, for a database you will use a text file format, with comma-delimited record data. Although rather simplistic, the concepts used in the example are applicable to conduits that interface with more advanced PC database formats.

The job of the conduit is to replicate any changes made to the Fish database on either side of the system, Palm or PC. This means if I add a new fish record on the Palm, after a hot-sync operation, a copy of that fish records should wind up in my PC text file. Similarly, modifications and deletions that occur on the PC or Palm will be replicated to the other side.

Enhancements to the Palm Fish Program

In order to fully illustrate the concepts in the example, I needed to make some minor enhancements to the Palm Fish program created in Chapter 18, "Palm Databases and Record Management." Specifically, the following enhancements were made:

- Ability to modify existing Fish records
- Ability to delete existing Fish records
- Ability to set a record's "dirty" bit to indicate it has been updated on the device

These changes are readily apparent in new code added to the example source code in tbls_mai.cpp and tbls_db.cpp.

Tool Assumptions

The example conduit presented here requires a few basic assumptions about development tools. First, I assume that you have downloaded and successfully installed the Palm Conduit Software Development Toolkit (CDK) on your PC. This example was specifically created using the CDK version 4.02a. Second, the conduit project assumes that you are using Microsoft's Visual C++ version 6.0 or better.

If you do not have these development tools installed, you can still read through the source code, but it is easier to understand the structure of the conduit when you can see it in the VC++ IDE as an actual project.

The SyncEx Project

Some explanation is in order as to how the SyncEx project came about. Perhaps the most difficult part of conduit development for the beginner is determining just how to get started. We've been developing conduits at my company since the beginnings of the CDK. Although you might say we've had to learn conduit programming "the hard way," I think we've actually benefited in the long run because it forced us to create a nice template project that we use as a basis for virtually any new conduit project we embark on.

Over the years, Palm has tried to make conduit programming easier for a wider audience of programmers by introducing Visual C++ "wizards" for a variety of conduit project types, and by introducing a series of frameworks and class hierarchies that attempt to hide away the complexities of the conduit structure. Although I applaud Palm's efforts in this area, I still find the code that results from the "wizard approach" to be hard to understand, and over the years we have found it necessary to build up a series of helper classes and functions to reach an acceptable level of reusability in our conduit projects.

What we have ultimately settled upon as a basic project template is a melding of the output from the conduit wizard, some additional glue to make the conduit MFC-friendly, and also some boilerplate helper functions that we find extremely useful in every conduit we develop.

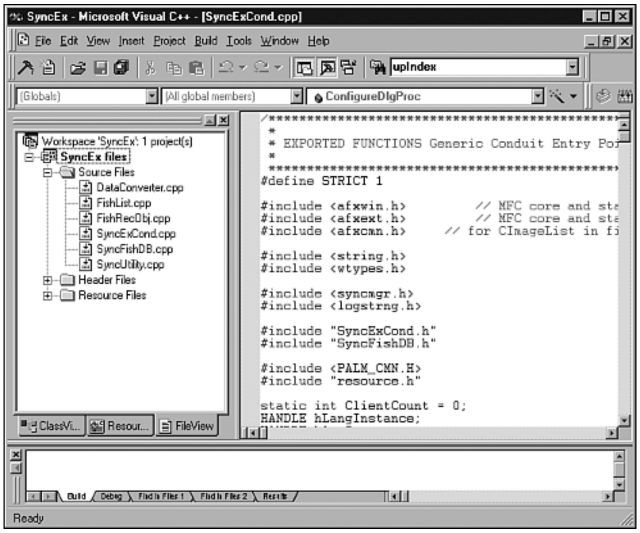
The SyncEx project is based in large part on this reusable template, and in my opinion it represents a pretty clean, simple conduit project that can be easily reused on further projects. To use it on other projects, all you really need to do is copy and paste the entire project, and then make just a very few name changes.

Of course, you are free to use a different approach to project creation than the one I use here; there is no right or wrong way to do it. And it's certainly useful at least as an exercise to run the Palm conduit wizard and view the output. But for the remainder of this chapter I assume the use of the conduit template project style.

A Tour of the SyncEx Project Files

Before I delve into the source code, it's useful to embark on a brief overview of the structure of the SyncEx project, so that you understand where everything is and the purpose of each file.

The SyncEx project is pictured in Figure 28.1.



```
/*********************************************
 *
 * EXPORTED FUNCTIONS Generic Conduit Entry Poi
 *
 *********************************************
#define STRICT 1

#include <afxwin.h>         // MFC core and sta
#include <afxext.h>         // MFC core and sta
#include <afxcmn.h>     // for CImageList in fi

#include <string.h>
#include <wtypes.h>

#include <syncmgr.h>
#include <logstrng.h>

#include "SyncExCond.h"
#include "SyncFishDB.h"

#include <PALM_CMN.H>
#include "resource.h"

static int ClientCount = 0;
HANDLE hLangInstance;
```

FIGURE 28.1 The SyncEx project in the Visual C++ IDE.

The SyncEx project is structured as a Windows MFC project. It has source CPP files, related header files, and resources.

The resources in SyncEx are rather minimal. They consist simply of the standard HotSync conduit configuration dialog, and associated bitmaps and string resources. These resources were all generated by the conduit wizard, and are included as a standard part of just about every conduit project.

There are six source CPP files in SyncEx, as follows:

- **SyncExCond.cpp:** This file contains the required entry points for a hot-sync Conduit, including `OpenConduit`, `GetConduitInfo`, `GetConduitVersion`, `GetConduitName`, and `ConfigureConduit`. It also contains some boilerplate MFC code, as well as the window procedure for the conduit configuration dialog.
- **SyncFishDB.cpp:** This file is the implementation for the `CsyncFishDB` class, which manages and handles all of the synchronization activity in SyncEx.
- **FishList.cpp:** This file contains the implementation of the `CFishList` class. `CFishList` is the interface layer for managing the PC Fish database (the fish.txt file). It can iterate through fish records, add them, modify them, and delete them from the PC database.
- **FishRecObj.cpp:** This file contains the implementation of the `CfishRecObj` class, which is responsible for interfacing to the Palm Fish database, referred to as the "remote" database.
- **SyncUtility.cpp:** This file contains `CSyncUtility`, a helper class that takes much of the boilerplate code associated with synchronizing remote records and hides it behind a few easy-to-use member functions.
- **DataConverter.cpp:** This file contains `CDataConverter`, another helper class. `CDataConverter` automatically manages the conversion of data fields between the remote device and the PC. All necessary byte swapping and other tasks related to endian-ness and data formats across platforms is handled in this class.

Of the six source files in the project, only three of them really change from project to project: `SyncFishDB`, `CfishList`, and `CFishRecObj`. Even those files remain constant in their purpose and their general structure, with only the nature and layout of the target database changing. The other three files contain boilerplate conduit code and helper functions, and rarely need modification from project to project.

The PC Fish Database Structure

One last thing to cover before you see SyncEx in action is the structure of the PC version of the Fish database. In the example, you have a rather simple structure. Fish.txt is a text file consisting of records separated by carriage returns. In turn, the records are composed of fields, each one separated by a comma.

The first record in the fish.txt file is a special record that contains the count of the number of records in the fish.txt database.

The remaining records in the fish.txt file are fish records. The structure of a fish record consists of the fish name field, the fish description field, the Palm database record ID, and a PC transaction code.

The fish name and description are the same as in the Palm version of the fish database, but what are the Palm record ID and the transaction code doing here?

The record ID is a good idea to maintain in your PC database if you can. The biggest benefit of doing so is the performance gains made by being able to directly jump to a single record on the remote Palm device without having to preload all Palm records into memory, or having to iterate through all the records to find the one you are looking for.

The transaction code in the record structure is helpful for understanding which action was last associated with each record on the PC. The valid transaction codes are found in the header file BaseRecObj.h, and are `RECORD_TRANS_NONE`, `RECORD_TRANS_ADD`, `RECORD_TRANS_UPDATE`, and `RECORD_TRANS_DELETE`. These codes are maintained for each record on the PC so that the conduit can easily determine which handheld action to take for any given record on the PC.

SyncEx in Action

In this section, SyncEx comes to life, and I walk you through the lifecycle of a HotSync session as seen from the perspective of the SyncEx conduit.

As with every conduit, life begins with a call to the standard hot-sync entry point `OpenConduit`. `OpenConduit` is called by HotSync Manager when it is your conduit's turn to perform synchronization.

In SyncEx, `OpenConduit` is found in the SyncExCond.cpp source file along with the other standard conduit entry points.

```
ExportFunc long OpenConduit(PROGRESSFN pFn, CSyncProperties& rProps)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());
   long retval = SYNCERR_UNKNOWN;
   if (pFn)
   {
      // instantiate a CSyncFishDB object
      CSyncFishDB* pSyncFishDB = new CSyncFishDB( rProps );
      if ( pSyncFishDB )
      {
         // perform sync
         retval = pSyncFishDB->Sync();
        // clean up
```

```
         delete pSyncFishDB;
      }
   }
   return(retval);
}
```

OpenConduit in SyncEx is a very short function, but that's because all of the real action takes place inside of the CSyncFishDB class, specifically in the Sync member function. So all that really happens in OpenConduit is some basic MFC housekeeping, followed by an instantiation of a CSyncFishDB object. If instantiation is successful, the CsyncFishDB's Sync() member function is called.

CSyncFishDB is the next stop. A listing of CSyncFishDB is next, followed by an explanation of the relevant member functions in sequence of execution.

```
/*
CsyncFishDB
Implementation of the CsyncFishDB class
Copyright © 2002 Bachmann Software and Services, LLC
*/
#include "stdafx.h"
#include <syncmgr.h>
#include <hslog.h>
#include "SyncFishDB.h"

CSyncFishDB::CSyncFishDB( CSyncProperties& rProps )
{
   memcpy( &m_rSyncProperties, &rProps, sizeof( CSyncProperties ) );
   m_pSyncUtility = new CSyncUtility( m_rSyncProperties );
   m_pFishList = NULL;
}

CSyncFishDB::~CSyncFishDB()
{
   delete m_pSyncUtility;
   CFishList::DestroyList();
}

// Sync
// Performs synchronization of FishDB between PC and Palm

long CSyncFishDB::Sync()
{
   CONDHANDLE   conduitHandle = (CONDHANDLE)0;
   long retval = SYNCERR_UNKNOWN;
   char      appName[] = "SyncEx";

   // Register this conduit with SyncMgr.DLL for communication to HH
   if (retval = SyncRegisterConduit(conduitHandle))
      return(retval);

   //perform the sync
   retval = SyncDB();

   // application name
   LogAddEntry(appName, slSyncFinished,FALSE);

   // clean up
   SyncUnRegisterConduit(conduitHandle);

   return retval;
}

// Sync
// Performs the actual DB sync according to the type of sync
// that is necessary
long CSyncFishDB::SyncDB()
{
   long retval = -1;

   InitializeDB();

   switch (m_rSyncProperties.m_SyncType)
   {
      case eFast:
         retval = FastSync();
         break;
      case eSlow:
         retval = SlowSync();
         break;
      case eHHtoPC:
         retval = HHToPCSync();
         break;
      case ePCtoHH:
         retval = PCToHHSync();
         break;
      case eDoNothing:
         retval = 0;
         break;

      default:
         break;
   }

   return retval;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////Local Store methods/////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

// InitializeDB
// Loads up the list of PC records into our CFishList object
long CSyncFishDB::InitializeDB()
{
   long err = 0;

   m_strDirPath = m_rSyncProperties.m_PathName;
   m_pFishList = CFishList::CreateList( m_strDirPath );

   strcpy ( m_pszRemoteName, "FishDB" );

   // for this example, assume that if any records have a
   // sync conflict, the Palm record wins the conflict.
   m_bPalmWins = TRUE;

   return err;
}


// AddFishRecToPC
// Adds a fish record to the list
long CSyncFishDB::AddFishRecToPC( CFishRecObj* pFishObj )
{
   long err = 0;
   m_pFishList->AddFish ( pFishObj );

   return err;
}

long CSyncFishDB::UpdateFishRecOnPC( CFishRecObj* pFishObj )
{
   long err = 0;
   BOOL bFound = FALSE;
   CFishRecObj* pFindObj;

   pFindObj = m_pFishList->GetFirstFish( );
   while ( pFindObj )
   {
      //we found the Fish record to update
      if ( pFishObj->GetRecordID() == pFindObj->GetRecordID() )
      {
         pFindObj->Copy ( pFishObj );
         bFound = TRUE;
         break;
      }
      pFindObj = m_pFishList->GetNextFish( );
   }

   //if we didn't find it, then it's added
   if ( !bFound )
   {
      err = AddFishRecToPC( pFishObj );
   }

   return err;
}

long CSyncFishDB::DeleteFishRecOnPC( CFishRecObj* pFishObj )
{
   long err = 0;

   m_pFishList->DeleteFish ( pFishObj );

   return err;

}
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////Different Sync Methods/////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

long CSyncFishDB::FastSync()
{
   long retval = SYNCERR_NONE;

   if ( m_bPalmWins )
   {
      SyncHHChanges();
      SyncPCChanges();
   }
   else
   {
      SyncPCChanges();
      SyncHHChanges();
   }
   return retval;
}

long CSyncFishDB::SlowSync()
{
   long retval = SYNCERR_NONE;

   retval = HHToPCSync();

   return retval;
}

//syncs the changes made on the pc to the HH
long CSyncFishDB::SyncPCChanges()
{
   long err = -1;
   CFishRecObj* pFishRecObj = NULL;

   //open the remote table
   err = m_pSyncUtility->OpenRemoteTable( m_pszRemoteName );

   // For each PC record, update the HH record
   pFishRecObj = m_pFishList->GetFirstFish( );
   while ( pFishRecObj && !err )
   {
      if ( pFishRecObj->GetTransCode() == RECORD_TRANS_ADD )
      {
         err = m_pSyncUtility->AddRemoteRecord( pFishRecObj );
         pFishRecObj->SetTransCode ( RECORD_TRANS_NONE );
      }
      else if ( pFishRecObj->GetTransCode() == RECORD_TRANS_UPDATE )
      {
         err = m_pSyncUtility->ChangeRemoteRecord ( pFishRecObj );
         pFishRecObj->SetTransCode ( RECORD_TRANS_NONE );
      }
      else if ( pFishRecObj->GetTransCode() == RECORD_TRANS_DELETE )
      {
         err = m_pSyncUtility->DeleteRemoteRecord( pFishRecObj );
      }
      pFishRecObj = m_pFishList->GetNextFish ( );
   }

   m_pSyncUtility->CloseRemoteTable(  );

   return err;
}

//syncs the changes made on the HH to the pc
long CSyncFishDB::SyncHHChanges()
{
   long         err = 0;
   CFishRecObj fishRecObj;
   CRawRecordInfo*   pRawRecord = m_pSyncUtility->GetRawRecord ( );

   m_pSyncUtility->ResetRawRecord( );

   //open the remote table
   err = m_pSyncUtility->OpenRemoteTable( m_pszRemoteName );
   if ( err )
   {
      return err;
   }
   BYTE hRemoteDB = m_pSyncUtility->GetRemoteDBHandle();
```

```
pRawRecord->m_FileHandle = hRemoteDB;
pRawRecord->m_RecIndex = 0;

// Read in each modified Remote (HH) record one at a time
while ( !err )
{
   if ( !( err = SyncReadNextModifiedRec( *pRawRecord ) ) )
   {
      //convert the remote rec to PC object
      err = fishRecObj.ConvertFromRemote( *pRawRecord );
      if ( !err )
      {
         //check for delete flag
         if ( pRawRecord->m_Attribs & eRecAttrDeleted ) // delete flag
         {
            //delete the record
            err = DeleteFishRecOnPC ( &fishRecObj );
         }
         //if the record has changed or been added; update it
         else if ( pRawRecord->m_Attribs & eRecAttrDirty )
                     // Dirty flag
         {
            //update the record on the pc
            err = UpdateFishRecOnPC ( &fishRecObj );
         }
      }
   }
}

//check for no more records and set flag to ok
if ( err == SYNCERR_NOT_FOUND )
{
   err = 0;
}

// Delete all records marked for deletion on the handheld
SyncPurgeDeletedRecs( hRemoteDB );

SyncResetSyncFlags( hRemoteDB );
m_pSyncUtility->CloseRemoteTable(  );

return err;
```

```
}

long CSyncFishDB::HHToPCSync()
{
   long         err = 0;
   WORD        recIx = 0;
   CFishRecObj fishRecObj;
   CRawRecordInfo*   pRawRecord = m_pSyncUtility->GetRawRecord ( );

   //open the remote table
   err = m_pSyncUtility->OpenRemoteTable( m_pszRemoteName );
   if ( err )
   {
      return err;
   }
   BYTE hRemoteDB = m_pSyncUtility->GetRemoteDBHandle();

   // Delete all records marked for deletion on the handheld
   SyncPurgeDeletedRecs( hRemoteDB );

   //clear out the records on the PC
   m_pFishList->ClearList();

   pRawRecord->m_FileHandle = hRemoteDB;
   pRawRecord->m_RecIndex = recIx;

   // Read in each Remote (HH) record one at a time
   while ( !err )
   {
      pRawRecord->m_RecIndex = recIx ;
      if ( !( err = SyncReadRecordByIndex( *pRawRecord ) ) )
      {
         //convert the remote rec to PC object
         err = fishRecObj.ConvertFromRemote( *pRawRecord );
         if ( !err )
         {
            // Add the remote record to the PC
            err = AddFishRecToPC( &fishRecObj );
         }
      }
      recIx++;
   }
```

```
    //check for no more records and set flag to ok
    if ( err == SYNCERR_NOT_FOUND )
    {
       err = 0;
    }

    SyncResetSyncFlags( hRemoteDB );
    m_pSyncUtility->CloseRemoteTable(  );

    return err;

}

long CSyncFishDB::PCToHHSync()
{
    long err = -1;
    CFishRecObj* pFishRecObj = NULL;

    //open the remote table
    err = m_pSyncUtility->OpenRemoteTable( m_pszRemoteName );
    if ( !err )
    {
       BYTE hRemoteDB = m_pSyncUtility->GetRemoteDBHandle();
       // Delete all Remote (Handheld) records
       err = SyncPurgeAllRecs( hRemoteDB );
    }

    //bailout if error occurred
    if ( err )
    {
       m_pSyncUtility->CloseRemoteTable(  );
       return err;
    }

    // For each PC record ...
    pFishRecObj = m_pFishList->GetFirstFish( );
    while ( pFishRecObj && !err )
    {
       err = m_pSyncUtility->AddRemoteRecord( pFishRecObj );
       pFishRecObj = m_pFishList->GetNextFish ( );
    }
```

```
   m_pSyncUtility->CloseRemoteTable(  );

   return err;
}
```

After class instantiation, the `CSyncFishDB`'s `Sync` member is called. `Sync`'s job is to kick off the actual database synchronization, but it also needs to register SyncEx with the HotSync Manager in order to be able to perform the sync. To do that, it calls `SyncRegisterConduit`.

```
// Sync
// Performs synchronization of FishDB between PC and Palm

long CSyncFishDB::Sync()
{
   CONDHANDLE   conduitHandle = (CONDHANDLE)0;
   long retval = SYNCERR_UNKNOWN;
   char      appName[] = "SyncEx";

   // Register this conduit with SyncMgr.DLL for communication to HH
   if (retval = SyncRegisterConduit(conduitHandle))
      return(retval);

   //perform the sync
   retval = SyncDB();

   // application name
   LogAddEntry(appName, slSyncFinished,FALSE);

   // clean up
   SyncUnRegisterConduit(conduitHandle);

   return retval;
}
```

The `SyncDB` member function is where SyncEx decides which type of synchronization activity to perform. This decision is driven by the default sync action chosen by the user in SyncEx's conduit configuration dialog, and also by the HotSync Manager in its assessment of the situation between the PC and the device.

There are five choices, and the first two—Fast and Slow—require some explanation. These are both the result of choosing the sync action that says to synchronize both the PC and the Palm. HotSync tries to determine whether it can be smart about the synchronization process based on the fact that the same Palm is syncing with the same PC as last time, in which case it can perform a Fast Sync, relying on the modification flags only. If HotSync determines that it cannot rely on this information (perhaps because the Palm device is synching with a different PC), it will call for a Slow Sync, meaning each record will be examined one by one.

The other sync actions correspond to the choices on the configuration dialog's screen: Handheld Overwrites PC, PC Overwrites Handheld, and Do Nothing.

`SyncDB` takes the recommended action and calls the corresponding member function based on the sync type called for by HotSync Manager. We will next look closely at what happens during the most common action, a Fast Sync.

In a Fast Sync, `CSyncFishDB` needs to in turn look for changes made on the handheld and sync them to the PC, and then it looks for changes made on the PC and syncs them to the handheld. These tasks are performed in two different member functions.

`SyncHHChanges` makes use of `CsyncUtility`, `CfishRecObj`, and the standard HotSync API to iterate through each handheld record it sees as modified, reads it into a `CrawRecordInfo` structure, and takes the appropriate action on the PC based on the attributes of the record. For example, if it sees that the "dirty" flag has been set on the handheld record, it calls `UpdateFishRecOnPC` to either add or update the record on the PC.

Moving along, `UpdateFishRecOnPC` must find the matching PC record in fish.txt. If it finds it, it simply copies the handheld record over the PC record. If it is not found, the record is assumed to be new, and will be added to fish.txt. Note that this is an example of "record-level synchronization." A more complex option would be to attempt field-level synchronization, in which a conduit attempts to determine which fields in a record have changed.

After all handheld records have been iterated, and the PC records have been added, updated, or deleted, any records marked for deletion on the handheld are physically purged, and the remote database is closed.

`SyncPCChanges` performs the same task, only in the opposite direction. Here you make use of the `FishList` class to iterate through all PC records, and you use the transaction code field to determine whether it's an add, update, or delete operation. You also utilize `CSyncUtility` to easily perform the appropriate operation on the handheld database, using the record ID as a unique key for update and delete operations.

The source for `CsyncUtility` is listed here:

```
/*
CSyncUtility - Base class for DB syncs
Copyright © 2002 Bachmann Software and Services, LLC
```

```
*/
#include "stdafx.h"
#include <syncmgr.h>
#include "SyncUtility.h"

#define RAW_REC_MEM  10240

CSyncUtility::CSyncUtility( CSyncProperties& rProps )
{
   memcpy(&m_rSyncProperties, &rProps, sizeof(m_rSyncProperties));
   m_pRawRecord = new CRawRecordInfo;

   // Allocate memory for m_pRawRecord->m_pBytes
   AllocateRawRecordMemory( m_pRawRecord );
}

CSyncUtility::~CSyncUtility()
{
   if ( m_pRawRecord->m_TotalBytes > 0 && m_pRawRecord->m_pBytes )
   {
      delete m_pRawRecord->m_pBytes;
   }
   delete m_pRawRecord;
}

long CSyncUtility::AddRemoteRecord( CBaseRecObj* pRecObj )
{
   long err = -1;

   ResetRawRecord( );
   m_pRawRecord->m_FileHandle = m_hRemoteDB;  // remote file handle

   // Convert record data for remote, upon return grab new RecordId.
   err = pRecObj->ConvertToRemote( *m_pRawRecord );
   if ( !err )
   {
      m_pRawRecord->m_RecId = 0 ; // remote HH will assign new RecId
      if ( !( err = SyncWriteRec( *m_pRawRecord ) ) )
      {
         pRecObj->SetRecordID( m_pRawRecord->m_RecId );
      }
   }
   return err;
}

long CSyncUtility::ChangeRemoteRecord( CBaseRecObj* pRecObj )
{
   long            err = -1;
   int            locRecId;

   locRecId = pRecObj->GetRecordID( );

   ResetRawRecord( );
   m_pRawRecord->m_FileHandle = m_hRemoteDB;  // remote file handle
   m_pRawRecord->m_RecId      = (DWORD)locRecId;  // remote HH record already
exists

   // Prepare record data for remote
   err = pRecObj->ConvertToRemote( *m_pRawRecord );
   if ( !err )
   {
      err = SyncWriteRec( *m_pRawRecord );
   }

   return err;
}

long CSyncUtility::DeleteRemoteRecord( CBaseRecObj* pRecObj )
{
   long            err = -1;
   int            locRecId;

   locRecId = pRecObj->GetRecordID( );

   ResetRawRecord( );
   m_pRawRecord->m_FileHandle = m_hRemoteDB;  // remote file handle
   m_pRawRecord->m_RecId      = (DWORD)locRecId;  // remote HH record already
exists

   // delete the record
   err = SyncDeleteRec( *m_pRawRecord );

   return err;
}
```

```
long CSyncUtility::AllocateRawRecordMemory( CRawRecordInfo* pRawRecord )
{
   long retval = -1;
   WORD wRawRecSize = RAW_REC_MEM;

   // Allocate memory for rawRecord data
   pRawRecord->m_TotalBytes = 0;
   pRawRecord->m_pBytes = (BYTE*) new char [wRawRecSize];
   if (pRawRecord->m_pBytes)
   {
      pRawRecord->m_TotalBytes = wRawRecSize;
      memset(pRawRecord->m_pBytes, 0, wRawRecSize);
      retval = 0;
   }
   return retval;
}

long CSyncUtility::OpenRemoteTable( char* pszRemoteName )
{
   long err = -1;

   // Call into SyncManager.DLL to open Remote data base
   err = SyncOpenDB( pszRemoteName, 0, m_hRemoteDB );

   // Create the remote dataBase if it's not there
   if ( err == SYNCERR_FILE_NOT_FOUND &&
            m_rSyncProperties.m_SyncType != eHHtoPC)
   {
      CDbCreateDB dbInfo;
      memset(&dbInfo, 0, sizeof(dbInfo));
      dbInfo.m_Creator  = m_rSyncProperties.m_Creator;
      dbInfo.m_Flags    = eRecord;
      dbInfo.m_CardNo   = (BYTE)m_rSyncProperties.m_CardNo;
      dbInfo.m_Type     = 'Data';
      strcat( dbInfo.m_Name, pszRemoteName );

      if (!(err = SyncCreateDB(dbInfo)))
      {
         m_hRemoteDB = dbInfo.m_FileHandle;
      }
   }
   // Close RemoteDB if an error occurred
   if ( err )
   {
      SyncCloseDB(m_hRemoteDB);
   }

   return err;
}

void CSyncUtility::CloseRemoteTable(  )
{
   // Close RemoteDB
   if ( m_hRemoteDB )
   {
      SyncCloseDB(m_hRemoteDB);
      m_hRemoteDB = 0;
   }
}

BOOL CSyncUtility::HasHHDatabaseChanged( char* pszRemoteName )
{
   BOOL bHasChanged = FALSE;
   long         err = 0;

   //open the remote table
   err = OpenRemoteTable( pszRemoteName );
   if ( err )
   {
      return bHasChanged;
   }

   ResetRawRecord( );
   m_pRawRecord->m_FileHandle = m_hRemoteDB;  // remote file handle

   err = SyncReadNextModifiedRec ( *m_pRawRecord );

   bHasChanged = ( err == SYNCERR_NOT_FOUND ) ? FALSE : TRUE;

   CloseRemoteTable(  );

   return bHasChanged;
}
```

```
void CSyncUtility::ResetRawRecord( )
{
   m_pRawRecord->m_FileHandle = NULL;
   m_pRawRecord->m_RecId = 0;
   m_pRawRecord->m_RecIndex = 0;
   m_pRawRecord->m_Attribs = 0;
   m_pRawRecord->m_CatId = 0;
   m_pRawRecord->m_ConduitId = 0;
   m_pRawRecord->m_TotalBytes = RAW_REC_MEM;
   m_pRawRecord->m_RecSize = 0;
   memset( m_pRawRecord->m_pBytes, 0, RAW_REC_MEM );
}
```

As you can see, the main operations for SyncEx are as follows:

1. Determine the type of synchronization necessary.
2. Iterate through either the PC or handheld database (or both).
3. For each add, update, or delete in the iterated database, make the corresponding change to the remote or PC database.

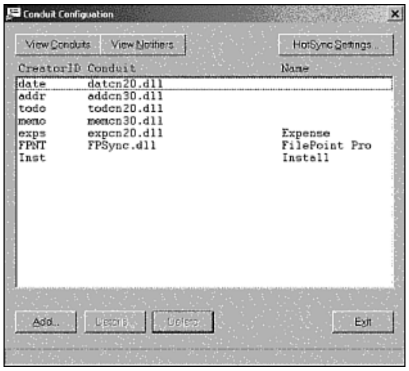
Certainly there are far more complex synchronization scenarios than the one presented here, so I don't want to oversimplify. But most of the mystery in conduit development is in dealing with the strange and different APIs, tools, environment, and program structure, not the actual sync operation.

Registering SyncEx with CondCfg

Once you are ready to test and run your conduit, you need to register your conduit so that it can be recognized by HotSync Manager. Be careful; this is a different kind of registration than what `SyncRegisterConduit performs`. `SyncRegisterConduit` registers your conduit at runtime so that it can proceed with synchronization. Another kind of registration must be performed so that HotSync Manager can even find your conduit in the first place.

In order to know which conduits are installed on a PC, which Palm applications they are associated with, and what to call them in the Conduit Configuration dialog, each conduit must be described in the Windows Registry. Rather than make the poor programmer manually insert all of the appropriate keys, a utility called `CondCfg` comes with the CDK for just this very purpose.

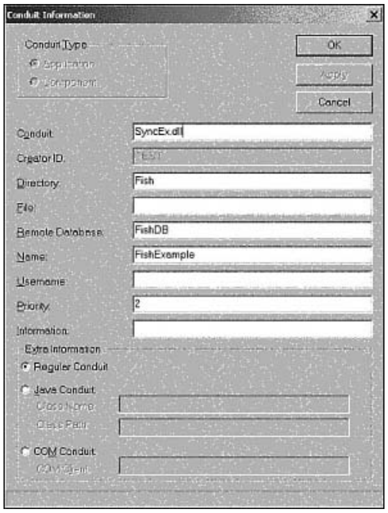
CondCfg.exe is found in your CDK installation, under `\Common\Bin`. You should copy CondCfg.exe to your main Palm directory, which is where all conduits should be installed. If you run CondCfg (see Figure 28.2), it lists all currently registered conduits on the PC.



```
CreatorID Conduit                   Name
date      datcn20.dll
addr      addcn30.dll
todo      todcn20.dll
memo      memcn30.dll
exps      expcn20.dll               Expense
FPNT      FPSync.dll                FilePoint Pro
Inst                                Install
```

FIGURE 28.2 CondCfg.exe's main screen.

To register your conduit, you should first make sure you've copied your conduit's DLL (in this case, SyncEx.dll) to the Palm main directory. Then click Add on the main screen for `CondCfg`, and you will see the configuration screen pictured in Figure 28.3.



Conduit: SyncEx.dll
Creator ID: TEST
Directory: Fish
File:
Remote Database: FishDB
Name: FishExample
Username:
Priority: 2
Information:
[✓ Regular Conduit]

FIGURE 28.3 Configuring SyncEx with CondCfg.exe.

There are really just a few essential fields that you need to fill out, as follows:

- **Conduit:** This is the name of your conduit DLL.
- **Creator ID:** This is the Palm Creator ID associated with your conduit's corresponding Palm application.

- **Directory:** This is the name of the directory under the user's Palm directory where the databases will be found. In this case, this would be `\Palm\BachmannG\Fish`. You need only specify the directory path relative to the HotSync username.
- **Remote Database:** This is the name of the Palm database that will be synced.
- **Name:** This is the descriptive name that will appear when you run the HotSync Manager and click the Custom menu to view all conduits.

When you are done, simply click OK (or Apply), and your conduit will be registered in the Windows Registry. A good way to test this is to right-click the HotSync Manager on your system tray, and choose Custom. If your conduit is registered properly, it will appear in the resulting conduit list.

Summary

Although you've explored the basics of conduit development in this chapter, I assure you that you have just scratched the surface. With conduit programming, it is not the number of APIs that is overwhelming, nor is the subject matter technically challenging. The hardest parts of conduit programming are making the right choices in terms of tools and framework, and also in dealing with the myriad synchronization scenarios that can occur in the real world.

Beyond this introduction, I recommend carefully reviewing and understand the CDK documentation and examples. If possible, seek the guidance of someone who has real experience in conduit development, who can pass on their knowledge and save you the headaches of learning them for the first time.

If you do pursue and eventually master conduit development as a discipline, it is an achievement to be proud of, there are few among us who have done so.

29 Shared Libraries: Extending the Palm OS

"Assembly of Japanese bicycle requires great peace of mind."

Robert Pirsig, *Zen and the Art of Motorcycle Maintenance*

Encapsulating Functionality: Shared Libraries

Packaging your code into shared libraries provides many benefits. Libraries provide a mechanism for reusing your code among two or more applications, thereby reducing development time. Multiple applications can use the same library, reducing both the installed and in-memory footprint. The lifetime of a shared library is not strictly tied to that of its clients, so its state can be maintained across applications. Architecturally, shared libraries reduce coupling between modules; reduced coupling results in more stable and maintainable code.

Although there is more information now available on how to create shared libraries in the Palm documentation, to many developers this remains a deep dark mystical area of Palm programming. Certainly they lack some of the familiar attributes of "normal" Palm programs, such as a user interface, but for the most part the concepts remain the same. As I hope you'll see by the end of this chapter, shared libraries are not nearly as complicated as they seem.

In this chapter you'll learn about

- The Palm shared library model
- The components of a shared library

- How to use Palm development tools to create shared libraries
- The development effort required for successfully creating a shared library

The Palm Shared Library Model

A shared library is a Palm database whose type is `libr` and that can be loaded and used by client applications. Shared libraries can be used by one or more clients. These clients might be applications or other shared libraries. The Palm OS comes with several shared libraries pre-loaded into ROM: `SerialMgr` and `IrMgr` are examples.

From the client side, the use of a shared library is straightforward. First the client retrieves the reference number for the shared library. This can be obtained either from `SysLibLoad` or `SysLibFind` by passing a pointer to an integer, which will receive the reference number upon success. If the library is being referenced for the first time, `SysLibLoad` then loads the library. If the library is already loaded, `SysLibLoad` returns an error, in which case `SysLibFind` returns the reference number of the loaded library. `SerialMgr` and `IrMgr` are used by the OS, so they are always loaded.

The following code snippet illustrates how this is typically done using a fictional `myLibrary` as the target shared library to be loaded:

```
UInt16 refNum = 0;
Err err = SysLibLoad (myLibraryTypeID, myLibraryCreatorID, &refNum);

if (!err)
{
   err = MyLibraryOpen (refNum);
   if (!err)
   {
      prvRefNum = refNum;
   }
   else
   {
      SysLibRemove (refNum);
   }
```

After the client obtains a reference number, it calls the shared library's functions, starting with the `Open` function. Under the Palm OS, inter-module function calls are resolved by a trap identifier rather than an address. The system uses the library reference number to multiplex calls into shared libraries. All shared library functions must take this reference number as their first parameter.

When the client is finished with the shared library, the client might need to remove it from memory using SysLibRemove. It is recommended that the library should return success (zero) if it can be unloaded, or an error (a nonzero value) if it still has active clients.

Traps and the Palm Software Development Kit

If you examine any Palm OS header file, you'll notice that every function is followed by a SYS_TRAP macro. The meaning of the SYS_TRAP macro will be discussed in depth later; for now it is sufficient to understand that every Palm OS SDK function is mapped by the compiler into the appropriate trap invocation. It is the job of the shared library developer to provide the appropriate mapping from function call to system trap and back to function call.

The SDK trap identifiers are unique throughout the operating system. Shared library function trap identifiers are unique only within the library. The header System\SysTraps.h lists the trap identifier for every SDK function. These are defined as an enumeration beginning with sysTrapBase (0xA000). This header also defines sysLibTrapBase (0xA800), which is the starting value for a shared library's trap identifiers. There are six predefined identifiers: sysLibTrapName, sysLibTrapOpen, sysLibTrapClose, sysLibTrapSleep, sysLibTrapWake, and sysLibTrapCustom. The specific uses of each of these will be described later. For now, what is important is that your custom function trap identifiers must start with sysLibTrapCustom and increment by one.

Services Provided by Palm OS for Shared Libraries

The system provides the three services for shared libraries: the system trap dispatcher, the system library table, and the library reference number.

The system trap dispatcher dispatches calls by library reference number to the appropriate library. When a client invokes a function by trap identifier, it generates an exception. The trap dispatcher handles this exception by putting the address of the appropriate function onto the stack. When the exception returns, program execution is transferred to this address.

For each shared library open in the system, there is a corresponding entry in the system library table. This entry holds both the library's dispatch table and the memory allocated by the library for its global memory. These items are discussed in more detail in the next section.

The library reference number is assigned by the system when the library is loaded for the first time. The client retrieves this number either from SysLibLoad, which loads the library, or SysLibFind, if the library is already loaded. This reference number must be the first parameter to every shared library call; the system trap dispatcher uses it to locate the correct library.

What the Shared Library Must Provide

The shared library provides an installation entry point and the library's dispatch table. It also implements domain-specific functionality.

The installation entry point is named `__Startup__`, and it must be the first function in the library's link order. It returns `0` for success or a negative error code. `SysLibLoad` returns the error code to the client application. The system passes the library's entry in the system library table as a parameter to this function; the library sets the dispatch table.

The library dispatch table contains a list of all routines in the library. This is a lookup table used by the system trap dispatcher, and it is described in great detail in the following sections.

Implementing a Shared Library

Three components are in a shared library: the API declaration, the API implementation, and the dispatch table. The API declarations specify how clients use the shared library. The API implementation provides this functionality. The dispatch table maps the declarations to the implementation through the system trap mechanism.

The API Declaration

The shared library publishes a header file describing the API to clients. Function declarations specify how the client invokes the API, including parameter information and return types. Trap identifiers allow the system to invoke the function using the trap mechanism on which the Palm OS relies. Result codes provide an expected set of errors for which the client should check. Domain-specific structures and constants provide additional information required by the custom portion of the API.

Function Declarations

Every shared library must publish four standard functions, including

- `Open`—The client must call this function first. It allows the shared library to initialize any resources it needs. No other API functions can be called prior to this function.
- `Close`—The client calls this function last. It allows the shared library to release any resources it is holding. When this function is called, the library is in an invalid state; it must be reopened before it can be used.
- `Sleep`—The operating system calls this function before the device enters sleep mode. It allows system-level libraries to shut down hardware components to conserve power.

- `Wake`—The operating system calls this function when the device returns from sleep mode. It allows system-level libraries to re-enable any hardware components that were shut down when the device entered sleep mode.

The shared library also publishes domain-specific functions. As a rule, these functions must be invoked after `Open` and before `Close`, to make sure that the library has valid resources. The one exception is a function that retrieves the API version of the library, which might be invoked before `Open` to ensure compatibility.

Function Trap Identifiers

If you examine the Palm OS SDK headers, you'll notice that every function declaration includes the `SYS_TRAP` macro:

```
Err SerOpen(UInt refNum, UInt port, ULong baud)
                SYS_TRAP(sysLibTrapOpen);
```

For the CodeWarrior compiler, this declaration expands to a Metrowerks extension called *opcode inline* syntax:

```
Err SerOpen(UInt refNum, UInt port, ULong baud)
= {m68kTrapInstr + sysDispatchTrapNum, trapNum}
```

Opcode inline syntax allows you to specify the 68k opcodes for the function's implementation. When you call an opcode inline function, the compiler replaces the function calls with the specified opcodes. This feature supports calls through the 68k processor's *A-Trap* mechanism; it generates the exception that is handled by the Palm OS's system trap dispatcher.

Invoking a call to a shared library function is a two-step process. First, for client code, the function declaration expands into a call to the A-Trap mechanism. This mechanism uses the library's reference number to identify the appropriate entry in the system library table; the library's reference number must be the first parameter to any shared library function. Then, using the trap identifier as an offset into the entry's dispatch table, the appropriate library function is called.

If this all seems like an awful lot of work, just remember that the compiler takes care of it; your job is to declare the trap ID enumeration and make sure that you use the right trap for each function declaration.

The system defines traps for the required `Open`, `Close`, `Sleep`, and `Wake` functions. You define the rest, starting with `sysLibTrapCustom` and incrementing sequentially.

Errors

The public API also contains the errors that the functions might return. These are based on the `appErrorClass`, which is defined in `SystemMgr.h`. The library functions should not return any results that are not either a success or one of the API-defined errors.

Structures and Constants

In addition to the functional API, there might be structures or constants defined by the library. For example, a printing library might publish a font structure and constants for bold, italic, or underline.

API Implementation

The functions described in the public header are implemented in a standard C file. In this module, the `SYS_TRAP` macro is disabled so that the function declarations evaluate to standard C declarations.

The `Open` Function

The `Open` function is responsible for allocating memory for this information and storing it in the system library table entry for the shared library. After this is done, it performs any domain-specific initialization.

The `Close` Function

The `Close` function is responsible for releasing memory allocated for the library's globals and removing this value from the system library table entry. Any domain-specific cleanup should be done prior to releasing this memory. By convention, this function returns `0` if the library should be removed, or an error code indicating that the library is still in use by other clients.

The `Sleep` and `Wake` Functions

The `Sleep` function handles notification from the system that the system is about to shut down. This notification allows system-level libraries to shut down hardware components in order to conserve power.

The `Wake` function handles notification from the system that the system is about to wake up again. This notification allows system-level libraries to re-enable hardware components that were shut down when the system went to sleep.

Because these functions are invoked by system interrupts, they might only use interrupt-safe system services, and must not take a long time.

Custom Functions

Domain-specific API functions depend on the global data allocated in the `Open` function, and so should only be invoked between the `Open` and `Close` functions. These functions follow a standard pattern. First, they retrieve the library's global data. Then they perform the domain-specific task they provide. Finally, they release the global data.

There are exceptions to this rule. For example, many libraries provide a function to retrieve the API version of the library. Because this function does not rely on any global information, it might be called before the `Open` or after the `Close` function. Indeed, it probably should be called prior to `Open` to ensure compatibility between the client and the library.

Dispatch Table Implementation

The dispatch table implementation is provided by two functions: `Install` and `DispatchTable`. The system invokes the `Install` function as part of `SysLibLoad`; this function invokes the `DispatchTable` function, which returns the address of the dispatch table. This address is stored in the library's entry in the system library table. `DispatchTable` is coded in Assembler and includes the declaration of the dispatch table, in addition to the code that returns its address.

The first section of the dispatch table is an array of offsets from the beginning of the dispatch table. The first offset points to the library name. This string is stored at the very end of the table. The remaining offsets point to entries in the next section, which is an array of jump instructions. The second offset points to a jump to the `Open` function; the third, a jump to the `Close` function, and so on. The library name is stored as a null-terminated string at the end of the jump table.

Let's say, for example, that you had a shared library named `FooLibrary`, with only one function—`Foo`. The dispatch table would look like Table 29.1.

TABLE 29.1 The Dispatch Table for the `FooLibrary` Shared Library

Section	Address	Contents	Comments
Offsets	0	20	Address of name string
	2	12	Offset to Open jump
	4	14	Offset to Close jump
	6	16	Offset to Sleep jump
	8	18	Offset to Wake jump
	10	20	Offset to Foo jump
Jumps	12	JMP Open	Jump to Open
	14	JMP Close	Jump to Close
	16	JMP Sleep	Jump to Sleep
	18	JMP Wake	Jump to Wake
	20	JMP	Jump to Foo
Lib Name	22	FooLibrary	Library name

The system executes library traps using this table. Using the trap identifier as an index into the first section of the dispatch table (see Figure 29.1), it retrieves the offset to the jump instruction corresponding to the trap. The system moves to this offset in the dispatch table and executes the next instruction. This instruction is a jump to the appropriate shared library function.

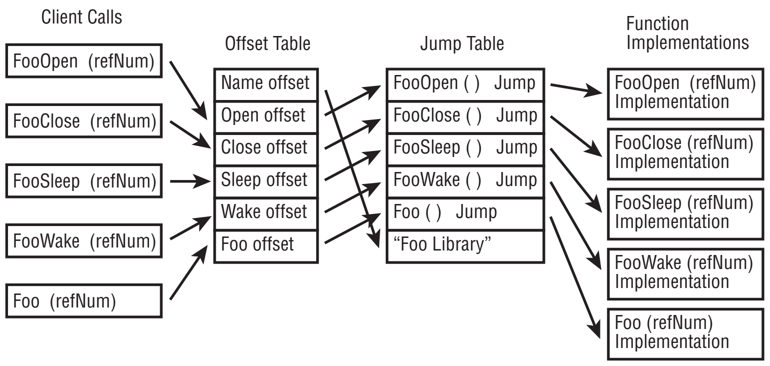


FIGURE 29.1 A conceptual model of a dispatch table.

Conceptually, this is no different from the implementation of virtual tables under C++. The client invokes a function that is mapped to an offset in a function table. Using this offset, the compiler retrieves the address of the function to execute. The main difference is that in C++ the compiler takes care of all this plumbing transparently; with shared libraries, we must do it ourselves.

More Rules of the Road

Just in case shared libraries weren't daunting enough, there are some rules and caveats of which you should be aware.

Do Not Use Static and Global Values

Shared libraries are stored on the device as resource databases. The memory occupied by the library is, therefore, on the storage heap. By default, this memory is protected; it can only be written to with the appropriate `DataMgr` function. A downstream effect of this is that all global or static values are, effectively, read-only.

You could use static or global variables in your shared library. But if you do, you need to allocate a writeable data segment for them, and initialize this data segment by hand. When you enter a library function, you need to set the A4 register to point to this data segment. When you exit the function, you need to set it back to its original value. You also need to do this "A4 magic" before calling certain SDK functions, such as `MemHandleNew`. In general, it's best not to use them at all.

Install Linkage

A shared library's Install function is actually a macro for the library entry point, `__Startup__`. The system expects this function to be the first entry point into the library. This function must therefore be first in the linkage order. I put it in its own module, and list that module first in the Segments panel of the project.

Library Resources

Palm application resources are opened with the application at all times. If you want to show a form or raise an alert, it is a straightforward task. Resources associated with a shared library are not left opened with the library; the library must specifically load its resource database. Consequently, it must unload the database as well.

Debugging

Bugs are a fact of life in software development. As a result, so are debuggers. Unfortunately, the CodeWarrior debugger does not currently support tracing into shared library code. This support is rumored to be coming in an upcoming release. Until this support is available, you must embed code in your library to display errors, variable values, and so on. You can use alerts, error messages, or the emulator logging facility. On some projects, I have created a debug message console on the Palm device. A more detailed discussion of debugging without a visual debugger is beyond the scope of this chapter.

Example—`ShrLib`—A Simple Shared Library

The chapter example is simple but it does illustrate the basics of shared libraries. The goal of the example project is rather modest: allow the client application to get and set a text string that resides in a shared library. In addition to the standard `Open`, `Close`, `Sleep`, and `Wake`, the example provides functions to determine the library's API version and to get and set the text string.

The Project File—`ShrLib.mcp`

You can use the project file and directory structure as a basis for your shared library development. There is no stationery for these projects, so I started with the Palm OS C application stationery and modified it, as shown in Table 29.2.

TABLE 29.2 Target Settings Modified from the Palm OS C Application Stationery

Panel	Setting	Value
Target Settings	Target Name	`ShrLib`
Access Paths	Always Search User Paths	`Checked`
68k Target	Project Type	`PalmOS Code Resource`
	File Name	`ShrLib.tmp`
	ResType	`libr`
PilotRez	Mac Resource Files	`ShrLib.tmp`
	Creator	`SHRL`
	Type	`libr`

Note that the 68k Target—File Name setting must match the PilotRez—Mac Resource Files setting. Also, the shared library's creator ID must be registered with Palm to avoid conflicts with other applications or libraries.

The Public Header—`ShrLib.h`

The header, `ShrLib.h`, includes the declarations for the library used by client applications. Our library publishes seven functions. Four of these are the standard functions every library must publish: `ShrOpen`, `ShrClose`, `ShrSleep`, and `ShrWake`. Additionally, we publish one function to determine the library's version, `ShrAPIVersion`. The last two functions, `ShrGetText` and `ShrSetText`, are custom functions that allow client applications to retrieve and modify the shared library's text string.

Trap Handling

The first thing you'll notice in this file is the definition of the `SHRLIB_TRAP` macro:

```
#ifdef BUILDING_LIBRARY
#define SHRLIB_TRAP(trapNum)
#else
#define SHRLIB_TRAP(trapNum) SYS_TRAP(trapNum)
#endif
```

The `SHRLIB_TRAP` macro is used in declaring the API functions. If the `BUILDING_SHRLIB` macro is defined, this macro does nothing and the functions are declared as normal C function calls. Otherwise, `BUILDING_SHRLIB` expands to the SDK `SYS_TRAP` macro, and the functions are declared using the Palm OS trap mechanism. The only place where the `BUILDING_SHRLIB` macro should be defined is in the implementation of the API functions; source files that want to invoke these functions should do so by the trap mechanism.

Library Identifiers

Clients identify a shared library in two ways. To load the library, the client needs creator and library type identifiers:

```
#define shrCreatorID 'SHRL'
#define shrTypeID  'libr'
```

These values are passed to `SysLibLoad`, and must match the respective project settings. Creator must match the PilotRez—Creator. Type must match both 68k Target—ResType and PilotRez—Type.

To find a library that has already been loaded, the client needs the library's name:

```
#define shrName      "Shared Library"
```

The client passes the library name to SysLibFind, which compares it to the names of the currently loaded libraries. This constant is also used internally in the dispatch table.

Error Declarations

This example defines three errors:

```
#define shrErrAlreadyOpen    (appErrorClass | 1)
#define shrErrParam         (appErrorClass | 2)
#define shrErrMemory        (appErrorClass | 3)
```

shrErrAlreadyOpen is returned if the client attempts to open a library that is already opened. This library does not support multiple clients.

shrErrParam is returned when at least one of the function parameters is invalid. An example of this would be if a NULL pointer were passed as an out-param.

shrErrMemory is returned when the allocation of memory or another resource fails.

Function Trap Identifiers

For each of the three custom functions published by my library, I define the corresponding trap identifier:

```
typedef enum {
   shrTrapAPIVersion = sysLibTrapCustom,
   shrTrapGetText,
   shrTrapSetText
} ShrTrapNumberEnum;
```

These are declared as an enumeration to ensure sequential values. Notice that the trap enumeration begins with the SDK constant sysLibTrapCustom.

Function Declarations

I provide function declarations for both standard and custom functions. I include declarations for Sleep and Wake for completeness; they are only ever called by the operating system and must conform to a standard declaration:

```
// Standard Library Functions:
extern Err ShrOpen(UInt refNum)
                SHRLIB_TRAP(sysLibTrapOpen);

extern Err ShrClose(UInt refNum)
                SHRLIB_TRAP(sysLibTrapClose);

extern Err ShrSleep(UInt refNum)
```

```
                SHRLIB_TRAP(sysLibTrapSleep);

extern Err ShrWake(UInt refNum)
                SHRLIB_TRAP(sysLibTrapWake);

// Get our library API version
extern Err ShrAPIVersion(UInt refNum, DWordPtr dwVerP)
                SHRLIB_TRAP(shrTrapAPIVersion);

// Retrieve the text stored by the library.
extern Err ShrGetText(UInt refNum, CharPtr string, UInt size)
                SHRLIB_TRAP(shrTrapGetText);

// Set the text stored by the library.
extern Err ShrSetText(UInt refNum, CharPtr string)
                SHRLIB_TRAP(shrTrapSetText);
```

All functions must take the library reference number as their first parameter. `ShrSleep` and `ShrWake` are invoked only by the operating system, so they must take only the reference number parameter. The trap mechanism returns an integer, so all library functions should do so as well.

These declarations are wrapped with macros that check if the header is being included in a C++ source file. If so, these functions are declared as `extern C`. This prevents function name mangling by the C++ compiler.

API Implementation—`ShrLib.c`

Now that you've declared your API to the public, it's time to implement it. You do this in `ShrLib.c`. Before you implement these functions, though, there are some supporting declarations you need to make.

Library Globals

As a rule, global or static variables are prohibited in a shared library. There are techniques to get around this prohibition, but they are best avoided. Instead, values that must persist between function calls are stored in a structure:

```
#define shrMaxText   (99)

struct ShrGlobalsType {
   UInt  refNum;
   Char  text [shrMaxText+1];
   };
```

This structure is stored in the system library table entry associated with the loaded library. The refNum attribute is used to store the shared library's reference number; this is the number returned to the client by SysLibLoad or SysLibFind.

The text attribute is used to store the string managed by this shared library. In your shared library, you would replace this attribute with domain-specific attributes.

ShrOpen

ShrOpen allocates the globals structure and associates it with the library in the system library table. First you retrieve a pointer to the system library table entry for the library:

```
// Retrieve the system library table entry.
   SysLibTblEntryPtr libEntryP = NULL;
   libEntryP = SysLibTblEntry(refNum);
```

Next you check the entry's globalsP attribute. If this attribute is not NULL, the library is already in use by another client and the function returns shrErrAlreadyOpen:

```
// Make sure the globals have not been initialized.
   if (NULL != libEntryP->globalsP)
   {
      // Globals already initialized.
      // Library is open for another client.
      return shrErrAlreadyOpen;
   }
```

Otherwise, the function allocates a block of memory to hold the globals structure. Set the owner of this memory to 0 (the system) using MemHandleSetOwner. This allows the library to maintain state between applications; if you didn't, the memory would be freed when the calling application exited. Even though your library will only support a single client application, the client application might start and stop as the user switches to other applications.

```
   VoidHand globalsH = NULL;

   // Allocate enough memory to hold data.
   // Note: We allocate a memory handle rather than a
   //       (locked) pointer so that the memory manager
   //       can move the block around. Remember, this
   //       library may be around for a long time.
   globalsH = MemHandleNew (sizeof (ShrGlobalsType));
   // Set owner of global data memory block to "system"
   // (zero).
   // Note: If the application were to remain the owner
   //       of the memory, the memory would be freed
   //       when the application exited.  For the library
   //       to maintain state between applications, this
   //       must not happen.  Any other allocated memory
   //       blocks should likewise be adopted by the system.
    MemHandleSetOwner(globalsH, 0);
```

When the memory is allocated, you store its handle in the library's system table entry. Use movable memory to minimize dynamic heap fragmentation, so you have to cast the handle to a pointer, as expected by the entry.

```
   // Save the handle of our library globals in the
   // system library table entry so we can later
   // retrieve it using SysLibTblEntry().
   // Note: lib entry structure expects a void pointer,
   //       so we must cast our handle accordingly.
    libEntryP->globalsP = (void*) globalsH;
```

Finally, initialize the values of structure. Lock the handle, set the reference number and text attributes, and then unlock the handle again.

```
ShrGlobalsType* globalsP = NULL;

// Lock the memory and cast it to a ShrGlobalsType pointer.
    globalsP = (ShrGlobalsPtr)MemHandleLock(globalsH);

   // Begin library-specific open functionality.

   // Initialize global data's reference number.
    globalsP->refNum = refNum;

   // Initialize global data's text buffer.
   StrCopy (globalsP->text,
           "Shared Library Example Text");

   // End library-specific open functionality.

   // Unlock globals.
   MemHandleUnlock (globalsH);
```

To optimize system performance, the memory should be locked only when needed, and unlocked as soon as possible. This allows the memory manager to move the block around as necessary.

ShrClose

Closing the library is fairly straightforward. You retrieve the global data handle and perform any domain-specific close functionality (in this case, there is none). Then you free the global data memory and set the library entry's globalsP attribute to NULL.

```
// Retrieve library system table entry.
   SysLibTblEntryPtr libEntryP = SysLibTblEntry(refNum);

   // Retrieve the handle of our library globals from the
   // system table entry.
   VoidHand globalsH = (VoidHand)(libEntryP->globalsP);

   // Begin library-specific close functionality.

   // End library-specific close functionality.

   // Clear library system table entry's global data.
   libEntryP->globalsP = NULL;

   // Free global data memory.
   MemHandleFree(globalsH);
```

ShrSleep **and** ShrWake

The implementations of ShrSleep and ShrWake are trivial; they simply return success. You have no hardware to disable or enable.

ShrAPIVersion

ShrAPIVersion returns the version number for the library. This version number is calculated according to the scheme defined in SystemMgr.h.

Version numbers are formatted as 0xMMmfsbb, where:

MM	Major version of library
m	Minor version of library
f	Bug fix flag
s	Stage of version (see stages, which follow)
bbb	Build number (for non-releases)

Stages of development are defined as:

0	Development build
1	Alpha build
2	Beta build
3	Release build

Examples of this scheme include:

`v1.2 Dev build 12`	`0x01200000C`
`v1.1 Alpha 2`	`0x011001002`
`v1.1 Beta 3`	`0x011002003`
`v1.0 Release`	`0x010003000`

The Palm SDK provides a set of macros for both creating and decoding versions. The `sysMakeROMVersion` takes the major version, minor version, bug fix flag, stage, and build number and creates a `DWord` representing the version number. The macros `sysGetROMVerMajor`, `sysGetROMVerMinor`, `sysGetROMVerFix`, `sysGetROMVerStage`, and `sysGetROMVerBuild` extract the respective values from a version number. These macros are defined in `SystemMgr.h`.

`ShrGetText` and `ShrSetText`

The implementations for custom functions are fairly boiler-plate. The function retrieves a pointer to the library's system library table entry and locks the globals memory:

```
// Retrieve library system table entry.
SysLibTblEntryPtr libEntryP = SysLibTblEntry(refNum);

// Retrieve the handle of our library globals
// from the library system
// table entry.
VoidHand globalsH = (VoidHand)(libEntryP->globalsP);

// Lock the memory and cast it to a ShrGlobalsType.
ShrGlobalsType* globalsP =
      (ShrGlobalsType*)MemHandleLock(globalsH);
```

The functions do their thing; then unlock the globals and return.

Dispatch Table Implementation—ShrDisp.c

The dispatch table is the heart of a shared library. It provides the mapping from the library's trap identifiers to the appropriate function implementations. This is not for the faint of heart; creating the dispatch table involves a fair amount of hand-coding assembler. You might want to take a deep breath before you dive in.

Declarations

Before we explore the implementation of the dispatch table, there are a few declarations in ShrDisp.c that we should examine.

As in ShrLib.c, you define BUILDING_SHRLIB, which turns off the SYS_TRAP macro and yields standard C declarations of our API functions. The dispatch table requires the actual addresses of these functions to implement the trap-mapping mechanism.

You also declare a macro, ShrInstall, which expands to __Startup__. __Startup___ is the entry point to the library, which is invoked by the system when the library is loaded.

Finally, we declare the function ShrDispatchTable. This is the function that implements the dispatch table and returns its address. This function has an interesting declaration:

```
static Ptr asm ShrDispatchTable(void)
```

Ptr refers to a generic pointer (it is actually a typedef of char* to maintain compatibility with the Mac). The asm qualifier means that the function itself is implemented in assembler.

ShrInstall

ShrInstall initializes the dispatch table in the library's system library table entry. The ShrInstall function takes two parameters: a reference number and a pointer to the library's system library table entry. It assigns the return value of ShrDispatchTable to the entry's dispatch table attribute:

```
   // Install pointer to our dispatch table
   entryP->dispatchTblP = (Ptr*)ShrDispatchTable();
```

This function also clears the entry's globals attribute:

```
    // Initialize globals pointer to zero (we will set up
    // our library globals in the library "open" call).
    entryP->globalsP = 0;
```

Finally, you return 0 to indicate success. If you returned a negative error code, SysLibLoad would fail and return this error.

`ShrDispatchTable`

The `ShrDispatchTable` function itself simply loads the address of the dispatch table into the A0 register and returns. The A0 register holds function return values.

```
LEA    @Table, A0    // table ptr
RTS                 // exit with it
```

The actual dispatch table is coded into this function as well. First, declare a series of macros to simplify the hand-coding effort required.

We define the total number of traps published by the library. There are the four standard traps for `Open`, `Close`, `Sleep`, and `Wake`, and three custom traps for `GetAPIVersion`, `GetText`, and `SetText`:

```
#define numTraps (7)
```

We add one to the number of traps to determine the number of entries in the dispatch table. The extra entry is for the library name:

```
#define numEntries (numTraps) + 1
```

Starting with the number of entries, you can calculate the offset from the beginning of the table to the first jump. For each entry, there is a word in the dispatch table:

```
#define offsetToJumps ((numEntries) * 2)
```

The last piece of information we need is the size of the jump instructions. Because Palm OS uses short jumps, this is four bytes; two for the JMP instruction and two for the address in which to jump:

```
#define jumpSize (4)
```

From these macros, we can define a macro that calculates the offset to a specific jump by index:

```
   #define libDispatchEntry(index) \
   (offsetToJumps + ((index)*jumpSize))
```

We use this macro to define the address of the dispatch entry for each trap:

```
@Table:
      // Offset to library name
      DC.W   @Name

      // Start of standard traps
      DC.W   libDispatchEntry(0)   // Open
```

```
    DC.W   libDispatchEntry(1)   // Close
    DC.W   libDispatchEntry(2)   // Sleep
    DC.W   libDispatchEntry(3)   // Wake

    // Start of the Custom traps
    DC.W   libDispatchEntry(4)   // GetAPIVersion
    DC.W   libDispatchEntry(5)   // GetText
    DC.W   libDispatchEntry(6)   // SetText
```

Note that the index parameters passed to the macro `libDispatchEntry` are numbered consecutively, beginning with zero. This is required to make sure that they point to the correct entry in the jump table. Also note that the first entry is the address of the Name label. This label is at the end of the jump table.

The jump table is a set of jumps to the addresses of the functions published by the library:

```
// Standard library function handlers
@GotoOpen:
    JMP         ShrOpen
@GotoClose:
    JMP         ShrClose
@GotoSleep:
    JMP         ShrSleep
@GotoWake:
    JMP         ShrWake

// Custom library function handlers
@GotoAPIVersion:
    JMP         ShrAPIVersion
@GotoGetText:
    JMP         ShrGetText
@GotoSetText:
    JMP         ShrSetText
```

The final entry in the dispatch table is the library name:

```
    DC.B        shrName
```

This name identifies the library to the system, which looks for its address at the beginning of the dispatch table. Client applications can check if the library is loaded by passing the library name, defined by `shrName`, to `SysLibFind`. If the library is already loaded, the library's reference number will be returned.

It is important to note that the order of the trap values must match the order of the dispatch entries, which must match the order of the jumps in the jump table.

A Final Note on Dispatch Tables

If all this sounds horribly complex and scary, it shouldn't be. The dispatch table in this chapter can be copied and modified to suit your shared library. You don't need to know how or why it works, only that it does. There are a few things to remember, though:

- The order of the trap identifiers as declared in the public header *must* match the order of the respective functions in the jump table.
- The `numTraps` macro *must* be equal to the number of public functions declared.
- There *must* be one entry in the dispatch table for each public function.
- The standard functions *must* appear first in the jump table and in the correct order: `Open`, `Close`, `Sleep`, and `Wake`.

If your library is crashing, or the wrong functions are being invoked, these are the first things to check.

Summary

Shared libraries are not hard to create. Conceptually, they are simply function tables that follow specific rules. The bad news is that these rules are not necessarily intuitive. The good news is that when you've created your first library, you can use it for a template and stop worrying about these rules. The best news is that you've just created your first library.

Shared libraries are not particularly hard to implement, but they require care and patience. Used correctly, they will repay the investment with greater code reuse, increased application stability, and reduced resource requirements. You might think of them as peace of mind, encapsulated.

30 Other Advanced Project Types: Multi-Segment and Static Libraries

IN THIS CHAPTER

- When Is a Palm Application Too Large?
- Other Options for Alleviating Size Problems
- Using Static Libraries to Reduce Project Complexity

Even though the Palm documentation, the style guidelines, and various resources and discussion groups on the Internet repeat the mantra "keep it simple, keep it small," you might at some point find yourself in the unenviable position of developing a Palm application that has grown too large. Alternatively, you might enter the design phase for a new Palm program and wonder how much functionality can (and should) be incorporated into the application. When is a Palm program too large? Perhaps more importantly, what kinds of project-development techniques are required to effectively manage a program that is non-trivial?

A Palm program can be considered "too large" for a couple of reasons, and proving that your program is too large is often difficult. You might see hard evidence such as an inability to build your program in CodeWarrior. The symptoms might be more subtle, surfacing only sporadically as mysterious, hard-to-replicate system crashes.

Sometimes instead of sheer size, an application is large enough in scope that it can be more effectively managed at a project level if it is broken into more easily identifiable and understood chunks of functionality. In Chapter 29 on shared libraries, you looked at one method of accomplishing this, the Palm Shared Library. Like a Windows DLL, a Palm Shared Library can encapsulate a set of program services and offer them as a set of functions that can be integrated efficiently into multiple client applications.

However, as you learned in Chapter 29, shared libraries do require some effort on the part of the developer to program, and done right, they require a decent amount of up-front planning. Although shared libraries are wonderful things, for many situations Palm application complexity can be managed quite effectively with some less labor-intensive techniques.

This chapter sheds some light on the problems experienced by large or complex Palm programs, and offers advice on ways to remedy those problems. Specifically, the chapter

- Defines the limitations on size
- Presents guidelines for reducing stack and memory usage
- Explains how to create multiple-segment projects
- Explores options for partitioning your application
- Shows you how to create static libraries that can be easily linked into your application

When Is a Palm Application Too Large?

A Palm application can become "too large" in several ways:

- Excessive dynamic memory allocation.
- Exceeding available stack space.
- The size of the application exceeds 64KB.
- One part of the application is making a function call to another part of the application that is more than 32KB away in the code.
- There are too many source files in the project, making it tedious to browse your project, find your code, and understand the overall program structure.

Murphy's Law dictates that in your project you will most likely fall into one or more of these categories on the day before your project is due. Hopefully that's not the case for you, but on the other hand I can't think of a really "good" time for a developer to be in any one of these situations. Nevertheless, I assume that because you are reading this chapter, it is possible that you fall into one of these categories. All of them are solvable; let's review them one at a time.

Excessive Dynamic Memory Allocation

If you read Chapter 16 on Palm memory architecture, you might recall that I scared you out of your wits by proclaiming that on some earlier Palm devices that run

Palm OS 3.0, such as the Palm III, your application will have approximately 36KB of dynamic RAM at its disposal during program execution. Compared with the vast amounts of memory available to applications on desktop operating systems, this seemed positively puny and maybe just a wee bit unfair. Although later versions of Palm OS dramatically increase the available memory in the heap, there are still limitations. Furthermore, your program might still need to support older devices, so you must find a way to make your program live within these limits and exercise restraint in how much dynamic memory is required at any one time.

If built-in applications such as the Address Book can manage thousands of records while using tiny amounts of memory, so can your program. Several strategies can alleviate the problems associated with limited heap space:

- Allocate handles rather than pointers for your memory. Palm OS can move handle-based memory around to avoid heap fragmentation, but pointers are locked down in memory and reduce the ability for the heap manager to locate free space.
- Do not load a lot of data into memory from database storage. Instead, read and write database records in place.
- Keep out a sharp eye for potential memory leaks.
- Do not hang on to memory chunks longer than necessary. Return them to the free store as soon as possible.
- Religiously check memory allocation calls for the possibility of failure and design a strategy for handling out-of-memory situations.

Exceeding Available Stack Space

Unless you fiddle with the Palm OS header definitions, your Palm program will be limited to approximately 2KB of stack space. The stack cannot grow larger than this during your program's execution. If your program pushes more than 2KB on the stack at any one time, you have "blown your stack," with consequences ranging anywhere from system crashes to data loss to sporadic bizarre program behavior.

In developing any reasonably complex application, it is all too easy to quickly use up stack space with automatic variables such as large character arrays for receiving entered form data or displaying messages to the users. Declarations such as `char szMessage[200];` might look innocent enough in a single function, but consider that the function in question might be called by another function having similar automatic variable declarations, and so on. It doesn't take much of this to quickly approach the stack limit. As is the case with the other types of problems described in this chapter, it is no fun to scour your program code the day before a product release, looking for ways to reduce your stack usage.

Some strategies for avoiding stack problems are as follows:

- Allocate small, temporary dynamic memory chunks rather than placing character arrays on the stack.
- If necessary, use global variables rather than large local variables.
- Do not use recursion!
- Keep an eye on careless overuse of the larger data types such as floating-point numbers.
- Use the stack-checking diagnostics that are supported in POSE, the Palm OS Emulator.

Applications That Are Larger Than 64KB

Chapter 16 on Memory Management, mentioned that memory chunks are limited to 64KB in size. Applications that consist of a single code resource (also known as a *code segment*) also live under this restriction because they are stored as a single chunk on the device.

CAUTION

When an application under development starts exhibiting some random, odd behavior, check whether it has exceeded the 64KB limit.

What is the way around the 64KB limit for application size? In CodeWarrior, the answer is to add support for multiple segments in your project. CodeWarrior allows you to create a multiple-segment project in the IDE, in which each segment by itself can be up to 64KB in size. To create a multiple-segment application, perform the following steps in CodeWarrior:

1. At the top of your project window, click the Segments tab.
2. Under the Project menu, choose Create New Segment.
3. Give your new segment a name.
4. Repeat step 3 as many times as you want to logically break up your application into segments smaller than 64KB.
5. Add new files to your segments or move files among segments by dragging them with the mouse.

The only rule is that your first segment should contain the runtime Pilot library (typically MSL RuntimePilot (2i).lib) as well as the source file that contains your

`PilotMain` and any other code that `PilotMain` directly calls when it receives a normal launch code. (This is another reason why I broke out `PilotMain` and event loop handling into a separate source file back in Chapter 2, "Anatomy of a Palm Application."

Making Calls to Functions More Than 32KB Away

As your application nears 64KB in size, and even more so after you create multiple segments to build a program greater than 64KB, you might begin experiencing the linker error `"16-bit reference out of range"`—if you are lucky. If you are not so lucky, the linker will try to link your application anyway, and you will get a program that sporadically goes bonkers.

The reason for this error is that Palm OS uses 16-bit signed values to represent function call jumps, imposing a hard limit of 32KB as the maximum distance between a piece of code and the destination function it is trying to call. Although you might be able to squeak by if you are lucky and move your source files around in the project to create a link order in which calls are not greater than 32KB, ultimately this is time-consuming and exceedingly error-prone.

CodeWarrior again comes to the rescue with the "Code Model" project setting. To edit this setting, go to your project's settings in CodeWarrior, and click the 68K Processor category in the left pane. The top-most setting is Code Model, and you have a choice between small, smart, and large:

- The small code model is the default and results in the 32KB limit just described because all calls are 16-bit. Because of the small jump size, code compiled in this model is most efficient in terms of function-call processing.
- The large code model forces the compiler to simulate "far jumps" by creating code that can jump by a 64-bit value. Although this neatly solves the far jump problem, there is a penalty in performance by making the processor do extra work with each function call. Depending on how desperate you are to solve the 32KB problem, you might not care about this performance loss.
- The smart code model attempts to create the best of both the small and large models: Calls made within the same source file are assumed to be less than 32KB and thus use 16-bit jumps. Calls made across source files are assumed to be greater than 32KB and thus use 64-bit calls.

How you organize your source files in the Project window, especially in a multiple-segment project, can drastically affect how much the 32KB limit comes into play. If you can group logically related files together so most calls are made within the same segment, you will probably not experience as many problems as you would with a project with random distribution of source files across the project tree. I happen to like organizing my projects into subcomponents, so this arrangement works well for me.

Other Options for Alleviating Size Problems

In some cases, size problems highlight architectural issues that are best addressed at design-time. If you can plan for the logical partitioning of your application into components that have a clear, well-defined purpose, you will give yourself the best chance of avoiding (or at least preparing for) the possibility of large application problems.

Some options to consider as you are partitioning your application follow:

- Consider how much of the application truly needs to be on the Palm.

  As I pointed out earlier, complex user interfaces requiring heavy data entry are not well suited for the Palm—nor are processor-intensive algorithms and number-crunching routines. If your application has any of these characteristics, you might want to consider moving some of the functionality off the Palm and onto either the desktop or a remote server application (assuming Internet connectivity is an option). It is unrealistic to assume a large application that formerly ran on fast workstations with 128MB of RAM will be suitable for a direct port to the Palm.

- Break off one or more components into shared libraries.

  Shared libraries, which are covered in Chapter 29, “Shared Libraries: Extending the Palm OS,” are somewhat akin to dynamic link libraries (DLLs) under Windows. Unfortunately, shared libraries are tagged with a bit of black-magic mystique in that not too many developers understand how to create them (or so the number of available shared library components would have you believe). Part of this is no doubt due to the relatively sparse documentation on the subject.

  This is a shame because they offer a way to package a set of program services in a .PRC and make them available to one application or even many applications at the same time. Some examples of shared libraries are the Palm OS built-in Serial Manager and the IR Library.

Using Static Libraries to Reduce Project Complexity

In addition to applications and shared libraries, another project type available to Palm developers is the static library. Static libraries are not unique to the Palm OS platform; they have been in use by C and C++ developers for years on Windows and other computing platforms.

A static library is really just a collection of C or C++ source files that have been grouped together in a project. These files are collectively compiled into a self-contained binary code unit, which as a .LIB file that is the target of the static library project. This .LIB file can then be included in other projects, making all of the functions and services in the included C or C++ source files available to the parent application. As a matter of fact, all Palm application projects include at least one static library, which is typically the “MSL Runtime Palm OS (2i).Lib” file that must be in your project’s first segment.

The best way to think about a static library is that it is a package, or a container, of functions available to your program. As opposed to shared libraries—on which you need to explicitly load and unload the library, and you have to carry around a reference number to make function calls—a static library is very simple to use. Your program simply calls the functions in the library the same way as it would if the source C or C++ files were directly included in your project.

Pros and Cons of Static Libraries

A great use for static libraries is if you are writing a series of programs that all need to make use of some common utility functions. At my company we’ve developed over the years a sizable toolbox of utility functions that cover areas of memory management, user interface, math functions, and file access. Rather than include the same set of five, ten, or more source C files into every one of our projects, we’ve created a simple static library that’s included in every project we do. This is very clean and very helpful; it gives each program a nice head start in terms of a rich library of added value functions. The nice thing about it is that only the functions I call are linked into my application’s .PRC file, so including our library doesn’t necessarily bloat otherwise simple applications.

There are only two downsides to using static libraries. One of them is that you need to create a separate project for each static library and build it before the .LIB file can be included in your application. This issue can be mitigated somewhat by using CodeWarrior’s multiple-target project type, which allows you to maintain both the library target and the application target in the same project file.

The second downside is one of visibility of the source files. When you include a .LIB file in an application project, you are including a binary file, not the original source code. Thus, if you need to look at the original source code to the functions packaged inside the library, you need to open the separate library project (unless you are using a standalone editor, in which case you can easily access any file whether its inside of your application project or not).

For these reasons, my recommendation is that you lean toward using libraries to package code that is relatively mature and well documented. Ideally if the functions in the library are robust, bug-free, and described well in the header file for the library, you will not need to debug into or read the original source code on a regular basis, and thus the library can be used as a "black box" of sorts.

Static Library Projects in CodeWarrior

Creating a static library project in CodeWarrior is easy. You can either create one from scratch, or you can simply take one of your application projects, make a copy, and then make the following changes in the copy:

- In the 68k Target section, set the project type to be Library, and choose Palm OS Library as the sub-type.
- Change the name of your target to reflect the name you want to have for your .LIB file.
- Do not include the MSL runtime library in your library.
- Do not include resource files in your library (resources cannot be part of a static library).
- Add the source files that you want to be part of the static library.

As you can see, this is a very simple set of steps, and much less work than creating a shared library project.

Summary

Palm's rise in popularity as a development platform, coupled with the ever-increasing storage, display quality, and processing power, presents the opportunity to move more and more functionality onto the Palm platform. The natural progression in such matters will virtually guarantee a corresponding rise in the number of large applications created for the Palm.

Large applications are not in and of themselves a bad thing, if properly planned. Developers can use the information in this chapter to keep an eye out for the signs that their programs have exceeded one or more of the size limitations and apply the appropriate remedies.

31 Saving Program State: Application Preferences

IN THIS CHAPTER

The Palm OS does not support the execution of multiple applications simultaneously. It might come as a bit of a surprise to developers new to the Palm platform that their applications shut down when a user switches to another application. Simply tapping the Applications icon is enough to close an application!

In fact, if you do nothing else, every time the users tap on your application to launch it, they will be greeted with your main screen, regardless of how their last session with your program ended. It is as if the users were starting over every time; in fact, from the perspective of your application, they are. Depending on your application, this behavior can be at the very least disconcerting, if not downright annoying, to the users. Imagine if the built-in Datebook application always opened to the same view and date, requiring you to navigate to the current date every time!

This chapter explains the problems associated with loss of program state and introduces the application preferences facility as a way to deal effectively with those problems.

Dealing with Palm OS Application Switching

Losing state causes some obvious (and not so obvious) problems with many applications. For example, it might not be acceptable to lose the current application state. Consider a situation in which a user is composing a long email, and they suddenly need to look at their calendar. Unless the developer of the email application carefully planned how to save the work that was started…poof! The email is lost.

Another example is the problem of how to maintain user preferences and configuration settings. The users could get quite annoyed if they preferred "sort by company" in the Address Book, yet the Address Book application always reverted to "sort by name" when it started.

On Windows or other multitasking platforms, this sort of inconvenience is not a problem because applications retain state as they are activated and deactivated on the desktop. Certain system facilities, such as profile-initialization files and the Registry, easily store configuration settings and user preferences in a standard manner.

On the Palm, with no "file system" per se, what tools are available to the developer to address this problem? The answer lies in the Palm SDK's system (or application) preference functions.

What Are Application Preferences?

Application preferences allow you to save the state of your application from one execution to the next. They are maintained in a special Palm database called "Preferences," (you can see this database using `DBBrowse` from Chapter 17, "Understanding Palm OS Databases") which is dedicated to managing the preferences for all applications, as well as the user's system preferences such as date and time format.

You don't have to do anything to create your application's special entry in the Preferences database: It is automatically maintained by the Palm OS.

Using Application Preferences

Application preferences are best employed for storing your program's state and configuration settings. For example, you have good candidates for preferences if you want your program to initialize to one of three possible initial main views or if you have sort or other data-view–oriented settings.

Using preferences for options such as partially completed records or most recently accessed data is less desirable. The problem lies in the link to data elsewhere on the Palm that might either change or be deleted by another process.

Using the Palm SDK Functions to Handle Application Preferences

Application preferences essentially have two functions: `PrefGetAppPreferences` and `PrefSetAppPreferences`.

`PrefSetAppPreferences` stores an application-defined block of memory into the application's preferences database entry. In this sense, this process is much like using `DmWrite` to save a block of memory to a record, with the application being responsible for interpreting the record structure properly. Here's an example:

```
// Define the structure of the application preferences
// block

typedef struct
{
    UInt16 ViewID;
    UInt16 CurrentSort;
}AppPreferences;
...
static AppPreferences s_Prefs;
static UInt16 s_wVersion = 2;
// In your NormalLaunch handler in your main event loop
...
UInt16 wSize = sizeof (AppPreferences);
// Load the app's preferences
PrefGetAppPreferences ( 'TEST',
            TEST_PREFS_MAIN,
s_wVersion,
            (VoidPtr)&s_Prefs,
            &wSize,
            TRUE);
...
// Do your main event loop
...
// App is closing, store preferences
PrefSetAppPreferences ( 'TEST',
TEST_PREFS_MAIN,
s_wVersion,
            (VoidPtr)&s_Prefs,
            sizeof (AppPreferences),
            TRUE);
```

For this fictional example, I have defined a structure that contains two application preferences: a view ID (corresponding to one of several possible form views) and a current sort, which stores the user's preferred sort order. The general logic flow is all driven from your `PilotMain` code in your `sysAppLaunchCmdNormalLaunch` handler.

When your application is first launched, you should read your application's preferences from the system into local storage where they can be read from the various parts of your program. You do this by calling `PrefGetAppPreferences`. To correctly identify the proper preferences, you provide the creator ID, the ID of the preference set (you are allowed to create multiple sets of preferences for your application, so you need to identify which one you are interested in), and your application's version number. The version number is insurance against changes in the block size or layout for the application's preferences over time. If you increment the version number for an associated change in the preferences structure, Palm will create a new preferences block tied to that unique version number.

Note that `PrefGetAppPreferences` has a `prefsSize` parameter that is both an `in` and an `out` parameter. You need to tell `PrefGetAppPreferences` the size of the preferences block you are passing in, but the function also passes back the actual size of the block to you if it succeeds in locating a block. You should check the return value, and even if the function call succeeds, make sure that the returned block size is what you expect.

Once you've loaded your application's preferences, you should copy them to some place in your program's memory where they will be accessible by other program modules in your application that need to retrieve the preferences.

After preferences are loaded, you proceed to enter your application's main event loop as normal. When the event loop exits, indicating application termination, this is the best place to save the current settings of the application to the preferences database. This process simply involves the inverse of the `PrefGetAppPreferences` call.

System Preferences

In addition to application preferences, predefined system preferences control some of the operation and appearance of the OS itself. These are defined in the SDK header file preferences.h in the enumerated type `SystemPreferencesChoice`. Some of the supported preferences are fairly useful, depending on your application's needs: date and time format, daylight savings, starting day of the week, and so on. Others are either undocumented or somewhat obscure (`RonamaticChar` and `AllowEasterEggs`).

System preferences are set individually using `PrefSetPreferences`; you pass it the `SystemPreferencesChoice` you want to modify. There is a function to get all the preferences at once in a `SystemPreferences` structure, but it's hard to imagine why you would need so many of them. Note that all preferences are stored as word-size values.

Conversely, you can retrieve an individual preference by calling `PrefGetPreferences`, which returns the associated word value.

Summary

Palm makes it fairly easy for application developers to maintain state across program executions by encapsulating a special common storage location behind well-defined SDK functions. You should carefully evaluate your application and determine how you can use application preferences to simplify your users' experiences.

32 Palm OS 5 and ARM: Looking into the Future

IN THIS CHAPTER

Things move so quickly in the computer world, and sometimes it seems that no sooner do you master the tools and techniques for the current platforms, things change right out from under you.

The handheld computing industry is no different in this respect. In fact, from what I've seen, things are moving and changing even faster for PDAs than for other areas of computing.

Just look at what has happened since the first edition of this book was published just three short years ago. The number of new companies producing devices for the Palm OS platform has grown dramatically. The standard memory configuration of Palm devices has grown from 2MB to 16MB. Metrowerks has released no less than four new major versions of CodeWarrior. Palm has released three new major versions of the Palm OS (3.5, 4.0, and 5.0).

There are also thousands upon thousands of Palm OS-based software applications out there covering every conceivable software category. Despite the "Zen of Palm" guidelines, some of these applications, including games, office applications, and communications utilities, are literally pushing the limits of what can be done on a relatively low-powered device with limited memory storage.

Interestingly, the one feature of Palm OS handhelds that has not changed much is the processor driving all of this activity. Today the typical Palm OS-based handheld device still ships with a 68KB Dragonball processor, running at a lowly 33MHz.

In the year 2001, Palm's operating systems group began spreading the word that the next major version of Palm OS would be built around the more powerful ARM set of processors. ARM processors offer scalable performance for mobile computers up to 200MHz and significantly beyond. Contrast this with the current Motorola 68KB processor that is found on all current Palm OS devices, which runs at speeds of 20MHz, 33MHz, and 66MHz. The new operating system combined with a new breed of Palm OS devices using the ARM processor will undoubtedly drive performance and application capabilities to a whole new level.

Palm OS 5 shipped to developers and device manufacturers in the summer of 2002. As of this writing, the very first ARM-based Palm OS devices have yet to ship, but their release is imminent.

This chapter presents information on Palm OS 5, and covers

- Palm OS 5 user features
- Palm OS 5 developer features
- New tools and APIs in Palm OS 5
- Impact on code written for Palm OS 4 and earlier

What is Palm OS 5?

Palm OS 5 is, to put it simply, Palm OS rewritten from the ground up to be compatible with ARM processors. Compared to the 68KB processor used today, ARM processors have a different instruction set, different ways of representing and storing data, and other differences at the "kernel" level that are important to the designers of operating systems.

ARM processors provide greater speed, a wider range of cost/value options for handheld makers, and over time will offer many additional capabilities for PDAs. Palm OS 5 and beyond will allow PDA users and software developers to leverage those advantages.

Palm's operating system group valued greatly the huge existing base of Palm OS-compatible software applications. Thus, despite the internal differences between Palm OS 4 and Palm OS 5, Palm worked hard to provide a high level of compatibility for programs that use the existing Palm APIs. This chapter explores this compatibility in more depth later.

It is important to understand that Palm OS 5 is a "compatible" operating system advance over Palm OS 4. Although there are new programming APIs in the OS 5.0 SDK, by and large it is still the same Palm OS. New versions of the Palm OS beyond

OS 5 will build on the new ARM kernel, and begin exposing more capabilities to both users and developers.

Palm OS 5 is not a multitasking/multithreaded OS, at least from the user/developer perspective. Applications still run one at a time; there is no application swapping that allows users to run multiple applications simultaneously. For developers, multithreading still remains in the future; all code writing at the application level is single-threaded.

What’s in Store for the Users?

For users who have used older devices, Palm OS 5 will appear very familiar. The same look and feel is present in both the built-in and third-party applications.

What users of the new devices will mostly see is an improvement in speed and performance. Most applications will run faster. Exactly how much faster depends on how the application is written, what kinds of operating system calls are made, and other factors. It is even possible that a few applications might run slower because of the kind of code they run, but in general the user experience will improve due to the faster processor speeds.

An indirect, longer-term benefit to users is that new types of applications can be created that would never have performed acceptably on older devices. Heavy graphics-intensive applications such as high-resolution games and animations will be possible. Storage-bound applications that handle large amounts of data will be feasible as well.

What’s in Store for the Developers?

The writing is on the wall: Palm OS 5 and ARM are the future. Even if there is no good reason to make your application take advantage of Palm OS 5, the simple fact is that sooner or later your application will need to run on Palm OS 5 devices. Although it will take some time for older devices to be phased out, slowly but surely the percentage of Palm OS handhelds running OS 5 will grow, to the point where you can’t ignore it.

But there are good reasons to more actively develop for Palm OS 5. Achieving faster application performance will require less work on your part, and permits you to write much more straightforward code.

Palm OS 5 combined with the ARM capabilities will open up the door to new types of applications, and permit features in existing applications that would not have previously been possible. Of course it is up to you to pursue this opportunity, but it is there for the taking.

Although relatively few new OS APIs were exposed in this release of the developers SDK, there are some that may be worth looking into, depending on the type of application you are developing. New APIs for security now exist that allow you to include robust encryption capabilities in your program. New high-density bitmap functions are also now available. Finally there is the option to create something called an *ARMLet*, which is a piece of code that is built specifically as a native ARM module, allowing Palm OS 5 developers to selectively optimize pieces of code for the new devices.

PACE and Application Compatibility

As stated previously, there is a high level of compatibility on Palm OS 5 for programs that were written for previous versions of the operating system. Achieving this was a challenge for Palm, because existing applications are compiled and linked assuming a 68KB processor. Palm needed to provide a way to run these 68KB-based applications on an ARM processor.

PACE and Emulation

Palm came up with a *compatibility layer* called PACE. PACE is short for Palm Application Compatibility Environment. PACE offers a layer of code that allows a 68KB application to run unchanged on Palm OS 5.

PACE does this by running your application in an emulation mode, allowing your program to think that it is running on a 68KB device. PACE takes all the function calls made by your application, and modifies them to be ARM-compliant before passing them onto the native Palm OS 5. In particular, 68KB processors are said to be *big endian*, whereas ARM uses *little endian* (endian refers to the order of data in bytes). PACE automatically makes the byte swaps for you.

Although this will impose some performance penalty, it is for the most part expected to be slight. And of course the benefit is that, in all likelihood, your program will be able to run on Palm OS 5 with only slight modifications.

Achieving Palm OS 5 Compatibility

Of course you knew there had to be some work involved; it would be too good to be true if you were able to simply expect Palm OS 5 compatibility with no effort!

Thankfully Palm has successfully minimized what needs to be done. In fact, Palm has been instructing the developer community for quite some time to do the things that will ensure future compatibility. As a result, most of the areas that require work on the part of the developer involve cleaning up code to make it “well-behaved” and play by the rules.

The main issue that is a no-no is directly accessing data that belongs to the operating system. For most developers, this means code that accesses form and user interface structures such as tables must be changed. For this purpose, Palm has recommended using *Glue* functions instead of accessing the structures directly. Glue functions in general provide a safe way for developers to deal with internal Palm OS structures by calling a function rather than directly setting the structure members. (Glue functions are described in the developer documentation, and are listing in PalmOSGlue.h.)

If your application does perform direct access and it runs on Palm OS 5, the program will almost certainly crash. This is because the memory address it is referencing is not available at the application level. Thus, you need to make those changes in order to run successfully on Palm OS 5.

The Palm Simulator

With Palm OS 5 come some new tools for developers. Metrowerks CodeWarrior 7 and above will still be fine for building your 68KB application. But for testing your application, there is a new tool called the Palm Simulator, or PalmSim.

PalmSim is not an emulator like POSE. Rather, it is Palm OS itself, recompiled as a native Windows application. Palm OS 5 is now a series of modules, and PalmSim is composed of a series of Windows DLL modules. These DLLs contain the system code, Palm applications, PACE, and the lower-level device support.

PalmSim, like POSE before it, needs a ROM to actually run Palm OS. Note that older POSE-compatible ROMs are not compatible with PalmSim. You will need new ROMs with PalmSim.

Running and debugging your application in PalmSim is simple, and the usual POSE ways of loading an application are supported, including drag and drop. In addition, a wealth of new command-line options helps you automate loading a specifically configured session. Debugging your application from CodeWarrior is also familiar and works automatically. You'll be pleased to note that debugging is much faster with PalmSim!

In terms of providing a desktop-based way of running your applications on Palm OS, PalmSim provides most of the features you've learned to use on POSE. Gremlins, logging, and other features are preserved, and even in some cases enhanced.

Figure 32.1 shows the current version of PalmSim.

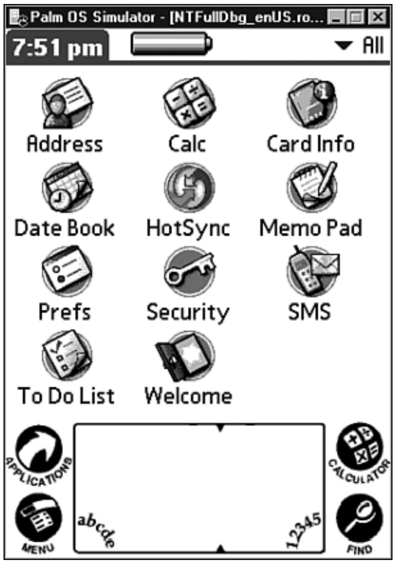


FIGURE 32.1 The Palm Simulator running OS 5.

ARMlets

ARMlets are ARM-native code resources that can be called from within 68k applications that are running on Palm OS 5's PACE subsystem.

ARMlets provide Palm OS programmers with a way to optimize certain performance-bound components of applications by running them as native ARM code, thus bypassing the PACE layer. As you'll see, due to the complexities and restrictions on ARMlet development, you would not want to count on pushing large portions of your application into an ARMlet, nor is it an appropriate vehicle for housing form-driven user interface code.

Rather, you should consider an ARMlet when you have found a function or module in your application that suffers poor performance under PACE, despite reasonable efforts to optimize the code through normal programming practices.

Restrictions on ARMlets

There are several important restrictions on how you develop ARMlet code, as follows:

- Each ARMlet code resource must be <64k in size
- Debuggers currently offer no support for debugging ARMlet code

- ARMlets cannot directly call the full set of Palm OS functions. Metrowerks has stated its intention to provide a thin library of APIs that provide stubs for the most common Palm OS functions. Beyond those functions an ARMlet will need to call back into the PACE layer in order to access a Palm API.

In addition, run-time library support for minimal C/C++ language services will be provided as a library in the new Metrowerks CodeWarrior toolset (see below).

Adding ARMlets To Your CodeWarrior Project

Just as this book is headed into publication, Metrowerks has provided us with a sneak peek of its next-generation version of CodeWarrior, which will be called CodeWarrior 9.

CodeWarrior 9 builds on the standard Palm application development tool in a number of ways, but perhaps most notably in that for the first time a Palm application developer will be able to use CodeWarrior to incorporate ARMlets into a CodeWarrior project.

The new CodeWarrior Palm OS linker directly supports adding code resources (in the form of a ".bin" file) to a project, if they are properly named. The naming convention for these .bin files is RRRRHHHHs.bin, where RRRR is the resource type, and HHHH is the resource ID in hex digits. After the first eight letters, any suffix may be applied to allow alternate versions of the same code resource in a project.

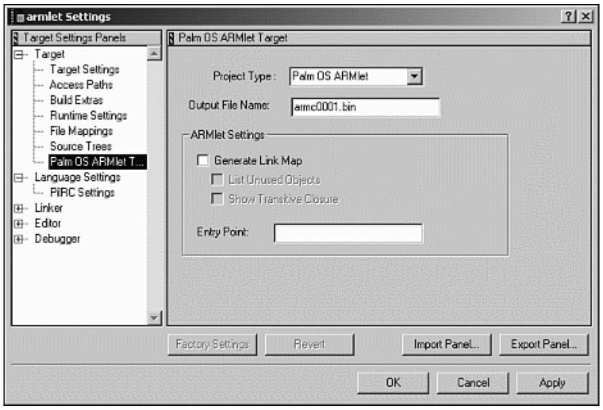
For example, an ARMlet with type 'armc' and ID 1000 would be added to the project as "armc03E8.bin." Files with a .bin extension that are added to a CodeWarrior target that do not follow this naming convention will not be used by the linker, and in fact may cause a linker warning.

Figure 34.2 below shows an early screenshot of the project preferences panel for an ARMlet project type in CodeWarrior 9:

Loading and Using an ARMlet in your program

ARMlets are developed as Palm OS code resources, which is just another project type in CodeWarrior. As with shared libraries, functions in code resources are not directly called by an application, rather the resource must be loaded into the application, and locked in memory. Once that is done, a new Palm OS api PceNativeCall may be used to call a function in the ARMlet.

Although the timing of publication prevents me from providing a working ARMlet project using CodeWarrior 9 in this edition of the book, Metrowerks has stated its intention to provide example ARMlet projects to developers upon release of CodeWarrior 9. As always, check with `http://www.metrowerks.com` and `http://www.palmsource.com` for the latest information on tools and sample code.



Project Type: Palm OS ARMlet
Output File Name: armc0001.bin
☐ Generate Link Map
☐ List Unused Objects
☐ Show Transitive Closure
Entry Point:

FIGURE 34.2 Project settings for an ARMlet target in CodeWarrior 9.

A Developer's "To Do" List for Palm OS 5

Until new Palm OS extensions emerge that enable native ARM applications to be developed, the most important thing you as a Palm developer can do is to make sure that your existing 68KB applications run successfully on Palm OS 5.

If you do not have a device to test with, you can still achieve compatibility by using PalmSim. If you have not already done so, I recommend that you download PalmSim from the Palm OS Web site (its free) and begin testing and debugging your applications. With few exceptions, any changes you make to enable Palm OS 5 compatibility will not adversely affect how your applications run on earlier versions of Palm OS. Be careful about making assumptions regarding the availability of APIs in older versions of Palm OS. You'll need to make sure you are doing robust version checking if you are taking advantage of newer functions.

The other thing you should be doing is contemplating how your existing application will take advantage of the greater ARM performance. Obviously, as you begin to add features that assume faster processors, those features will not run acceptably on lower-end devices. Because 68KB devices will still need to be addressed by commercial software developers, this presents a significant design issue.

Depending on your application, you might want to begin producing two separate versions of your application, one for each target platform. Although this sounds complex, if you manage and plan it well, it should not be too difficult. Doing so will allow you to focus on the best user experience for each target.

Summary

Palm OS 5 opens up a new world of speedy handheld devices, richer application functions, and scalability for both users and developers. There are important compatibility issues that you need to understand and address as you move your development forward to embrace both the old and the new. This chapter introduced you to the most immediate issues facing developers today.

As mentioned in the beginning of the chapter, things are indeed moving fast in the mobile computing world. Although this introduces complexity for software developers who seek to support an increasing array of screen types, devices, operating system versions, and processors, the other way to look at it is as a wealth of opportunity. This change indicates further signs that the future does indeed look bright for Palm programs. Bring it on!

Index