

Windows[®] CE Developer's Handbook[™]

Terence A. Goggin

Unparalleled Instruction
and Advice from the
Acclaimed "Dr. CE"

Put Your Windows
Development Skills to
Work on Windows CE



Featured on the CD:

- Sample Code
- Free ActiveX Control for CE
- Evaluation Software
- Links to Important CE Web Sites
- Electronic Reference to the CE API and the C Runtime Library Functions





Windows CE Developer's Handbook

TX 5-008-224



T:0005008224

Windows[®] CE Developer's Handbook[™]



Terence A. Goggin
with
David L. Heskett and
Jason M. MacLean



San Francisco • Paris • Düsseldorf • Soest • London

Associate Publisher: Gary Masters
Contracts and Licensing Manager: Kristine O'Callaghan
Acquisitions & Developmental Editor: Brenda Frink
Editor: Sally Engelfried
Project Editor: Bronwyn Shone Erickson
Technical Editor: John Psuik
Book Designer: Kris Warrenburg
Graphic Illustrator: Tony Jonick
Electronic Publishing Specialist: Robin Kibby
Proofreader: Judy Weiss
Project Team Leader: Shannon Murphy
Indexer: Meg Fortune McDonnell
Companion CD: Ginger Warner
Cover Designer: Design Site
Cover Photographer: The Image Bank

DA76
56
10636636
1999
CIP 2
MRC

Developer's Handbook is a trademark of SYBEX Inc.

The HP Jornada is a trademark or registered trademark of Hewlett-Packard Company, copyright ©1999.

The Casio E-11 is a trademark or registered trademark of Casio, Incorporated, copyright ©1999.

The Casio PA-2400 and Casio PA-2500 are trademarks or registered trademarks of Casio Business Solutions Group, copyright ©1999.

TRADEMARKS: SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Copyright ©1999 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 99-61299
ISBN: 0-7821-2414-3

Manufactured in the United States of America
10 9 8 7 6 5 4 3 2 1

Software License Agreement: Terms and Conditions

The media and/or any online materials accompanying this book that are available now or in the future contain programs and/or text files (the "Software") to be used in connection with the book. SYBEX hereby grants to you a license to use the Software, subject to the terms that follow. Your purchase, acceptance, or use of the Software will constitute your acceptance of such terms.

The Software compilation is the property of SYBEX unless otherwise indicated and is protected by copyright to SYBEX or other copyright owner(s) as indicated in the media files (the "Owner(s)"). You are hereby granted a single-user license to use the Software for your personal, noncommercial use only. You may not reproduce, sell, distribute, publish, circulate, or commercially exploit the Software, or any portion thereof, without the written consent of SYBEX and the specific copyright owner(s) of any component software included on this media.

In the event that the Software or components include specific license requirements or end-user agreements, statements of condition, disclaimers, limitations or warranties ("End-User License"), those End-User Licenses supersede the terms and conditions herein as to that particular Software component. Your purchase, acceptance, or use of the Software will constitute your acceptance of such End-User Licenses.

By purchase, use or acceptance of the Software you further agree to comply with all export laws and regulations of the United States as such laws and regulations may exist from time to time.

Software Support

Components of the supplemental Software and any offers associated with them may be supported by the specific Owner(s) of that material but they are not supported by SYBEX. Information regarding any available support may be obtained from the Owner(s) using the information provided in the appropriate read.me files or listed elsewhere on the media.

Should the manufacturer(s) or other Owner(s) cease to offer support or decline to honor any offer, SYBEX bears no responsibility. This notice concerning support for the Software is provided for your information only. SYBEX is not the agent or principal of the Owner(s), and SYBEX is in no way responsible for providing any support for the Software, nor is it liable or responsible for any support provided, or not provided, by the Owner(s).

Warranty

SYBEX warrants the enclosed media to be free of physical defects for a period of ninety (90) days after purchase. The Software is not available from SYBEX in any other form or media than that enclosed herein or posted to www.sybex.com. If you discover a defect in the media during this warranty period, you may obtain a replacement

of identical format at no charge by sending the defective media, postage prepaid, with proof of purchase to:

SYBEX Inc.
Customer Service Department
1151 Marina Village Parkway
Alameda, CA 94501
(510) 523-8233
Fax: (510) 523-2373
e-mail: info@sybex.com
WEB: [HTTP://WWW.SYBEX.COM](http://WWW.SYBEX.COM)

After the 90-day period, you can obtain replacement media of identical format by sending us the defective disk, proof of purchase, and a check or money order for \$10, payable to SYBEX.

Disclaimer

SYBEX makes no warranty or representation, either expressed or implied, with respect to the Software or its contents, quality, performance, merchantability, or fitness for a particular purpose. In no event will SYBEX, its distributors, or dealers be liable to you or any other party for direct, indirect, special, incidental, consequential, or other damages arising out of the use of or inability to use the Software or its contents even if advised of the possibility of such damage. In the event that the Software includes an online update feature, SYBEX further disclaims any obligation to provide this feature for any specific duration other than the initial posting.

The exclusion of implied warranties is not permitted by some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights; there may be other rights that you may have that vary from state to state. The pricing of the book with the Software by SYBEX reflects the allocation of risk and limitations on liability contained in this agreement of Terms and Conditions.

Shareware Distribution

This Software may contain various programs that are distributed as shareware. Copyright laws apply to both shareware and ordinary commercial software, and the copyright Owner(s) retains all rights. If you try a shareware program and continue using it, you are expected to register it. Individual programs differ on details of trial periods, registration, and payment. Please observe the requirements stated in appropriate files.

Copy Protection

The Software in whole or in part may or may not be copy-protected or encrypted. However, in all cases, reselling or redistributing these files without authorization is expressly forbidden except as specifically provided for by the Owner(s) therein.

*I dedicate this book to Tim, my brother, my editor,
and my best friend. I could not have written this
book without you. Your hard work means more to
me than I could ever put into words.
Thank you.*

ACKNOWLEDGMENTS

There are a great many people who made this book possible, and I would like to thank them all.

First, thanks to Ann Goldmann, Brooke Richardson, and Suzanne Lamberton of Waggoner-Eddstrom for helping me on a project that was a little outside of the norm. You really helped me to get started and I want you to know that I appreciate it very much.

To Barry Raymond, Eric Drew, and Norman Hills of Casio's Vertical Markets Division: Thank you for sharing with me some of the unique and exciting projects you've created with Windows CE. Your real-world perspective was extremely helpful and got me started off on the right track.

To Bob Smith, Pat Carrasco, and Cheryl Balbach of Casio New Jersey. It was a pleasure working with you, and I hope you're pleased with the result.

To Scott Crossen, thank you for creating the excellent RAPI source code for Delphi, and thank you for graciously allowing me to reprint it here in this book.

To Helen Chan and Chris Yien of Hewlett-Packard, thank you for your help in completing this book and making sure it contained the very latest information about the newest CE devices.

To everyone at Microsoft who made time to help out: to David Streams, thank you for the initial interview and your insight and advice on what to cover; to Amy Stuhlberg, thank you for stepping in and helping me get in touch with the right people; to Scott Horn and Cyra Richardson, thank you for your advice on what developers want to learn about; to Doug Yip, thank you for putting me in touch with the right people; to Brian Sherrell, thank you for answering all of my questions and helping me sort through the issues; to Scott Henson, thank you for working with me on the promotion of the book before it was even finished; and to Prashant Sridharan, thank you for making sure I had the latest versions of the toolkit. I am indebted to all of you.

To Shirley Macbeth and Liam Cavanagh of Sybase; Neil Shepherd, Jay Botelho, and Etienne Viellard of Oracle; and Larry Lundgren and John Deurbrouck (who wrote the Raima section of Chapter 12) of Raima Corporation, thank you all for providing samples of your products and write-ups of their capabilities.

To Andy Mallinger and Michael Snyders of InstallShield; to Yuko Tanaka, Mike Nydam, Mike Krautkramer, and Dan Spalding of Proxim, Inc.; to Peter Phillips of Socket Communications; and to Tom Carpenter and Mark Gentile of Odyssey Software, thank you for all of your hard work and for making sure that I was able to include your products in the book. I think—and I hope you do, too—that the book is much better for it.

To everyone at Sybex—to Peter Kuhns, who got this whole thing started, thank you for believing in me and in the book. To Brenda Frink and Gary Masters, thank you for taking over and for being so patient. To Bronwyn Erickson, thank you for keeping me organized and focused. Thanks also to Shannon Murphy, Project Team Leader; Robin Kibby, Electronic Publishing Specialist; Kristine O'Callaghan, Contracts and Licensing Manager; and Liz Paulus, Production Technician.

To Sally Engelfried, thank you for doing a great job editing my work. Thank you for being flexible and really working with me on some of the early issues. What more can I say but thank you?

To David Heskett and Jason MacLean, thank you for helping me to complete this book. You guys really pulled through for me, and I appreciate that. You both created some great code, and I hope when it's all done you're as proud of it as I am. But more important than that for me, I wanted to let you know that it was great to work this closely with you both after such a long absence. It's just not the same without you around.

To John Psuik, my technical editor, I've said it before, and I'll say it again: Thank you. Your knowledge, your attention to detail, and your generosity were invaluable. I look forward to working with you on future projects.

Thanks also to Fred Wilf, for your excellent advice.

Last but not most, thanks to my dad, who first suggested that I write.

CONTENTS AT A GLANCE

	<i>Introduction</i>	<i>xix</i>
PART I	<u>Inside the Core Operating System</u>	1
	Chapter 1: What Does CE Do for Me?	3
	Chapter 2: The Three Commandments of Writing for CE	41
	Chapter 3: To C or Not to C?	75
	Chapter 4: CE's Structured Storage	113
PART II	<u>Mastering the Developer's Tools</u>	142
	Chapter 5: CE Toolkit for Visual C++	145
	Chapter 6: Yes, It's Possible—MFC on CE!	173
	Chapter 7: Real MFC Applications Ported to CE	197
	Chapter 8: Visual Basic Toolkit for CE	225
	Chapter 9: A Real VB Application Converted to CE	243
PART III	<u>Advanced Topics</u>	278
	Chapter 10: RAPI: How the Outside World Talks to CE	281
	Chapter 11: How CE Talks to the Outside World	305
	Chapter 12: Third-Party Database Engines	335
	Chapter 13: Windows CE Case Studies and Cost Analysis	361

PART IV	Finishing Touches	373
	Chapter 14: Distributing Your CE Application	375
	Chapter 15: Microsoft's Logo Requirements	405
PART V	Appendices	432
	Appendix A: The C Runtime Library Functions of Windows CE	436
	Appendix B: The CE 2.0 API	472
	<i>Index</i>	585

TABLE OF CONTENTS

	<i>Introduction</i>	<i>xix</i>
PART I	<u>Inside the Core Operating System</u>	1
	1 What Does CE Do for Me?	3
	What Is CE, Anyway?	4
	Why CE Isn't Windows 98/NT	5
	What Is Unique to CE?	10
	CE Devices	12
	Palm-Size PC Devices	12
	Handheld PC Devices	25
	Handheld PC/Pro Devices	27
	Other Devices	34
	Accessories for CE	36
	Ethernet/Networking Solutions	36
	Bar Code Readers	38
	Summary	38
	2 The Three Commandments of Writing for CE	41
	The First Commandment: Your Application Must Use the Unicode Character Set	42
	Declaring Strings Using Unicode Types Rather Than Char Types	43
	Using Unicode Strings for All Text Literals	46
	Choosing the Correct RTL Functions for Unicode Strings	47
	Equipping Your Program to Handle Two Types of Text Files	48
	The Second Commandment: Your Application Must Be Low-Memory Aware	55
	Keeping the Size and Number of Static Variables to a Minimum	56
	Keeping the EXE File Size Low	56
	Checking the Return Result of Memory Allocation	57
	Mass-Allocating Your Application's Memory	58
	Handling the WM_HIBERNATE Message	60

The Third Commandment: Know Your Form Factor!	61
The UI of the Application Should Be Tailored to the Device	62
Maintaining a Single Codebase Is Nearly Impossible	67
Creating Your Own Conditional Defines	69
Creating a Runtime Platform Detector	70
Summary	73
3 To C or Not to C?	75
Finding Substitute Functions	77
calloc()	77
MoveToEx() / LineTo()	77
WM_RBUTTONDOWN	78
Changing the Program's Logic: try..catch and Exceptions	79
Writing Your Own Functions	80
FILE*	81
fopen()	83
fclose()	88
fgetc()	89
fputc()	90
fgets()	92
fputs()	94
fread()	95
fwrite()	96
fprintf()	97
fscanf()	99
fseek()	108
Summary	110
4 CE's Structured Storage	113
The Registry	115
Proper Uses of the CE Registry	116
Improper Uses of the CE Registry	116
The Windows CE Database Engine	119
Differences between CE's Database Engine and Familiar Database Engines	119
Similarities between CE's Database Engine and Familiar Database Engines	122
The CE Database Engine API	135
Summary	141

PART II	Mastering the Developer's Tools	142
5	CE Toolkit for Visual C++	145
	Using VC++ to Develop for Windows CE	146
	The Windows CE Toolkit for VC++	146
	The Platform SDKs	158
	MFC vs. SDK-Style Coding	160
	Storage Space	161
	Ease of Development	161
	A Sample SDK-Style Application	162
	Displaying a List of Running Tasks	163
	Switching to One of the Tasks	166
	Closing an Application	166
	One Last Snag: Keeping the List of Tasks Current	168
	Summary	170
6	Yes, It's Possible—MFC on CE!	173
	New Classes for CE	174
	CCeSocket	175
	CCeDBEnum	176
	CCeDBProp	176
	CCeDBRecord	179
	CCeDBDatabase	179
	Modified Classes	191
	Classes That Lost Functionality	192
	Classes That Gained Functionality	192
	Missing Classes	194
	Summary	194
7	Real MFC Applications Ported to CE	197
	The Shopping List Application	198
	Mechanical Issues of Porting	201
	Toolbars and Status Bars	201
	Printing Support	206
	The Grid	207
	Optimizing for CE	217
	One Final Surprise	220
	Summary	222

8	Visual Basic Toolkit for CE	225
	The Windows CE Toolkit for VB	226
	The Application Templates	226
	The Debugger	228
	The Runtime Files	229
	The Control Manager	230
	The ActiveX Control Pack	231
	The Standard VBCE Controls	235
	The Setup Wizard	236
	Changes in the Visual Basic Language	237
	A Sample Application	238
	Summary	240
9	A Real VB Application Converted to CE	243
	The Application: International ATM	244
	Porting the ATM Application: Not As Simple As It Looks	246
	Mechanical Issues of Porting	249
	The Optional Features	262
	Re-Adding the WAV Files	262
	The Country and Flag Bitmaps	266
	Optimizing for CE	270
	Eliminating a Form	270
	Eliminating Optional Controls and DLLs	274
	Optimizing the Code Itself	274
	Summary	276
PART III	Advanced Topics	278
10	RAPI: How the Outside World Talks to CE	281
	What Is RAPI, Anyway?	282
	General RAPI Management Functions	283
	System Information Functions	285
	Registry Access Functions	287
	File Access Functions	288
	Database Access Functions	292
	Miscellaneous Shell and System Functions	292
	A Sample RAPI Application	293
	Using Other Languages	298
	Summary	302

11	How CE Talks to the Outside World	305
	What's in the Box?	306
	The Hardware Aspect	306
	The Software Aspect	308
	Summary	333
12	Third-Party Database Engines	335
	Raima's RDM/CE for Data Storage	337
	Network and Relational Data Models	337
	Introducing HpcLadr	338
	Putting RDM to Work	341
	Sybase's Adaptive Server Anywhere	341
	Sybase's Adaptive Server Anywhere Sample Application	342
	Oracle Lite Introduction	347
	Oracle Lite for Windows CE	348
	Comparison of Features	356
	Summary	358
13	Windows CE Case Studies and Cost Analysis	361
	Study 1: Inventory Management System	362
	The Technical Issues	362
	Cost	365
	Conclusion of Study 1	366
	Study 2: Insurance Agents in the Field	366
	Solution	367
	The Technical Issues	367
	Cost	368
	Conclusion of Study 2	370
	Study 3: Choosing Your Development Machine	370
	Solution	370
	Cost	372
	Conclusion of Study 3	372
	Summary	372
PART IV	Finishing Touches	373
14	Distributing Your CE Application	375
	Creating Help for Windows CE	376
	The Two Types of CE Help Files	377

	The Single-File Help System	378
	The Multiple-File Help System: The HTC File	383
	The Multiple-File Help System: The HTP File	386
	Getting the Application to the User's Device	389
	The Cab Wizard Option	390
	InstallShield for Windows CE	400
	Summary	403
15	Microsoft's Logo Requirements	405
	What Is the Logo Program and Why Should You Care?	406
	So What?	407
	The Logo Requirements	408
	Installation	409
	UI Requirements	413
	Functionality Requirements	425
	File-Handling Requirements	429
	Summary	431
PART V	Appendices	432
	A The C Runtime Library Functions of Windows CE	436
	B The CE 2.0 API	472
	<i>Index</i>	585

INTRODUCTION

The war on the desktop is over.

Well, okay...maybe it isn't truly *over*, but the focus has definitely shifted a bit.

Suddenly, as if out of nowhere, there appeared a whole new class of Windows-based machines, unlike anything we'd ever seen before. These were the very first Handheld PC (HPC) devices. Soon after, there was a version 2.0 release of Windows CE, and with it, a greater sophistication of devices, as well as a greater variety of form factors and features.

But unlike other flavors of Windows, CE is a brand new Windows operating system, built entirely from scratch. There is no old DOS core—but then again, there's also no backwards compatibility. Instead, CE is a wild mixture of new and old elements. On the one hand, developers can write programs for it using a good portion of the Win32 API. On the other hand, it's amazingly compact and has some very tight memory requirements. Likewise, it's a version of Windows that runs on several different CPUs, yet it has a look and feel that's very close to that of desktop-based Windows.

Developing CE applications is also a strange mix of elements. One moment you're reminded that you're writing applications for a tiny device with a small grayscale screen, and the next moment you think it's just like Windows 98 or NT.

Yes, CE does have some limitations. But in this book, we'll be treating these limitations simply as challenges to be dealt with. Here we'll develop all kinds of applications designed to show off everything that you need to know in order to be a successful CE developer. We'll cover everything from new common controls to memory limitations to dealing with the Unicode character set.

Who Should Read This Book

This book was written with the experienced developer in mind. Perhaps you've been hearing or seeing a lot about Windows CE, and you've decided that you'd like the inside scoop with as little hassle as possible. Or perhaps you've already

decided CE is the way to go and you just need a way to dive right in. Or maybe you're interested in CE for some other reason completely.

In any event, this book assumes that you are an experienced Windows developer, with at least the ability to read C/C++, Visual Basic, or Delphi source code. This book assumes that you know the basics of Windows programming tasks such as creating a window, adding items to a list box, etc.

Instead of re-emphasizing the basics, this book concentrates on the differences between Windows 98/NT and Windows CE. It spells out exactly what you need to know to navigate the ins and outs of Windows CE.

What This Book Contains

This book uses numerous examples in C/C++, Visual Basic, and even Delphi to demonstrate Windows CE programming techniques. There is a sample program for each technique discussed in the book, and the source code for each of these programs can be found on the CD.

The book is organized into five major sections:

Part I: Inside the Core Operating System

Part II: Mastering the Developer's Tools

Part III: Getting the Most Out of Windows CE

Part IV: Finishing Touches

Part V: Appendices

Part I: Inside the Core Operating System

The main purpose of this section is to lay out exactly what Windows CE offers developers, both in terms of hardware and software. You'll jump right into code in Chapter 1, learning about the different hardware platforms and how to work with what each one offers. For example, how *do* you use a CE device to output graphics to a VGA monitor?

Chapter 2 details some of the main differences between Windows 98/NT programming and Windows CE programming. Some of the topics covered here include getting your application to look good on all of the CE form factors, working with the Unicode character set, and memory allocation issues.

Chapter 3 highlights some of the functions and features missing from Windows CE and demonstrates solid techniques for replacing or working around the missing functionality. For instance, when you're developing for certain CE platforms, you don't have a `stdio.h` available. Of course, this makes it very difficult to port existing code...unless, of course, you recreate the missing file-access functions of `stdio.h` as we do in this chapter!

In the last chapter of this section, Chapter 4, the book introduces a feature of CE not available on any other operating system: a built-in database engine. Windows CE is one of the few—if not the only—operating systems to offer a database engine as part of the operating system itself, and this chapter shows you how to make the best use of it.

Part II: Mastering the Developer's Tools

In this section, we look at how to best utilize the tools Microsoft makes available for us. Specifically, Chapter 5 discusses the Windows CE toolkit for Visual C++. You'll learn how to deal with common problems encountered with the toolkit. In addition, the chapter analyzes the tradeoffs between SDK-style C programming and MFC programming for CE.

In Chapter 6, we'll look at the differences between MFC on Windows 98/NT and MFC on Windows CE. Specifically, we'll look at some new classes to deal with CE's Database Engine and its communications model. And, as you may have guessed, there are also a few classes that don't exist under CE, and some that have undergone changes.

In Chapter 7, we'll see exactly what it takes to port a real Windows 98/NT MFC application to Windows CE-based MFC. We'll cover issues ranging from the basic port to printing support.

In Chapter 8, we explore the Windows CE toolkit for Visual Basic. There are some areas where the toolkit for Visual Basic can present some real surprises for anyone who's used to working with Visual Basic for Windows 98/NT, and this chapter looks at both the surprises and features.

In Chapter 9, **just as we did in Chapter 7**, we'll port a real Windows 98/NT Visual Basic application to Windows CE. We'll explore all aspects of what has to happen before **we can take a working desktop VB application** and produce a solid VB application.

Part III: Getting the Most Out of Windows CE

In this section, we take full advantage of the more advanced features of the CE operating system. Specifically, in Chapter 10, we look at how desktop machines and desktop applications can communicate with Windows CE devices. Using the Remote API (or RAPI), desktop applications can access a surprisingly rich set of features on the CE device. What's more, we'll even see how RAPI programs can be written in any desktop application development language—even a non-Microsoft product like Borland's Delphi can talk to CE devices, thanks to RAPI!

In Chapter 11, we'll explore all forms of Windows CE communications. CE devices have a multitude of possible ways to communicate with the outside world, and this chapter looks at all of them: the serial port, the IR port, modems, PCMCIA devices, and Winsock connections!

In Chapter 12, we look at ways to extend the usefulness of Windows CE devices by examining the features of some **third-party CE Database Engines**. Oracle, Sybase, and Raima Corporation offer the **three leading third-party database engines** available for Windows CE. Each of these **vendors has created a sample** application and a brief article describing the application and/or the features of their engine.

Chapter 13 offers something few programming books do: pseudo case studies. Based on real case studies but modified to be general enough to apply to anyone, these case studies present a view of what it takes to put together a full Windows CE-based solution for your clients, whether you're a consultant or a part of a corporation's IS department.

Part IV: Finishing Touches

In this section, we look at some additional touches you might want to put on your application. In Chapter 14, we look at what it takes **to create a help system**—whether it's one simple file or multiple files—for your CE application. Also, we'll look at what it takes to create a simple setup program for your application, either by hand or using a third-party tool.

Then, in Chapter 15, we'll explore what the Windows CE Logo Program is all about and how it can help you. We'll see how, in addition to ensuring that your application has a standard consistent look and feel, the Logo Program can help you market your CE application much more effectively.

Part V: Appendices

To round out the book, we've included two of the most useful appendices possible. Appendix A details all of the C runtime library functions guaranteed to be available for *all* CE versions/platforms. Using this appendix as a guide, you can safely pick and choose which runtime functions to use if you're compiling an application for more than one CE version/platform. As an added benefit, the header file and a simple example are provided for each and every function.

Appendix B takes a similar approach to the Windows CE API. In this appendix, we detail every single API call that is unique to Windows CE 2. For every function, we list the header file, a description of the function's purpose, and a sample demonstrating the use of the function.

Perhaps the best feature of this section is that both of these appendices are provided on the CD in PDF form, so that you can easily load each appendix, browse to the function you want, and copy/paste the function's example right into your own code.

About the Examples

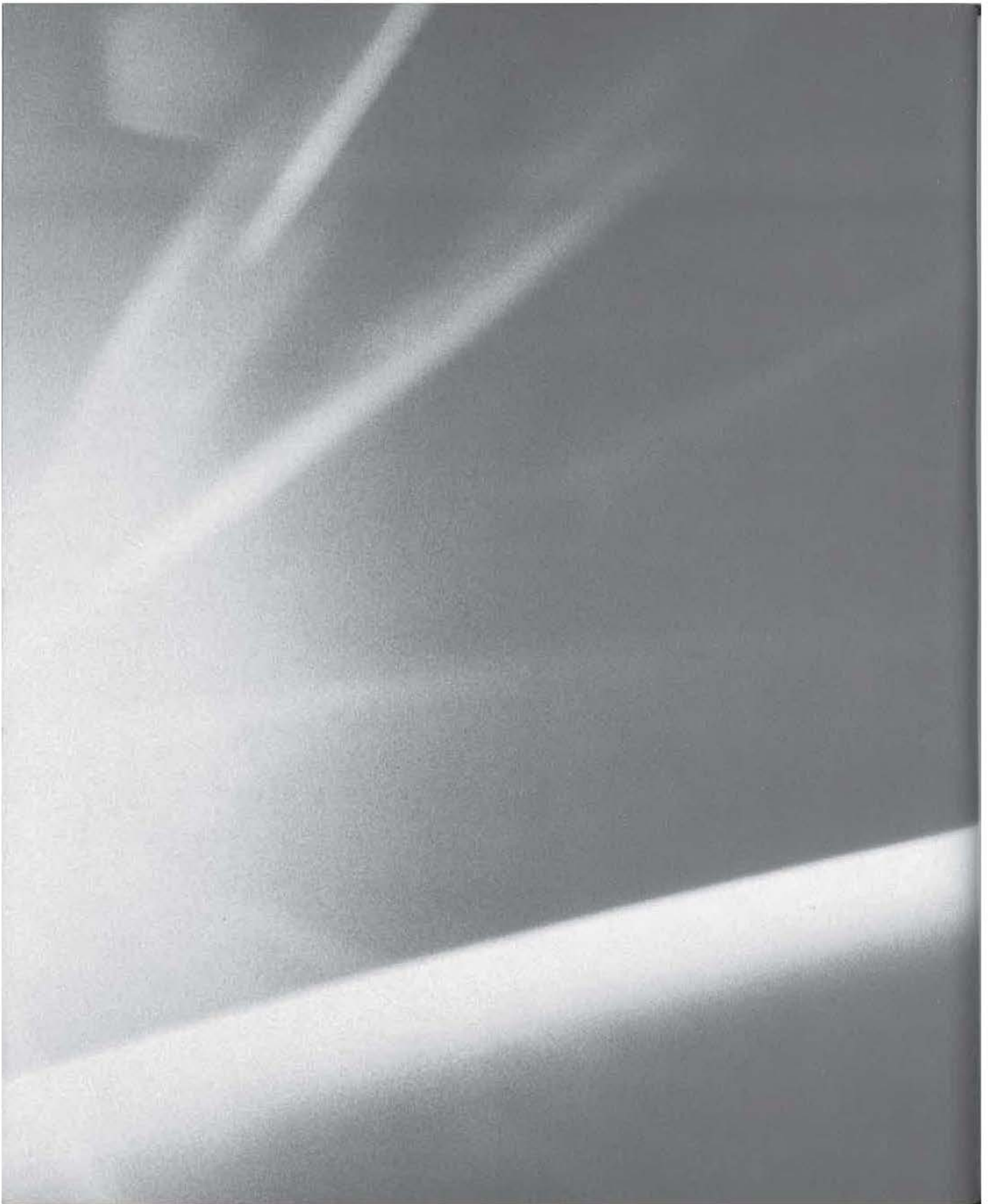
The majority of the examples in this book were written in Microsoft Visual C++ 6 with the Windows CE Toolkit for VC++ 6. There are also examples that were created in Visual Basic 5/6, with the appropriate version of the Windows CE toolkit for Visual Basic. Also, in Chapters 10 and 14, there are two samples developed using Borland's Delphi product.

As of this writing, Microsoft has the only C/C++ based compiler that builds applications for the Windows CE operating system. Although there is talk on the newsgroups about a possible GNU compiler port, no such product has yet been released.

PART I

Inside the Core Operating System

- CHAPTER 1: What Does CE Do for Me?
- CHAPTER 2: The Three Commandments of Writing for CE
- CHAPTER 3: To C or Not to C
- CHAPTER 4: CE's Structured Storage



CHAPTER

ONE

1

What Does CE Do for Me?

- What is CE, Anyway?
- Why CE isn't Windows 98/NT
- Features Unique to CE
- CE Devices: Shapes, Sizes, and What You Need to Know About Each One
- CE Accessories

In a promotional video sent out by Microsoft late last year, Harel Kodesh of Microsoft's Consumer Appliances Group was quoted as saying, "CE was built with one thing in mind, and that is powering information appliances." These information appliances give people such a degree of freedom and power in their personal and business lives that they've created an instant market for new software.

A large part of what makes these devices so powerful is that they're not just organizers or wizards. Yes, every CE device comes with pocket versions of Outlook, but you can run real applications on the devices, too.

In many ways, Windows CE puts a virtual desktop computer in the palm of your hand or in your pocket. It's that capability that makes Windows CE a "shrunk head" version of Windows 98/NT, as we'll see in this chapter. We call CE a shrunk head version of Windows because you have to work with far less memory than you are used to, smaller screens, fewer colors, etc.

The questions we'll be answering in this chapter are, "What does CE do for me?" and, "How can I get my programs ported to CE?" To answer them, we'll look at the different types of CE devices and what each one offers in terms of functionality. In many cases, you'll be surprised to find out just how powerful a CE device can be. Despite their small sizes, their CPUs rival those we have in our desktop computers.

We'll also be looking at some of the CE-compatible hardware you can add to your application to make it a more complete CE solution. For instance, what if you need to add some kind of secondary storage to your device because 8MB just isn't enough? That said, let's get right into it with a look at the different device types.

What Is CE, Anyway?

Windows CE is a stripped-down version of Windows 98/NT that's been engineered specifically for small, low-resource, portable devices such as:

- The Palm-size PC
- The Handheld PC
- The Handheld PC/Pro

From an end user's point of view, it is a miniature Windows, complete with a pocket version of Outlook, Start button, and Taskbar. However, there are a few significant differences between CE and 98/NT. After all, if you're going to take the world's most popular desktop operating system and make an embedded, tiny version of it, you end up sacrificing some features.

NOTE

According to Microsoft, CE doesn't officially stand for anything at all; however, it's generally assumed that CE means *consumer edition* or *compact edition*.

Why CE Isn't Windows 98/NT

CE and Windows 98 differ in five major ways:

- CE won't run your existing programs
- CE has serious memory constraints
- CE has a reduced runtime library and API
- CE devices usually don't have a mouse
- CE hardware is not very standardized

Of course, there are ways to deal with each of these differences, and we'll be looking at them in more detail in later chapters. For now, here's what all this means.

CE Won't Run Your Existing Programs

Perhaps the biggest difference between the two operating systems is the fact that CE will not run *any* existing Windows applications. This means that all programs must be recompiled especially for the Windows CE operating system. This is a striking departure from the backwards compatibility goals of Windows 98 and NT.

CE Has Serious Memory Constraints

In recent years, the amount of memory in any given PC has gone up astronomically. That's why, under Windows 98, there is usually plenty of memory to go around. Under Windows CE, however, that's not always the case. And, since this

is one of the main reasons you can lose your way, we'll pay a lot of attention to it. When it comes to CE's memory constraints, there are three key issues:

- Reduced amounts of physical memory
- User control over memory availability
- Operating system requests to your program

Reduced Amounts of Physical Memory Most CE devices ship with comparatively little physical memory to begin with. The Palm-size PC devices, for example, usually have just 4 or 8MB. For a Windows 98 machine, such a small amount of memory would be ridiculous.

NOTE

However, the amount of physical memory in CE devices is set by the manufacturer, and there really isn't a whole lot you, as a developer, can do about it. Most of the makers of Palm-size PC devices seem to feel that 4MB is an adequate amount of RAM for their devices, and, truth be told, they're correct. The operating system and most CE applications will run quite comfortably under these conditions. While it is possible to upgrade the amount of RAM in many of these devices, asking your users to upgrade is not a preferred option. Before doing that, try eliminating some noncritical features to reduce the requirements of your application.

User Control over Memory Availability When Microsoft and the various CE device makers collaborated on the designs and standards for CE devices, they decided that CE devices would not come with hard drives. This was probably so they could hit a lower price point.

To compensate for the lack of true "secondary storage," Microsoft chose to make the physical memory (RAM) serve two purposes: our more traditional understanding of *RAM* (program memory) and a sort of *RAM-disk* (storage memory).

The result of this is that you do not have complete control over how much memory is available for your program. Unfortunately, the user of the device largely controls the allocation of memory into program memory and storage memory. And, although the user can't adjust the settings so that no programs will run, they can adjust it so fewer programs will run. Or, they can adjust it so programs with greater memory needs don't run properly.

With Windows 98, of course, the user has no such control. The amount of physical memory is the amount of RAM, and the size of the disk is the size of the disk.

On the other hand, if enough memory is available on your CE device, your program will continue to steam right along, with the advantage that all the memory it needs is already reserved for its use.

Operating System Requests to Your Program All CE programs must be able to handle low-memory situations. Specifically, they should try to relinquish some of the memory they're using whenever the operating system requests it. This is a complete departure from what you're used to doing in Windows 98.

For this reason and others, you are going to have to pay infinitely more attention to memory when writing your Windows CE programs. In fact, it may not be going too far to say that memory is, perhaps, the most important CE issue you have to understand before tackling your program. We'll deal with these memory issues in more detail in Chapter 2.

CE Has a Reduced Runtime Library and API

A big difference between Windows CE and the Windows 98/NT platforms is the scaled-down C/C++ runtime library (RTL) and API of Windows CE. Many of the more common ANSI functions have been substituted with Windows API functions or removed entirely. In other words, your existing Windows programs probably won't compile for CE the first time out, and you'll have to make a few modifications. In Chapter 3, we'll look at some of the functions that are missing from the RTL and API, and what we can do to fix this situation.

CE Devices Usually Don't Have a Mouse

Under Windows 98, the mouse is used to control an application and its appearance—that is, select objects, resize windows, drag things around the screen, and navigate menu items. Very few Windows 98 programs can be reasonably operated without a mouse.

However, most Windows CE devices do not have a mouse. Instead, CE devices have a penlike tool called a *stylus*. Unlike a mouse, a stylus has no constant visual representation on the screen. In other words, there's no cursor.

NOTE

There is only one CE device that has a mouse: the Hewlett-Packard Jornada HPC/Pro device, which we'll be looking at later in this chapter.

What You Need to Know about the Stylus Unlike the always-present mouse of Windows 98/NT, CE's stylus only interacts with the operating system and the applications when the user taps (or taps and holds) it to the screen. CE users can still efficiently select objects, navigate menu items, etc. However, there are some areas where the stylus-as-a-mouse substitute just doesn't cut it; for instance, resizing windows and right-clicking. In fact, resizing windows is so difficult with a stylus, Microsoft removed that functionality completely! This means that no CE windows can be resized at all. Under CE, windows are either maximized or fixed at whatever size they're created.

Right-clicking is available, but it can be a chore for a user. When the user wants to right-click on something, they must hold down the Alt key and tap the screen with the stylus at the same time—a very awkward procedure. It's so inconvenient, in fact, that you probably won't want to use pop-up/right-click menus at all.

NOTE

If you still want to include right-click functionality in your application, see Chapter 3 for a demonstration of how to trap the Alt-click combination.

The upshot of all this is that it's not a good idea to have your program rely too heavily on right-click operations. Similarly, unless your application is a dialog-based app that doesn't require the full screen, you'll probably want to launch most of your programs in a maximized window, since there's no way for the user to resize the window with the stylus.

WARNING

Using the Alt-click combination for right-click functionality only applies to the HPC and HPC/Pro devices—Palm-size PCs don't have any mechanism for right-clicking at all!

CE's Hardware Is Not Very Standardized

With Windows 98, your programs are likely to run on fairly standardized hardware. Everybody's running an Intel (or compatible) chip, some reasonable amount of memory, a 16- or 256-color display, and at least 640 × 480 resolution. With very few limitations, it's expected that everything's going to work the same on any Windows 98 machine.

That's not the case under CE, though. Each manufacturer designs their devices differently, and CE devices don't have a single standard configuration as PCs do. For instance, some CE devices run on a Hitachi microprocessor, while others run

on a chip from NEC. Some devices have a color display, while others have only grayscale. And some devices run 640×240 display, while others run a 240×320 display.

NOTE

With the exception of the non-Intel platforms of Windows NT, Windows CE is one of the few Microsoft products not designed exclusively for the x86 Intel platform.

The key differences between manufacturers' CE devices are:

- Chip type
- Display type
- Display size

Chip Type When using Microsoft's CE development tools, you are required to compile your program once for each of the chip types supported by CE. It's then up to you to test your program on each of the chip types, and it's up to your install program to copy the correct executable file when it uploads the file to the user's device. Of course, this also means that unless you want to tie your application to a specific processor or device, you probably want to stay away from any assembler.

Display Type Some of the Handheld PC models have a color display; others have only grayscale. So, unless you are targeting a specific device, model, and manufacturer (e.g., Casio PA-2500), you probably don't want your program to rely too heavily on any color-related operations. Clearly, you wouldn't want your program to tell a user with a grayscale device to "click the red button."

Display Size If you design your application for the Handheld PC devices, they will probably look fine on all of those devices. However, if you merely recompile for the Palm-size PC devices, the application probably won't even be useable. This is because Handheld PCs generally have a display of about 640×240 , whereas most Palm-size devices have a display of only 240×320 . This makes it challenging to design one application for multiple devices.

While it might be *possible* to simply recompile an HPC application for the PPC platform, it probably wouldn't be a good idea. If you did, half of your program's main window would likely be outside the visible area of the display! In the next chapter, we will examine how to change a few simple elements of your program in order to make this recompilation possible.

Now that you know how CE isn't Windows 98/NT, let's take a look at some of the features that make CE unique and special.

What Is Unique to CE?

Although there are a lot of things CE doesn't do—from a Windows 98/NT developer's perspective, that is—there are also a lot of things that it does do. And, given the small size of CE, some of these items are quite impressive.

Core OS Features

First, there are the core operating system features. Windows CE is a 32-bit, multi-tasking, multithreaded operating system. It will address memory using the flat memory model, run multiple programs at once, and support multiple threads in a program. In other words, its architecture is based on 98/NT's architecture.

Networking- /Internet-Related Features

CE supports Winsock 1.1 for writing custom Internet applications and supports portions of the WinInet library so that your CE application can retrieve documents via HTTP operations. CE also supports network shares accessing on Desktop/server-based Windows machines. Later in this chapter, we'll see how it's possible to get your CE device onto your network using some third-party hardware.

Communications

Windows CE devices support communication over a serial port or an Infrared (IR) port. Because the serial port is a standard serial port just like the one(s) on your PC, you can connect standard modems and other serial devices to a CE device's serial port. This can be a big cost saving to your users, who might otherwise believe that only expensive cellular modems will work with CE, when a standard external modem will do the trick.

The IR port is especially efficient if your application needs to transfer data between two CE devices. While it's possible to transfer data between serial ports, the IR port is much faster and much more convenient to the end user. After all, there are no cables to connect—you just point the two devices at each other and click OK.

Databases

When doing database development for Desktop-based Windows platforms, you must select and purchase some kind of third-party database engine, or you must choose a development tool that includes a database engine. One of the qualities

that makes CE unique, however, is that the operating system itself has support and functionality for creating and accessing databases from within your programs. CE is the *only* operating system with database functionality built right into the OS itself. Granted, it's not a very sophisticated database engine, but it should meet most simple needs. This will save you the time and money associated with choosing and purchasing a third-party engine.

Just How Powerful Is a CE Device, Anyway?

Most developers are surprised to learn just how powerful a CE device really is. In fact, many of the CE devices sport CPUs that, while impressive by themselves, are even more impressive when compared to a Desktop system's CPU.

For instance, the Casio E-11 features an NEC VR4111, which is a 64-bit chip that runs at 80MHz! The dhrystone rating on this chip is 105. (Dhrystone is a benchmarking test used to rate how fast a chip can do integer-based operations. The score indicates millions of instructions per second, so a higher number is better.)

Similarly, the Hewlett-Packard Jornada HPC/Pro device uses a strongARM processor, which is a 32-bit chip that has a dhrystone rating of 220. Considering that a typical 133MHz Pentium has a dhrystone rating of about 240, these are very impressive numbers!

Here's a practical example, however, that's even more telling than these ratings. When the Palm-size PCs were first released, CE development newsgroups debated whether or not the PPC devices would be powerful enough to play MP3 music files. As you probably know, MP3s are very tightly compressed, very high quality sound recordings. Because they are so tightly compressed, their playback is an incredibly CPU-intensive task—so much so that it's generally assumed that you need a Pentium running at 100MHz just to get decent, uninterrupted listening. Many people argued that the CPUs of the Palm-size devices simply could not process the decompression fast enough to be practical for the playback of MP3s.

After several months, the debate was finally settled when XAudio of Mountain View, California (<http://www.xaudio.com>) released their XAudio MP3 player for Windows CE. Although the product is still in beta form as of this writing, it does, in fact, play MP3 files on any Palm-size PC device. And it does it very well!

If a Palm-size PC can handle the complex operations required to play an MP3 file without skipping and dropping sections of the recording, it ought to be powerful enough to do just about anything (within the limits outlined in this and the next several chapters)!

CE Devices

There are currently four distinct types of CE devices, each with its own purpose:

- Palm-size PC devices
- Handheld PC devices
- Handheld PC/Pro devices
- Devices that don't fall into the above categories

Palm-Size PC Devices

Palm-size PCs (or PPCs) are usually the smallest of the Windows CE devices, with approximately the same dimensions as a package of 3 × 5-inch index cards. They have a long, narrow, portrait-oriented display; no true keyboard; a touch screen; and integrated voice recording features. In addition, all PPC devices have a Compact Flash slot, which can be used to add additional memory or peripheral devices to the PPC.

The shape and portability of PPC devices make them ideal for applications that allow the user instant, "on the road" access to some specific kind of data, with a minimum of data entry. Figure 1.1 shows an example of a Palm-size PC device, the Casio E-11.

There are several features that make PPCs different from other CE devices:

- Display
- Lack of a true keyboard
- The Ink control
- The CapEdit control
- Voice recording and playback services

FIGURE 1.1:
The Casio E-11



Copyright Casio © 1999

Display

The first thing that makes the display on a PPC so unique is that it is portrait-oriented, as illustrated in Figure 1.2. As you probably realize, this is a sharp contrast to the Desktop world, where everything is more landscape-oriented than anything else.

In addition to its unusual orientation, the display on a PPC is very small, typically **around** 240×320 pixels. **This, too, contrasts** to the Desktop world where we can expect at *least* 640×480 . **Obviously, this** means that a lot more thought must go into the design of dialogs and windows to get them to look decent in such a tight space.

The final element of the display that makes it unusual when compared with our Desktop programming is that it is only a four-color grayscale LCD. In the Desktop world, we can count on 16 colors at the bare minimum.

FIGURE 1.2:

Portrait vs. landscape orientation

**NOTE**

As this book was going to press, HP announced a color PPC device.

Together, these elements mean that you're going to have to pay much more attention to your visual design. In many cases, you will end up having to reduce the width of any control on your windows or dialogs simply because you've got to fit everything into such a small space. Figures 1.3 and 1.4 show sample dialogs designed for Windows 98/NT and a Palm-size PC, respectively.

FIGURE 1.3:

A dialog designed for Windows 98/NT

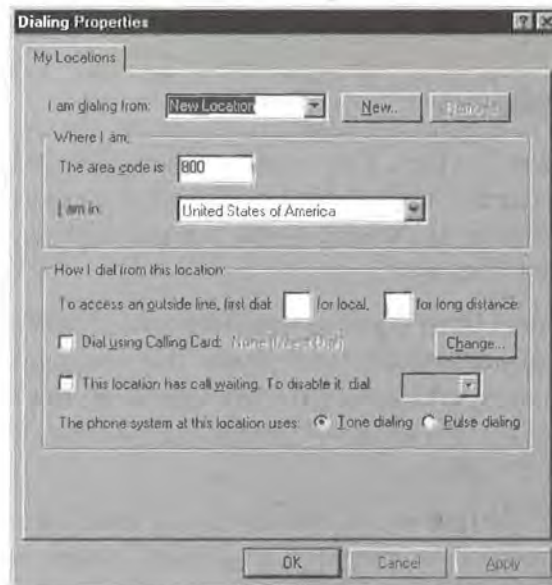


FIGURE 1.4:

A dialog designed for a PPC device



As you can see, making this dialog work on a PPC involved stacking the various controls a little more closely and making each control just a little bit narrower. If we wanted to, we could also have used a tabsheet control to help break up the various controls into two distinct sections.

Lack of a True Keyboard

Palm-size PCs do not have a physical keyboard. Instead, they have a “virtual keyboard,” a pop-up panel, called the System Input Panel, that displays letters and numbers in a QWERTY arrangement. The user taps out the keys on this panel with the stylus. The System Input Panel can be raised and lowered as needed, as shown in Figures 1.5 and 1.6, respectively.

As you can imagine, this can create havoc for your program. When the Input Panel is raised, it occupies roughly one-third of the display area. To accommodate it and to give your program some control over how it interacts with the Input Panel, there are two modifications you need to make to any PPC application:

- Programmatically raise and lower the Input Panel
- Detect when the user pops up the Input Panel

FIGURE 1.5:

The Input Panel raised



FIGURE 1.6:

The Input Panel lowered



Programmatically Raise and Lower the Input Panel You need to be able to raise and lower the Input Panel programmatically because not all of your programs will use the Input Panel in the same way, or to the same extent. For example, if you designed a blackjack game for CE, you would probably want to hide the

Input Panel when the game started because the user probably won't need the keyboard in a blackjack game.

Similarly, if you designed some kind of data entry application, it would make your program friendlier if you showed the keyboard when the program started. For these reasons, you need to be able to control the Input Panel's state. In Chapter 15, we'll explore exactly how to do this.

Microsoft already provides code to lower the Input Panel, so you only have to write the code to raise the Input Panel. The original lowering code, as provided by Microsoft, looks like this:

```
BOOL LowerSip( void )
{
    BOOL fRes = FALSE;
    SIPINFO si;
    memset( &si, 0, sizeof( si ) );
    si.cbSize = sizeof( si );
    if( SHSipInfo( SPI_GETSIPINFO, 0, &si, 0 ) )
    {
        si.fdwFlags &= ~SIPF_ON;
        fRes = SHSipInfo( SPI_SETSIPINFO, 0, &si, 0 );
    }
    return fRes;
}
```

What's happening here is that you're first filling a structure (*si*) of type *SIPINFO*. For our purposes here, though, you're only interested in one member of the structure, *fdwFlags*, which you'll use to set the state of the panel. There are three possible values for the *fdwFlags* member:

- *SIPF_ON* The panel is visible.
- *SIPF_DOCKED* The panel is attached to the Taskbar.
- *SIPF_LOCKE* The panel is locked in place, and its state cannot be changed by the user.

NOTE

These values can then be combined by using the AND (&&) and OR (||) operators. Although the second two *fdwFlags* values are not of interest to us here, they do have meaning for other programs. For instance, with regard to the second value *SIPF_DOCKED*, some Input Panels allow themselves to be detached from the toolbar and dragged around the screen much like a dockable toolbar.

To lower the panel, all you have to do is remove the `SIPF_ON` value from the `fdwFlags` member. Do this by using `AND` on the current value of `fdwFlags` with the logically negated `SIPF_ON` code. Then, write your `si` structure back to the Input Panel, and it lowers itself. Return the result of your work, and you're done. The following code is an example of this procedure:

```
si.fdwFlags &= ~SIPF_ON;
```

Then, write your `si` structure back to the Input Panel, which then lowers itself:

```
fRes = SHSipInfo( SPI_SETSIPINFO, 0, &si, 0 );
```

Return the result of your work, and you're done:

```
return fRes;
```

Using this piece of code as a model, you can see that to get a routine that *raises* the Input Panel, all you need to change is the line that sets the value of `fdwFlags`. In other words, instead of removing the `SIPF_ON` flag, use `OR` on `fdwFlags`, as in the following:

```
si.fdwFlags |= SIPF_ON;
```

Or, when put into the full routine:

```
BOOL RaiseSip( void )
{
    BOOL fRes = FALSE;
    SIPINFO si;
    memset( &si, 0, sizeof( si ) );
    si.cbSize = sizeof( si );
    if( SHSipInfo( SPI_GETSIPINFO, 0, &si, 0 ) )
    {
        si.fdwFlags |= SIPF_ON;
        fRes = SHSipInfo( SPI_SETSIPINFO, 0, &si, 0 );
    }
    return fRes;
}
```

But raising and lowering the Input Panel is only one of the two requirements you need to take care of to manage the Input Panel's interaction with your programs. The second requirement is much more involved.

Detecting When the User Pops Up the Input Panel You've got to be able to detect when the user pops up the Input Panel because the Input Panel occupies so much of the screen area when it's visible. Your application will be much more

user-friendly if it resizes itself and allows the user to scroll the window whenever the Input Panel is shown.

You can observe this behavior in the Owner control panel applet. With the Input Panel lowered, all of the controls are visible and the window does not have a scroll bar, as shown in Figure 1.7. When you show or raise the Input Panel, the Owner window actually resizes itself and adds a scrollbar on its right side, as shown in Figure 1.8. The user can then scroll to that portion of the window that would otherwise be obscured by the Input Panel.

FIGURE 1.7:

The Owner applet with the Input Panel lowered



FIGURE 1.8:

The Owner applet with the Input Panel raised



As you can probably guess, CE provides a way to detect the state change in the Input Panel, and Microsoft even provides a sample of how to do this. However, their sample is incomplete in that it doesn't show how to add the scroll bar to the application based on the state of the Input Panel. To get your application working as smoothly as possible, you'll need to take their sample and make it complete.

This can get a bit confusing, so let's start by looking at some simplistic pseudo-code for the logic at work here:

```
If we're showing our scroll bar
  and If the input panel is not visible
    Then let's hide our scroll bar.
Otherwise (we're not showing our scroll bar)
  and If the input panel is visible
    Then let's show our scroll bar.
```

WARNING

This logic assumes that your application doesn't normally need a scroll bar and that it only displays one when the user pops up the Input Panel.

Start by adding a handler for the Windows CE message that tells you one of the system settings has changed. When the Input Panel's state changes, CE will send you a `WM_SETTINGCHANGE` message with the `wParam` value set to `SPI_SETSIPINFO`:

```
case WM_SETTINGCHANGE:
    switch( wParam )
    {
        case SPI_SETSIPINFO:
```

Then, just as you did when you wanted to show or hide the Input Panel, fill your `si` variable (of type `SIPINFO`) with information about the Input Panel:

```
    memset( &si, 0, sizeof( si ) );
    si.cbSize = sizeof( si );
    if( SHSipInfo( SPI_GETSIPINFO, 0, &si, 0 ) )
    {
```

Your next task is to check whether or not your application already has a scroll bar visible. Do this by retrieving the current window style by calling `GetWindowLong()`

and passing `GWL_STYLE` as the second parameter. `GetWindowLong()` returns a Long integer; store it into a variable called `lWndStyle`:

```
lWndStyle = GetWindowLong(hWnd, GWL_STYLE);
```

Then check to see if you're already displaying a scroll bar by comparing `lWndStyle` against `WS_VSCROLL` style:

```
if (lWndStyle & WS_VSCROLL)
{
```

If you are already showing a scroll bar, then you've got to decide whether or not to hide it based on whether or not the Input Panel is hidden. You must then check the `fdwFlags` value of your `si` (`SIPINFO`) variable to see if the Input Panel is still visible:

```
if (!(si.fdwFlags & SIPF_ON))
{
```

If the Input Panel is hidden, use AND on `lWndStyle` and the logical negation of `WS_VSCROLL` together in order to remove the `WS_VSCROLL` style and, therefore, the scroll bar itself:

```
lWndStyle &= ~WS_VSCROLL;
```

Finally, make your changes take hold by calling `SetWindowLong()`, passing in the newly modified style:

```
SetWindowLong(hWnd, GWL_STYLE, lWnd-
Style);
}
}
```

If, however, you're not showing your scroll bar and the Input Panel is visible, show your scroll bar by using OR on the `WS_VSCROLL` style with your `lWndStyle`:

```
else
{
if (si.fdwFlags & SIPF_ON)
{
lWndStyle |= WS_VSCROLL;
}
```

Then—just as you did above—make your window take on the new style by calling `SetWindowLong()`, passing in the new `WndStyle`:

```

        SetWindowLong(hWnd, GWL_STYLE, Wnd-
Style);
    }
}

```

Next—and this is the core piece from the Microsoft sample code—resize your window to whatever the visible screen area is. The `SIPINFO` structure, which has as one of its members a `RECT` representing the visible desktop area, allows you to do this. In code, the entire operation can be done in one call to `MoveWindow()`:

```

        MoveWindow(
            hWnd,
            si.rcVisibleDesktop.left,
            si.rcVisibleDesktop.top,
            si.rcVisibleDesktop.right -
si.rcVisibleDesktop.left,
            si.rcVisibleDesktop.bottom -
            si.rcVisibleDesktop.top,
            TRUE );
    }
    break;

```

Finally, to ensure that your application can detect the state of the Input Panel when it starts, you must manually send your program a `WM_SETTINGCHANGE` message in the `WM_CREATE` message handler:

```
SendMessage(hWnd, WM_SETTINGCHANGE, SPI_SETSIPINFO, 0);
```

You're finished. Now your applications will behave correctly and interact as efficiently as possible with the Input Panel.

NOTE

You'll also need a message handler for the `WM_VSCROLL` message to handle any requests that the user might make while the scroll bar is visible. This code is included on the CD in the samples for this chapter.

The Ink Control

Because most of the input to the system is done via a touch screen, it seems only natural that the PPCs have a control that accepts handwriting input. It's called the

Ink control, and it will record any mixture of handwritten and typewritten data. This control is especially useful in applications where a signature is needed.

WARNING

The Ink control does not do any handwriting recognition; any handwritten data that it stores is stored as graphical, rather than textual, data.

Working with the Ink control is a bit trickier than you might imagine. In fact, there are three steps required just to create an individual Ink control within your own application.

1. Initialize the common control library:

```
InitCommonControls();
```

2. Specifically initialize the Ink (or InkX as it is alternately called) control:

```
InitInkX();
```

3. Call `CreateWindow()`, specifying `WC_INKX` as the window class:

```
hInk = CreateWindow(WC_INKX, TEXT(""),
                   WS_VISIBLE | WS_CHILD | WS_BORDER,
                   5, 90, 200, 160, hWnd,
                   NULL, hInst, NULL);
```

WARNING

Note that `WC_INKX` is not an `lpClassName` string, but rather a defined constant to be passed in place of the `lpClassName` string.

The CapEdit Control

To allow for easier data entry, Microsoft added a special kind of edit control called the CapEdit control. The CapEdit control looks and behaves like a standard Edit control, with one exception: the first letter typed in the control is automatically capitalized. This allows users to enter data much faster because they do not have to press the Shift key before entering the first letter.

There are a number of situations where this is important, such as when a user enters an address where, typically, the first letter of every piece of data is capitalized. If the user doesn't have to worry about capitalization, it's easier to enter addresses.

Using the CapEdit control is very easy—all you have to do is call `SHInitExtraControls()` to initialize the control, then call `CreateWindow()`, specifying "CAPEDIT" instead of "EDIT" as the class name, as in this example:

```
#include <aygshe11.h> //for SHInitExtraControls

//...

SHInitExtraControls();
hEdit = CreateWindow(TEXT("CAPEDIT"), TEXT(""),
                    WS_VISIBLE | WS_BORDER | WS_CHILD,
                    5, 5, 150, 21, hWnd,
                    NULL, hInst, NULL);
```

Voice Recording and Playback Services

Although all of the CE devices offer some kind of voice recording software, the PPC platform is the only CE platform that exposes this functionality to developers in the form of a control. The "control" in this case is a dialog box that can be launched from within a program. The dialog will then send notification messages to the calling program, indicating the user's actions.

The PPC Taskbar

Although the PPC does follow the Windows 98/NT user interface very closely, there is one subtle difference that can take some getting used to. PPC devices have a Taskbar, but running applications don't show up there as they do under desktop versions of Windows. The reason for this is that the Taskbar is already so narrow because of the form factor of the device itself that there isn't enough room to maintain a button for each application that the user starts.

Instead, it's Microsoft's intent that a PPC program shouldn't really be closed or exited in the normal fashion. As you can see in any of the Microsoft Pocket Outlook applications, it's preferred that a program not have a File ➤ Exit menu, or a system close (X) button.

Instead, as we'll be exploring very shortly, applications are supposed to stay running indefinitely, hidden in the background. It is the responsibility of the developer to ensure that the user cannot start two copies of the same program. When the user tries to start a program, the program should first check if there's already a copy of itself running. If it finds another copy, it should bring that to the front rather than opening a new copy of itself. Likewise, when the system is running low on memory, it will ask the running programs to close down.

Continued on next page

Of course, it's worth pointing out that many developers have ignored the PPC style of program management and have exposed all of the standard means of closing an application. That's probably because they believe that users are more comfortable with a program that can be closed. The examples in this book generally include at least the system close (X) button, as this seems more comfortable to use while debugging and testing.

Handheld PC Devices

Handheld PCs (or HPCs) are usually the midrange models, being moderately larger than the PPCs (about $4 \times 7\frac{1}{2} \times 1$ inches), yet possessing less-than-full-size keyboards. Designing programs for HPCs has a more familiar feel to it since the displays are landscape-oriented and applications are represented on the Taskbar when running. In addition, most HPC devices have both a Compact Flash slot and a PCMCIA (Type II) slot, either of which can be used to add additional memory or peripheral devices to the HPC. However, like the PPCs, HPC devices have a touch screen instead of a mouse.

The shape and portability of HPC devices make them ideal for pocket or mini versions of some Desktop applications, with reduced sets of features and not as much data entry as, say, a Desktop application might require. Figure 1.9 shows an example of a Handheld PC device, the Casio PA-2500.

There are several noteworthy features that make HPCs different from other CE devices:

- Display
- A true keyboard
- The HTML Viewer control

FIGURE 1.9:

The Casio PA-2500



Copyright Casio Business Solutions Group © 1999

Display

The first thing that makes the display on an HPC different from other CE devices is that it is landscape-oriented, just like Desktop machines. In addition, the displays on HPCs typically measure about 640×240 . That's twice as large as the screens on the PPC devices. Of course, that's still quite small when compared to resolutions available on Desktop machines, but this increase in size does help—especially when you're designing an application that displays lots of information and you really need the extra room.

The final characteristic of HPC displays that makes them unique is that there are *some* color HPCs. However, not all HPCs have color displays, so it's still a good idea to play it safe and not make any assumptions about your user's ability to see colors.

A True Keyboard

Handheld PCs have an actual physical keyboard. From a development standpoint, this means that you can design the HPC version of your program to require some data entry. Of course, because the keyboard is not really a full-size keyboard, you won't want to insist on intensive data entry.

The HTML Viewer Control

Because HPCs have a larger display and are therefore better for viewing data, they have a special control called the HTML Viewer control. As you can guess, it's a control that you can use in your own programs to display HTML pages. The tricky part about this control is that it notifies your application of any HTML-based event. For example, when an HTML page contains a request for an image, the HTML Viewer control notifies your application, passing in the URL of the image, so that you can then download and display the image. Similarly, when a user clicks on a link, the HTML Viewer control notifies your application, passing in the text of the link, so that you can then choose to retrieve and display the requested page.

NOTE

The HTML Viewer control does not actually manage the retrieval of documents over the Internet. Rather, it is a means of displaying HTML data. Your application is still responsible for actually fetching the document from a server and then sending it to the HTML control.

WARNING

Be aware that if you're developing for both the PPC and the HPC that the HPC does not support the CapEdit control! The Ink control is supported, although it is undocumented as of this writing.

Handheld PC/Pro Devices

Handheld PC/Pro devices (or HPC/Pros) are the top-of-the-line models. They are typically the size of a small laptop, about 10 × 7 × 1½ inches. HPC/Pros have a laptop-size keyboard, which is more than adequate for any amount of data entry by the user. Their screen is landscape-oriented and huge compared to the other CE devices, a full 640 × 480.

Further, at least one HPC/Pro model has a mouse pointer instead of a touch screen. In addition, most HPC/Pro devices have both a Compact Flash slot and a PCMCIA (Type II) slot, either of which can be used to add additional memory or peripheral devices to the HPC/Pro. Of all the CE platforms, the HPC/Pro is the most powerful and the most familiar-looking device. Figure 1.10 shows an example of an HPC/Pro, the Hewlett-Packard Jornada.

FIGURE 1.10:

The Hewlett-Packard
Jornada



Copyright Hewlett-Packard © 1999

The form and design of the HPC/Pro devices make them an ideal platform for doing just about anything short of compiling code. As noted earlier, the keyboard is easy to type with, and the battery on an HPC/Pro device lasts up to 10 hours.

There are several features that make HPC/Pros unique from other CE devices:

- Display
- VGA-out port
- A laptop-size keyboard
- The HTML Viewer control

Display

The displays on HPCs typically measure about 640×480 . That's twice as large as the screens on the HPC devices, and four times the size of the PPC screens. And it's not too far out of line with the resolutions available on Desktop machines.

The final characteristic of HPC/Pro displays that makes them unique is that all of the HPC/Pros have color displays capable of displaying 256 colors.

VGA-Out Port

Some HPC/Pros, such as the Hewlett-Packard Jornada, offer a VGA-out port, so that your application can send data to a standard VGA monitor in a number of resolutions not supported directly by the CE device's own display. While it does take some additional battery power to drive the VGA-out port, this is still an excellent feature for any application that needs to display presentation-oriented data.

As a demonstration of this feature, let's borrow some simple graphics code that paints simple shapes from one of the MSDN samples and modify it to display its output on the VGA-out port.

In order to activate the VGA-out port, we'll set up a simple window with a large button marked "VGA!!!". When the user clicks the button, we'll change the caption of the button (to "End VGA"), activate the VGA-out port by calling `CreateDC()` with a special parameter, and launch a separate thread (which we'll also write) to do the drawing. When the user clicks the button a second time, we'll change its caption back to the original "VGA!!!", set a flag to stop the drawing, and do any necessary cleanup.

Your first task is to create the handler for the button click event by adding a block of code to your `WM_COMMAND` message handler. First, test to make sure that the `WM_COMMAND` message refers to your button:

```
case WM_COMMAND:
    if ((HWND)lParam == hButton)
    {
```

If it does, next check a Boolean flag, `bVGA`, to see if you're already displaying something on the VGA-out port:

```
    if (bVGA)
    {
```

If `bVGA` is set to `TRUE`, that means you're currently using the VGA-out port. If you arrive at this point, it means that the user would like to end the VGA graphics show. Your first job then is to reset the `bVGA` flag so that your graphics thread can terminate:

```
bVGA = FALSE;
```

Then delete the special VGA-out HDC you created with a call to `CreateDC()`, which we'll look at in just a moment. Next, reset the caption of your button so that it now reads "VGA!!!" again:

```
DeleteDC(hVGA);
SetWindowText(hButton, TEXT("VGA!!!"));
```

Then exit the procedure:

```
return TRUE;
}
```

If, however, the `bVGA` flag was not set, you'll know that you need to initialize the VGA-out port and begin drawing to it. In order to open the VGA-out port, call `CreateDC()` but pass as the first parameter the name of the DLL that serves as the VGA-out port driver. On the Hewlett-Packard Jornada, there are three such DLLs to choose from, as illustrated in this table:

Driver DLL Name	VGA display resolution
Skvout0.dll	640 × 480
Skvout1.dll	800 × 600
Skvout2.dll	1024 × 768

TIP

This information can be found at runtime by querying the registry under `HKEY_CLASSES_ROOT\Drivers\Display\Active\`. Any keys under this key represent the different available output devices.

For the purposes of this demonstration, use the 1024 × 768 driver. Your call to `CreateDC()` will then look like this:

```
hVGA = CreateDC(TEXT("skvout2.dll"), NULL, NULL, &Init-
Data);
```

Next, test to make sure your call to `CreateDC()` worked, and, if it didn't, handle the error with a simple message:

```

        if (!hVGA)
        {
            MessageBox(hWnd, TEXT("Unable to initialize VGA
output device."), TEXT("ERROR:"), MB_OK);
            return TRUE;
        }

```

If you were successful, set the `bVGA` flag to true, indicating to other parts of our program both that the VGA-out port is in use and that it's now safe to paint to the `hVGA` display context.

Set the `bVGA` flag to true and start your drawing/graphics thread:

```

        bVGA = TRUE;
        StartVal = 0;
        hVGAThread = CreateThread(NULL, 0, &VGAThreadProc,
&StartVal, 0, &dwVGAThreadID);

```

Then, set the button's text to "End VGA" and exit the procedure:

```

        SetWindowText(hButton, TEXT("End VGA"));
        return TRUE;
    }
    break;

```

Now, take a quick look at your drawing/graphics thread. It's fairly standard code, so it will probably look familiar. The first thing to do in this routine after declaring your variables is to initialize a `RECT` structure to the full dimensions of the VGA resolution you're using:

```

DWORD WINAPI VGAThreadProc(LPVOID lpParameter)
{
    int x1,y1,x2,y2,x3,y3,x4,y4,r,g,b,nObject;
    RECT rect, textrect;
    HBRUSH hBrush;
    LPTSTR DebugMsg;

    rect.top = 0;
    rect.left = 0;
    rect.right = 1024;
    rect.bottom = 768;

```


We'll be using the RECT a little later, but first, paint a black rectangle on the hVGA HDC to clear it out, and start with a clean background:

```
if(hBrush = SelectObject(hVGA,CreateSolidBrush(RGB(0,0,0))))
{
    Rectangle(hVGA, -2, -2, 1026, 770);
    DeleteObject(SelectObject(hVGA,hBrush));
}
```

TIP

Paint the rectangle just a little bit larger than needed to make sure it covers the whole area.

Now, enter a while loop based on the value of bVGA:

```
while (bVGA)
{
```

At this point, begin your drawing code. First, obtain some random color values to draw with by calling the C runtime function, rand(), applying modulus 25 to the result, and then multiplying that value by 10:

```
r = (rand() % 25) * 10;
g = (rand() % 25) * 10;
b = (rand() % 25) * 10;
```

Next, create an HBRUSH to paint with and, if that operation is successful, choose some random coordinates at which to do your drawing:

```
if(hBrush = SelectObject(hVGA,CreateSolidBrush(RGB(r,g,b))))
{
    x1 = rand() % rect.right;
    y1 = rand() % rect.bottom;
    x2 = rand() % rect.right;
    y2 = rand() % rect.bottom;
    x3 = rand() % rect.right;
    y3 = rand() % rect.bottom;
    x4 = rand() % rect.right;
    y4 = rand() % rect.bottom;
```

Next, make a random choice as to which of three possible shapes you should draw, and via a `switch...case` block, draw the randomly chosen shape:

```
nObject = rand() % 3;

switch(nObject)
{
    default:
    case OBJ_RECTANGLE:
        Rectangle(hVGA, x1, y1, x2, y2);
        break;

    case OBJ_ELLIPSE:
        Ellipse(hVGA, x1, y1, x2, y2);
        break;

    case OBJ_ROUNDRECT:
        RoundRect(hVGA, x1, y1, x2, y2, x3, y3);
        break;
}
```

Next, clean up by deleting the `HBRUSH` you created earlier:

```
DeleteObject(SelectObject(hVGA, hBrush));
}
```

And, because you don't want this thread to lock the rest of the program, put it to sleep for one second at a time:

```
Sleep(1000);
}
```

When your `while` loop exits, it means that the user has clicked on the button a second time, and you've now closed the VGA-out port. Exit the thread and the actual function:

```
ExitThread(0x0000);
return TRUE;
}
```

You're done. Although this demonstration only paints ellipses and rectangles, the VGA-out port is a great way to display charts, graphs, or any kind of graphical data. The Jornada even comes with an application that displays the entire CE desktop on the VGA monitor!

A Laptop-Size Keyboard

HPC/Pros' keyboards are very easy to use. This means that you can develop an application for an HPC/Pro device with the expectation that the user can and will be doing a lot of typing. Therefore, it's both possible and reasonable to request some serious data entry from users of an HPC/Pro.

The HTML Viewer Control

Just as HPCs do, the HPC/Pro devices offer the HTML Viewer control. For information on how to use the HTML Viewer control, see the section on HPC devices above.

WARNING

Be aware that if you're developing for both the PPC and the HPC/Pro that the Pro supports the Ink control but not the CapEdit control!

Version Numbering of the CE Operating System

With the release of the HPC/Pro devices, Microsoft announced a new version numbering scheme to help keep track of the different features offered by each version of Windows CE.

The HPC version of CE is known as v2.00 because it was the first second-generation CE operating system. The PPC version of CE is known as v2.01, and the HPC/Pro version is v2.11. It's worth noting that, just as some manufacturers allowed users to upgrade their CE 1.0 devices to CE 2.0 devices, some HPCs can be upgraded to CE v2.11. However, the software upgrade alone does not give them the additional HPC/Pro features, such as longer battery life, etc.

Other Devices

Because Windows CE is also available as an embedded operating system, and because each OEM can customize the operating system, there are any number of devices that offer some unique configuration or sets of features that might be just what you're looking for in your CE-based solution.

Casio's Vertical Markets division puts out one such device, known as the PA-2400. Casio has taken the larger screen of the HPCs and combined it with the software-based keyboards of the PPC devices to create a Handheld PC that doesn't have a keyboard. The result, shown in Figure 1.11, is a tablet-like device, adequate for displaying moderate amounts of data and ideal for bar code/inventory-type applications. From a developer's standpoint, the PA-2400 is just like any other HPC with its 480×240 screen, grayscale, and all of the standard controls available on an HPC.

Of course, the PA-2400 is only one example of the variety of Windows CE devices that exist. By the time you read this, there will also be an Auto-PC device. To accommodate this variety, developers must write code as generically as possible, so that it will run on as many CE platforms as possible with as little modification as possible. Another option is to target a specific device from a specific manufacturer as part of your CE solution.

FIGURE 1.11:

The Casio PA-2400 "tablet" device



Copyright Casio Business Solutions Group © 1999

Accessories for CE

In this section, we'll be looking at Ethernet/networking solutions and a bar code solution, some of the accessories that you can add to a CE device to make your CE solution that much more complete.

Ethernet/Networking Solutions

Adding a network connection to a CE device is one of the most important things you can do to enhance the functionality of a CE solution. Once a CE device is on the network, it can browse files, access data, and (in the case of the HPC/Pro) even print to a network printer.

Furthermore, from a developer's point of view, a network connection can be your best friend thanks to the Ethernet debugging features of VC++. As the name suggests, Ethernet debugging lets you execute and debug your application over a network. Debugging your applications this way is considerably faster and easier than the standard serial connection.

When it comes to networking and CE, there are two solutions to consider:

- A traditional wired connection
- A wireless connection

The tradeoff between a wired connection and a wireless connection can be summarized in one sentence: Can your device be tethered to a network hub, or do you require absolute freedom to go anywhere with your device?

Of course, that question itself is really answered by the application. If you are developing an application that's going to be used, say, all over a warehouse, then you'll definitely need a wireless connection. On the other hand, if your application is more oriented toward a sales rep in the field who only needs to get onto the corporate network once a day, then a wired connection would be the way to go.

Wired Connections

Socket Communications of Newark, California (<http://www.socketcom.com>) is the leader in the field of wired network accessories for CE. They offer two different kinds of Ethernet cards for Windows CE:

- PCMCIA-based “Low Power” Ethernet card
- Compact Flash (CF) Ethernet card

PCMCIA-Based Low-Power Ethernet Card This version of Socket Communications’ Ethernet card is ideal for HPCs and HPC/Pros that have a PCMCIA slot available. The biggest feature of this card is that it uses extremely low power, which, of course, is good for the battery life of the user’s device. The standard version of the card supports only 10BaseT Ethernet, but the “plus” version supports both 10BaseT and 10Base2 Ethernet.

Compact Flash Ethernet Card The fact that there’s a Compact Flash version of this Ethernet card available is no small feat! What this means is that you don’t have to use your device’s PCMCIA slot to get on your network. Instead, you can get the Compact Flash Ethernet card to use the Compact Flash slot, which you’re probably not using anyway! Furthermore, the CF slot is the only way to get a PPC device on the network, as they do not have PCMCIA slots.

A Wireless Connection

With a wireless connection, your CE device can be anywhere—literally—and still have access to every resource on your network. Proxim, Inc. of Mountain View, California (<http://www.proxim.com>) is the leader in the field of wireless Local Area Network solutions for CE. Proxim offers a wireless bridge (a kind of network hub) that has a radius of 500 feet indoors and well over 1000 feet outdoors.

The way the Proxim RangeLAN2 system works is that you have one or more wireless bridges (hubs) which hook up to your existing network. Then, for each CE device that needs to be on the network, you have a PCMCIA-based Ethernet card.

One of the advantages of the Proxim system is that their PCMCIA cards are also designed to use smart power management. For instance, if there is no network traffic after a certain amount of time—and you can set this value yourself—the card will enter a sleep mode, merely signaling the minimum amount to remain connected to the network. This of course, uses less power and improves battery life.

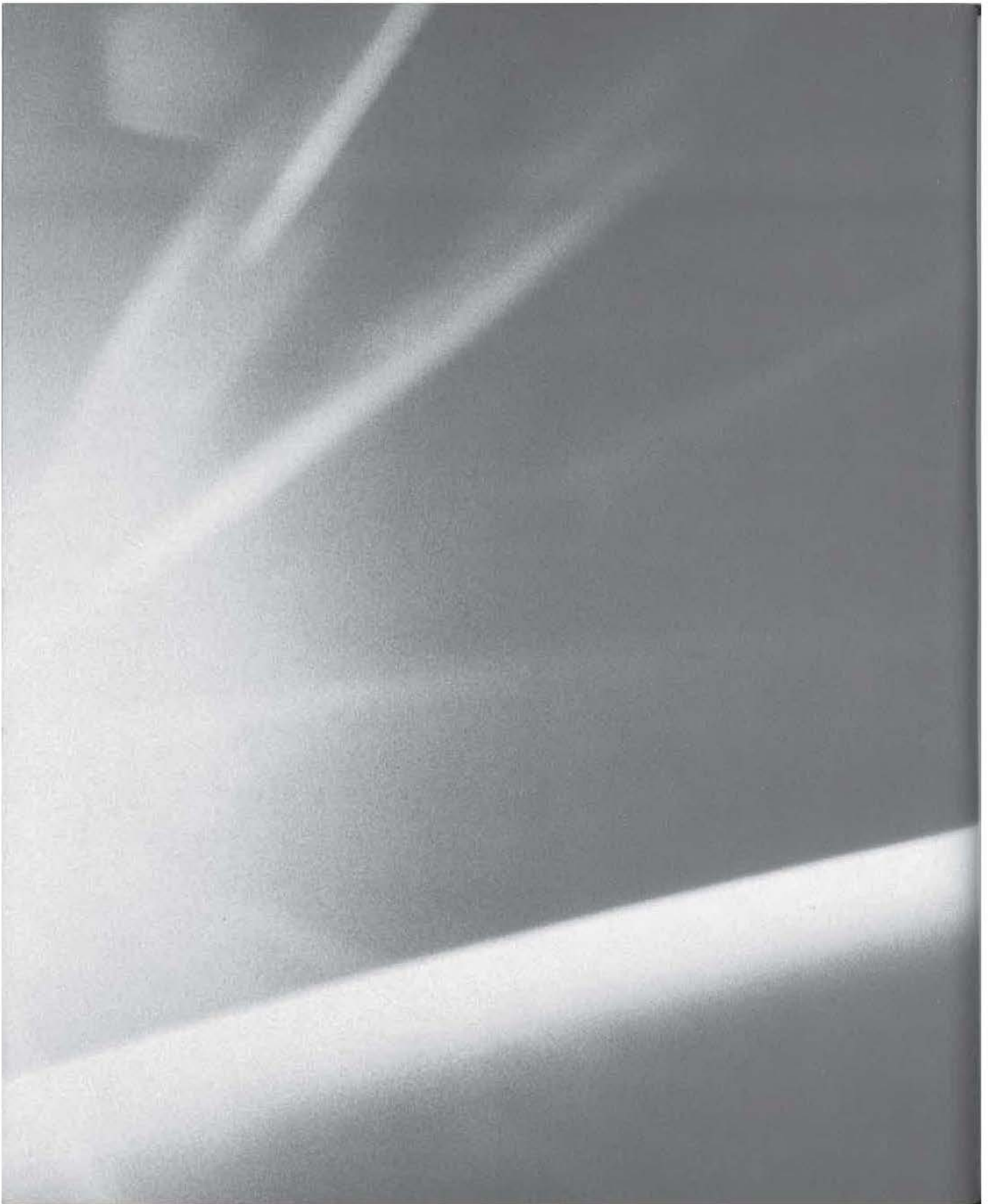
Bar Code Readers

Bar code readers and CE seem to go hand in hand. Many developers are already implementing bar code-based solutions with Windows CE. As far as bar code readers and CE are concerned, Socket Communications is the leader here, too. They have a variety of bar code readers that come in all shapes and sizes, depending on the nature of the application. And, just as with their Ethernet cards, Socket Communications' bar code readers are available in both PCMCIA form as well as CF form.

The great thing about Socket's CE solutions is that when a user scans in a bar code, their drivers make it appear to the application that the user has typed in the information using the keyboard. This means that integrating one of their bar code readers into your CE solution requires no extra coding on your part!

Summary

This chapter gave a general overview of some of the more important CE topics. In the next several chapters and throughout the book, we'll be exploring each of these topics in depth, finding solutions to problems and using whatever new features we find along the way.



CHAPTER

TWO

2

The Three Commandments of Writing for CE

- The First Commandment: Your Application Must Use the Unicode Character Set
- The Second Commandment: Your Application Must Be Low-Memory Aware
- The Third Commandment: Know Your Form Factor!

The purpose of this chapter is to provide you with a general “heads up” on the unique requirements and considerations you should be aware of when writing for Windows CE. Not all of the items in this chapter will apply specifically to you and your programs, but most of them will. As you already know, writing for CE is definitely not the same as writing for Win32. There are a number of new challenges, and it is those challenges that we’ll focus on now.

The Three Commandments that you must follow when programming for CE are as follows:

1. Your application must use the Unicode character set.
2. Your application must be low-memory aware.
3. Know your form factor!

Because these are the three rules that will affect almost every single CE program you will write, we will take a serious, detailed look at each one.

The First Commandment: Your Application Must Use the Unicode Character Set

The First Commandment is that your application must use the Unicode Character Set. Unicode is a worldwide character standard, which theoretically makes it much easier to internationalize your applications.

Windows CE is a Unicode-based operating system. That means that all of the text displayed to the user by the operating system is actually Unicode text. It also means that all of the Edit boxes, List boxes, Combo boxes, etc., display Unicode character string only. From the practical standpoint of working with text strings under Windows CE, however, Unicode is just something that makes your life more difficult.

Unicode will create the following changes in the way your programs work with text:

1. All strings must be declared using Unicode types rather than char types.

2. All text literals in your programs must be Unicode strings.
3. You must choose the correct C runtime library (RTL) functions for Unicode strings.
4. Your program must now handle two types of text files.

Unicode forces you to make these changes because it is a two-byte (16-bit) character set. That means that a single Unicode character is actually two bytes wide—twice as wide as each character in the ANSI character set of Windows 98.

TIP

The Unicode strings are often called *wide* strings, and Unicode characters are *wide* characters. You might see Windows NT source code relying on a variety of functions with the letter *W* prominently in the function names or type declarations. The *W* is what indicates a wide or Unicode string. For example, `LPWSTR` is a long pointer to a wide string.

Even if you decide that your program(s) will only read and write ANSI-based data files, you'll have to deal with the fact that all of the Windows CE API and standard Windows controls work with only Unicode text. This means that you'll still have to convert that data to Unicode to display it, and then you'll have to convert it from Unicode to read it!

In other words, there's no way to completely avoid Unicode. Your program will be either 100 percent Unicode-based or some mix of Unicode- and ANSI-based.

Let's take a look at each of the five Unicode-related changes and how, exactly, they will affect your code.

Declaring Strings Using Unicode Types Rather Than Char Types

Previously, under Windows 98, you probably declared your strings using `char`, `char *`, or `LPSTR`, as in either of the following examples:

```
char * lpszMessage;  
LPSTR lpszMessage;
```

Under CE, however, neither of these string types will display properly, if at all. That's because both `char *` and `LPSTR` are ANSI, 8-bit character strings. Therefore,

to get your strings to behave correctly under Windows CE, you must declare them using either

- Unicode-specific string types
- or
- *Generic* string types

Unicode-Specific String Types

Unicode-specific string types are just that—strings in which each character is explicitly set to be two bytes wide. They are 100 percent compatible with all of the Unicode-based CE API and RTL functions. Table 2.1 shows the Unicode-specific type to ANSI-specific type mappings.

TABLE 2.1: Unicode-Specific to ANSI-Specific Type Mappings

Type	Description	What It Replaces
WCHAR	2-byte Unicode character	char
WCHAR*	pointer to a string of Unicode characters	char*
LPWSTR	long pointer to a Unicode string	LPSTR
LPCWSTR	long pointer to a Unicode string constant	LPCSTR

NOTE

The code in this book employs both the Unicode-specific string types and the generic string types.

So, when using Unicode-specific strings, your declarations will look like either of the following examples:

```
WCHAR * lpwzMessage;
```

```
LPWSTR lpwzMessage;
```

Generic String Types

Generic string types are macros that are mapped at compile time to the correct character set (ANSI or Unicode), depending on the target operating system. In other words, if you are targeting Windows 98—an ANSI-based OS—TCHAR is

defined as a 1-byte char. If you are targeting Windows CE, TCHAR is defined as a WCHAR or 2-byte Unicode character. Therefore, if you use generic string types to declare your strings and characters, your program will be Unicode-aware as soon as you compile. Table 2.2 shows the generic string type to ANSI-specific string-type mappings.

TIP

Because generic string types change their definitions based on the target platform, you can write string-handling code that will compile for a variety of platforms.

TABLE 2.2: Generic String Type to ANSI-Specific String Type Mappings

Type	Description	What It Replaces
TCHAR	generic character type	char
TCHAR*	pointer to generic character string	char*
LPTSTR	long pointer to generic string	LPSTR
LPCWSTR	long pointer to generic string constant	LPCSTR

If you're using these generic strings type, your declarations will look like either of the following examples:

```
TCHAR * lpszMessage;
```

```
LPTSTR lpszMessage;
```

Of the two methods for declaring strings under CE, the generic method is usually preferred because of the cross-compilation advantage. However, there are times when you may want to use the explicit Unicode types. For instance, you may want to distinguish between functions that require a Unicode string and functions that will require an ANSI string type.

NOTE

The Unicode-specific string types are defined in WCHAR.H. The generic string types are defined in TCHAR.H.

Using Unicode Strings for All Text Literals

Text literals are any hard-coded character strings that exist in your program. They can be strings that are explicitly declared, or they can be embedded in some other command and never declared at all, as in the following respective examples:

```
char lpszMessage[] = "Good Morning!";  
MessageBox(hwnd, (LPSTR)"Cats and mice do not have fun.", (LPSTR)"Oh  
no!", MB_OK);
```

When you compile the above lines for CE, though, you will get warnings, and your program probably won't run at all.

There's a simple solution to this problem, but it can be time-consuming, especially if you're porting code that contains a fair number of text literals.

To mark your text literals as Unicode, you must enclose them in the TEXT macro, as in the following examples:

```
TCHAR lpszMessage[] = TEXT("Good Morning!");  
MessageBox(hwnd, TEXT("Cats and mice do not have fun."), TEXT("Oh  
no!"), MB_OK);
```

The TEXT macro takes care of converting your text literals at compile time—and that's all you need to do to ensure that your text literals will show up properly under CE.

Alternatives to the Text Macro

There are two additional macros that do exactly the same thing as the TEXT macro.

The L macro, which looks like this:

```
TCHAR lpszMessage[] = L"Good Morning!";
```

The only difference between the L macro and the TEXT macro is the lack of parentheses.

The _T macro, which looks like this:

```
TCHAR lpszMessage[] = _T("Good Morning!");
```

The only difference here is that using the _T macro instead of the TEXT macro saves you from typing two characters.

The TEXT, L, and _T macros are all functionally equivalent and choosing between them is strictly a matter of preference. There is no penalty for choosing one over the other.

Choosing the Correct RTL Functions for Unicode Strings

One of the consequences of changing all of your strings over to Unicode-aware string types is that some of the RTL (runtime library) functions are ANSI-based and will not work with Unicode strings. Of course, there is a set of Unicode-based functions that replaces the ANSI-based one, and there are also some generic string-type routines that you can use. The trick is finding and choosing the right ones.

The generic string type routines will work just fine with Unicode-specific strings, and—when you compile for CE, at least—the Unicode-specific functions will work with the generic string types.

As a rule, though, you don't want to mix and match the string types and their related functions. Mixing generic string types with Unicode-specific functions defeats the cross-platform purpose of generic string types and means that your code will be much more difficult to port or reuse later. Furthermore, mixing string types and functions can cause your code to be more difficult to read and debug.

Table 2.3 lists some of the more common ANSI-based RTL functions and their Unicode and generic string equivalents.

TABLE 2.3: ANSI, Unicode, and Generic String Function Equivalents

ANSI Function	Unicode Function	Generic String Function	Description
<code>strlen()</code>	<code>wcslen()</code>	<code>_tcslen()</code>	Returns number of bytes in a string.
<code>atoi()</code>	<code>_wtoi()</code>	<code>_ttoi()</code>	Converts a string to an integer.
<code>strcmp()</code>	<code>_wcsicmp()</code>	<code>_tcscmp()</code>	Compares two strings.
<code>atof()</code>	<code>wcstod()</code>	<code>_tctod()</code>	Converts a string to a floating point value. (There is no direct mapping from <code>atof()</code> to a Unicode or generic string function. Instead, you must use the string to double-precision functions.)

TIP

It's still possible to increment Unicode string pointers as if they were ANSI string pointers, like this: `*str++`;

Equipping Your Program to Handle Two Types of Text Files

Unicode isn't universal—just because it might be easier for CE programs to store their data using Unicode doesn't mean that other operating systems can read Unicode, nor will they necessarily be providing data in Unicode form. If your program is going to be receiving data in the form of text files, it's a good idea to ensure that you can handle both Unicode data and ANSI data.

The good news is that it's not very difficult to do this. Mostly, you just have to be able to determine whether a given file is Unicode or ANSI. Although ANSI text files do not have any kind of identifying header or signature, Unicode files do. The Unicode signature consists of two bytes (one Unicode character) that appear as the very first data in the file: in hex, FEFF. To determine whether or not a file you're working with is Unicode, simply open it up and read the first two bytes. If those bytes are FEFF, you know the file is a Unicode file. Otherwise, you can safely assume that it's an ANSI file.

WARNING

The FEFF signature is a generally agreed upon standard for working with Unicode files on a PC. It's entirely possible that the data you receive from some other source may completely ignore this standard. Also, the order of the two bytes may depend on the "endian-ness" of the machine producing the file. In other words, in most cases, you can count on the FEFF signature, but it is not always 100 percent guaranteed to be there.

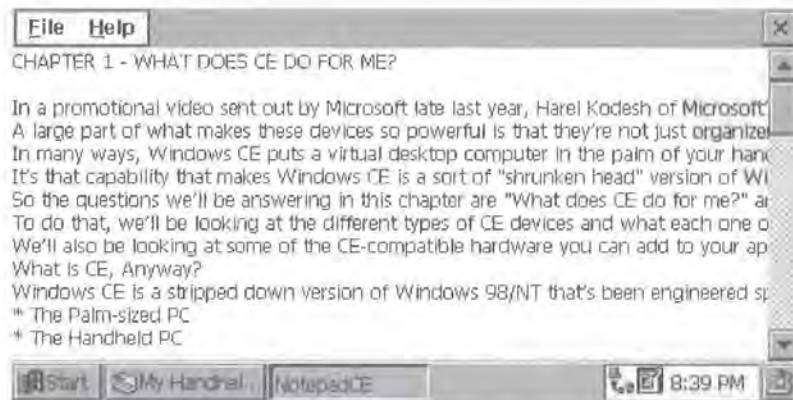
Converting between Unicode and ANSI Character Sets— A Sample Application

One example of an application that would need to properly accommodate files in both ANSI and Unicode formats is a notepadlike application. (After all, NT's notepad handles both Unicode and ANSI files transparently.)

For a demonstration of the techniques and functions required to convert smoothly between one character set and the other, let's build a CE version of the familiar notepad text editor. The finished CE-based notepad is shown in the graphic below.

FIGURE 2.1:

The finished CE notepad application



Under Windows 98/NT, creating a notepad application is so simple, it's usually only used as a teaching exercise. However, writing a notepad application for CE requires you to support the two character sets, which makes the project a bit more complicated. There are two tricks that you'll have to employ in order to pull this off:

1. Account for the user opening Unicode- and ANSI-based text files.
2. Save the user's text files in both character sets for maximum compatibility.

Opening Both Unicode- and ANSI-Based Text Files To be able to open both Unicode- and ANSI-based text files, you must be able to auto-detect the character set of the text file you're opening and then read the data correctly. As you know, ANSI text files do not have any kind of header or signature bytes that uniquely identify them as ANSI text files. However, Unicode text files do. The Unicode signature bytes can be defined in C code as

```
#define UNICODE_HEADER 0xFEFF
```

Header bytes enable you to open the file, read the first two bytes, and compare those bytes against the UNICODE_HEADER value.

To check whether a file is Unicode-based, open it and read the first two bytes:

```
if(ReadFile(*fp, buf, 2, &num, NULL))
{
```

Then, so that the calling routine can still treat the file as though it's just been opened, set the file pointer back to the beginning of the file:

```
fseek(fp, 0, SEEK_SET);
```

Next, make the actual comparison of the two bytes you read against the definition of a Unicode header and return the appropriate value:

```
if( (buf[0] == UNICODE_HEADER) )
{
    //it's Unicode!
    return(TRUE);
}
else
{
    // it's ANSI!
    return(FALSE);
}
}
```

Finally, you need to handle the condition of not being able to read from the file in the first place:

```
else
{
    return(-1); // indicates error
}
```

When your code is assembled into final form, it will look something like the following:

```
int fisunicode(FILE *fp)
{
    unsigned short buf[4];
    DWORD num = 0;

    //initialize buffer
    buf[0] = 0x0000;
    buf[1] = 0x0000;
    buf[2] = 0x0000;

    if(ReadFile(*fp, buf, 2, &num, NULL))
    {
        fseek(fp, 0, SEEK_SET);
        if( (buf[0] == UNICODE_HEADER) )
```

```
    {  
        //it's Unicode!  
        return(TRUE);  
    }  
    else  
    {  
        // it's ANSI!  
        return(FALSE);  
    }  
}  
else  
{  
    return(-1); // indicates error  
}  
}
```

Using this code, your applications can now branch on the file type (Unicode or ANSI) and handle the reading of the file in whatever manner is appropriate. In the case of your notepad application, of course, the appropriate action is to read the text from the file and put it into an edit control.

Saving Unicode- and ANSI-Based Text Files Merely opening both types of files would be pointless if you couldn't save in both formats, however. In order to account for both text formats, when saving data, you can simply provide an extra Save As menu for the ANSI format, as shown in Figure 2.2.

FIGURE 2.2:
The Save As ANSI menu



Of course, ANSI is an arbitrary choice here, and it could just as easily have been a Save As Unicode menu. However, since it's easier to save a file as a Unicode-based document, it's a good idea to make that the default.

Once you've chosen one character set as the default, implementing the different saving functions becomes quite simple. Before we can take a look at the Unicode-based file-saving code, though, you need a function that can write the Unicode signature bytes out to the file. `WriteUnicodeTag()` does just that:

```
void WriteUnicodeTag(FILE *fp)
{
    unsigned short buf[4];
    DWORD num = 0;

    buf[0] = UNICODE_HEADER;
    buf[1] = 0x0000;
    buf[2] = 0x0000;

    //now write the bytes to the file
    WriteFile(*fp, buf, 2, &num, NULL);
}
```

NOTE

In these examples, we're using `stdio.h` file access functions. For more information on using the `stdio.h` functions on Windows CE, see Chapter 3.

Now you can proceed with the main task of writing the contents of the edit control out to a file. First, open the file for writing and write the Unicode header bytes. The next step is to determine the number of characters you'll be writing to the file, dynamically allocate memory for a string (`szChunk`), and store the contents of the edit control into that string. Notice that, because `GetWindowTextLength()` reports the number of characters and not the number of bytes, you must allocate twice the result of `GetWindowText()`.

```
fp = fopen(ofn.lpszFile, TEXT("w"));
if (fp != NULL)
{
    WriteUnicodeTag(fp);
```

The next thing you need to do is determine the number of characters you'll be writing to the file, dynamically allocate memory for a string (`szChunk`), and then store the contents of the edit control into that string. Note that because

`GetWindowTextLength()` reports the number of characters—not the number of bytes—you have to allocate the result of `GetWindowText()` twice:

```
filesize = GetWindowTextLength(hEdt) + 1;
szChunk = LocalAlloc(LMEM_ZEROINIT, filesize * sizeof(WCHAR) +
                    sizeof(WCHAR));
GetWindowText(hEdt, szChunk, filesize);
```

Next, write that string out to the file, making sure to remove the trailing NULL (which, in the case of Unicode-based strings, is actually two NULL characters):

```
if (fwrite(szChunk, sizeof(byte), filesize * sizeof(WCHAR) -
          sizeof(WCHAR), fp) == 0)
    break; //didn't write anything, so nothing to do
```

Finally, free the memory you allocated and close the file:

```
LocalFree(szChunk);
fclose(fp);
}
```

To save text in ANSI format, you follow roughly the same procedure, with the exception that the text of the edit control—which is in the Unicode character set—must be converted to ANSI before it can be written out to the file. This requires that you create a routine called `UnicodeToANSI()`, which, as you may have guessed, will take a Unicode-based string and return an ANSI-based string. The work of the conversion itself is done through the API function `WideCharToMultiByte()`, which takes eight parameters that specify a number of conversion options. For our purposes here, we'll use the simplest settings and specify an ANSI conversion with no options. Your `UnicodeToANSI()` function, then, looks like this:

```
LPSTR UnicodeToANSI(LPWSTR str)
{
    LPSTR buf;
    int len = 0;
    // return string length - add one for NULL and halve it.
    len = (wcslen(str) + 1) / 2;
    buf = (LPSTR) LocalAlloc(LMEM_ZEROINIT, len);
    // caller's responsibility to free this!
    //CP_ACP == ANSI code page conversion
    WideCharToMultiByte(CP_ACP, 0, str, wcslen(str), buf, len, NULL,
                        NULL);
    return (buf);
}
```

Now that we've got that out of the way, let's take a look at how the overall ANSI-based file-saving code works. Just as with the Unicode-based file-saving routine, the first thing to do is open the file for writing:

```
fp = fopen(ofn.lpszFile, TEXT("w"));
if (fp != NULL)
{
```

Next, as with the Unicode-based file-saving routine, you need to allocate memory for a string (`szChunk`) and store the edit control's text in that string:

```
    filesize = GetWindowTextLength(hEdt) + 1;
    szChunk = LocalAlloc(LMEM_ZEROINIT,
        filesize * sizeof(WCHAR) +
        sizeof(WCHAR));
    GetWindowText(hEdt, szChunk, filesize);
```

Normally, you'd just save the text to the file at this point, but because you're writing out the ANSI-based version of your text, first call your conversion function, storing the resulting ANSI-based string into a variable (`szChunkA`), and then write the data to the file:

```
    szChunkA = UnicodetoANSI(szChunk);
    if (fwrite(szChunkA, sizeof(byte),
        filesize-1, fp) == 0)
        break;
    //didn't read anything, so nothing to do
```

Then, as you did before, free the memory you allocated for the strings and close the file:

```
        LocalFree(szChunkA);
        LocalFree(szChunk);
        fclose(fp);
    }
```

When assembled, the full routine looks like this:

```
fp = fopen(ofn.lpszFile, TEXT("w"));
LocalFree(szChunkA);
if (fp != NULL)
{
    filesize = GetWindowTextLength(hEdt) + 1;
    filesize = filesize;
    szChunk = LocalAlloc(LMEM_ZEROINIT,
        filesize);
```

```
GetWindowText(hEdt, szChunk, filesize);
szChunkA = UnicodetoANSI(szChunk);
if (fwrite(szChunkA, sizeof(byte),
          filesize-1, fp) == 0)
    break;
//didn't write anything, so nothing to do
LocalFree(szChunkA);
LocalFree(szChunk);
fclose(fp);
}
```

You have now successfully saved the Unicode data in the ANSI character set. Typically, a few helper functions, such as `ffileisunicode()`, `WriteUnicodeTag()`, and `UnicodetoANSI()` are all that you'll need in order to get your applications to work with both Unicode- and ANSI-based text files.

NOTE

The code for the full CE notepad application can be found on the CD-ROM in the subdirectory for this chapter.

The Second Commandment: Your Application Must Be Low-Memory Aware

As you know, CE devices typically have nowhere near as much memory as a desktop PC. The Second Commandment, therefore, involves tailoring your program for the low-memory environment of a CD device.

In addition to having less memory than a desktop PC, the CE user can adjust both how much of the physical memory is to be used as storage (i.e., simulated hard-disk space) and how much is to be used as program memory (i.e., RAM). As if that isn't enough, the operating system itself can also request that your program free some of its memory whenever the amount of available memory gets below a certain level.

In order to correctly navigate this maze of memory-related requirements, you must make sure your program is doing the following:

1. Keeping the size and number of static variables to a minimum
2. Keeping the EXE file size low

3. Checking the return result of memory allocation
4. Mass-allocating your application's memory
5. Handling the WM_HIBERNATE message

Keeping the Size and Number of Static Variables to a Minimum

When you're coding for Windows CE, everything related to memory allocation becomes an issue. Even something like local variables and stack space could be a problem down the road if it's not handled properly from the beginning.

While it's unlikely that you'll actually run out of stack space, it is possible. Windows CE allocates approximately 60K per stack per thread, so you do have quite a bit of room. However, all it takes is some careless static allocation of large strings to quickly push that limit, as in the following code:

```
char LargeString[1024];
```

WARNING

Recursive functions also require special attention, as those that may work fine on a desktop machine might "blow the stack" on a CE device.

Even without the issue of using up the stack space, the general rule is that you don't want to allocate memory you're not using. As Rule 5 in the list above suggests, this could cause trouble for other applications that are running but are not active when your program attempts to grab its memory because, if there's not enough memory for your program, CE will ask the other applications to free up some memory.

Keeping the EXE File Size Low

Keeping the EXE file sizes low is a big issue because programs on CE are executed just as they are in Windows 98/NT; namely, the executable is loaded into RAM and then executed. The very act of launching an executable is, in effect, a memory allocation.

Under Windows 98/NT, you can pretty much use as much disk space as you need, and it's the user's responsibility to ensure that they have enough space to install your application. Under Windows CE, however, there is a fixed amount of

memory available. Most users won't be able or willing to upgrade that memory just for one program. And, even if they have the memory to support such a program, they probably won't appreciate the fact that it will keep them from installing other programs down the line. Therefore, you want to ensure that your executable is as small and as tight as it can be.

This involves taking a close look at the resource file that's compiled into your EXE. Have you converted the bitmaps to a lower-quality (i.e., smaller-memory) format? Windows CE supports a number of lower-quality, space-saving bitmap types that can greatly reduce the size of an executable.

If you ported the application from a desktop program, are there resources you're not using in the CE version, such as WAV files or other such resources? Although the WAV file format supported by CE is a low-quality, compressed form, these files can still take up quite a bit of storage space on the device.

TIP

For more information on converting resources, see Chapter 9.

Checking the Return Result of Memory Allocation

This rule is common sense that applies to all types of programming, not just for CE. But because it matters for CE, too, we'll mention it briefly.

As you know, just because you request memory doesn't mean the operating system will be able to allocate it. Here's a simple block of code that demonstrates how easy it is to handle an allocation error:

```
lpStr = LocalAlloc(LMEM_FIXED, 50 * sizeof(TCHAR));
if ( lpStr )
{
    //...
}
else
{
    //handle error here
}
```

Using this simple code can save you a lot of trouble later on.

Mass-Allocating Your Application's Memory

Typically, in a Windows application, memory is allocated as needed. The flow of a program, then, might look something like this:

```
//beginning of program
var1 = LocalAlloc(LMEM_ZEROINIT, 124);
//some additional operations
//...
var2 = LocalAlloc(LMEM_ZEROINIT, 512);
//...
LocalFree(var1);
var3 = LocalAlloc(LMEM_ZEROINIT, sizeof(SOMESTRUCT));
//etc.
//end of program
```

And so on. The problem with this style of program design is that memory is not always going to be available on CE. An application could be in the middle of a complex operation, only to fail at a critical moment because memory could not be allocated. That would mean that all of the work done up to that point would likely be lost, and the user would have to close some additional applications just to retry the operation.

Casio's Vertical Markets division has developed one technique designed to deal with this. This technique is to allocate all (or nearly all) of the memory needed by the application at once. While this might seem a bit unorthodox, it actually solves this problem in a very creative manner. If you were to redesign the above pseudocode using this technique, it would look something like this:

```
//beginning of program
var1 = LocalAlloc(LMEM_ZEROINIT, 124);
var2 = LocalAlloc(LMEM_ZEROINIT, 512);
var3 = LocalAlloc(LMEM_ZEROINIT, sizeof(SOMESTRUCT));
//now do your work
//...
LocalFree(var1);
//etc.
//end of program
```

Another way to accomplish this is to create a private heap as soon as your application starts, as in the following example:

```
//beginning of program
hHeap = HeapCreate(0, 1024, 1024);
```

```
//...
//anytime later, allocate "off the heap"
var1 = HeapAlloc(hHeap, 0, 124);
var2 = HeapAlloc(hHeap, 0, 512);
var2 = HeapAlloc(hHeap, 0, sizeof(SOMESTRUCT));
//now to your work
//...
HeapFree(hHeap, 0, var1);
//etc.
//...
//just before end of program
HeapDestroy(hHeap);
//end of program
```

The advantage of this kind of coding is twofold:

- If a low-memory condition exists when your program starts up, you'll be able to detect it right away. You can then alert the user as to this condition and take the appropriate action.
- You avoid the possible scenario described above, in that your program will not crash in the middle of an operation due to lack of available memory. This, of course, is the main reason to organize your memory allocation in this manner.

It won't always be possible to organize your memory allocation this way, and there are times when you may find it more efficient to allocate as needed, but whenever possible, this is a technique that's worth using.

Different Types of Memory Allocation

In the examples above, you used two different types of memory allocation.

`LocalAlloc()` is the default memory allocation function under CE, much as `malloc()` is in the C runtime library. It gets the memory it allocates from the process heap, which is a general-purpose space for your program's use. The problem with `LocalAlloc()` is that when you free memory reserved with `LocalAlloc()` (by calling `LocalFree()`), the memory is not freed right away. Instead, it is freed after some time, in an almost lazy manner.

Continued on next page

`HeapAlloc()` works somewhat differently, however. With `HeapAlloc()`, it is possible to reserve memory from either the process heap or a private heap, that you can create yourself by calling `CreateHeap()`. The advantage to using the `HeapAlloc()`-related functions is that `HeapFree()`, the de-allocation function, frees memory immediately.

`VirtualAlloc()` is a third type of memory allocation that involves reserving one page (either 1024 or 4096 bytes, as set by the device manufacturer) at a time.

For most uses, `LocalAlloc()` will do an adequate job. In cases where you're concerned about memory being freed immediately, `HeapAlloc()` is the way to go. And, if you need to grab really large blocks of memory at once, `VirtualAlloc()` is something to look into. Whichever function you choose, you'll still want to follow the advice above and allocate as much memory as possible when your program starts up.

Handling the *WM_HIBERNATE* Message

Under Windows NT, when the system is nearing the upper limit of its available memory, a dialog box pops up warning the user that the “system is running low on virtual memory.” It's then up to the user to start closing some applications so that memory can be freed and the remaining programs can go about their business.

Similarly, under Windows CE, when memory gets tight, the operating system takes action. Instead of notifying the user right away, however, CE asks inactive programs to release some of the memory they're using. It does this by sending a special message called `WM_HIBERNATE` to these programs. This message is unique to Windows CE.

The `WM_HIBERNATE` message is sent to all visible application windows, starting with the window that has been inactive for the longest time. When an application receives the `WM_HIBERNATE` message, it must free up as much memory as it can and, at the same time, save the state of any unfinished work. Freeing memory might include destroying unused window handles, closing noncritical dialog boxes, and so on. For some applications, merely exiting is appropriate.

When the application is reactivated, it can restore the unfinished work to its previous state, and the user will never know the difference. For example, an application might save the text from any of its edit boxes and then destroy those edit boxes to free some memory. Then, when the application is reactivated, it will re-create the edit boxes and populate them with the saved text.

The most important thing to remember about the WM_HIBERNATE message is that your application *must* respond to it in some way. Even simply closing down and letting the user restart the application later is better than no response at all.

Reactivating Your Application

There is no unique-to-CE message that parallels WM_HIBERNATE to signal your application that it has been reactivated. Instead, CE uses WM_ACTIVATE. The problem with this is that your application will receive WM_ACTIVATE messages in the normal course of business, regardless of whether or not it's currently in a hibernating state. Therefore, you can't expect to restore from hibernation every time you get a WM_ACTIVATE message.

The solution to this is to set a global variable to indicate that the application has gone into hibernation whenever you receive a WM_HIBERNATE message:

```
case WM_HIBERNATE:
    //set hibernation flag
    bHibernating = TRUE;
    //finish with hibernation processing
    //...
case WM_ACTIVATE:
    if (bHibernating)
        //restore from hibernation
    //...
```

This way, you can be assured of proper handling for both the hibernation and the re-awakening actions.

The Third Commandment: Know Your Form Factor!

Form factor is a term used to describe the shape, look, and feel of a given device. The Palm-size PC, for example, is one form factor; the Handheld PC is another form factor.

The tricky part is that each form factor has different design requirements. For instance, the screen of a Palm-size PC is long and narrow, about 240×320 . The screen of a Handheld PC, however, is closer to a desktop size, about 640×320 . Similarly, programs running on Palm-size devices may want to resize themselves whenever the user pops up the input panel (software-based virtual keyboard).

The difficulty with all of these different form factors and their unique display types is that tailoring your code to each of these machines can create two problems:

1. The UI of the application should be tailored to the device.
2. Maintaining a single codebase is nearly impossible.

Each of these presents a different set of challenges, which we'll now examine.

The UI of the Application Should Be Tailored to the Device

You will probably find that you can compile your applications for both the HPC and PPC platforms without changing a line of code. However, it's not so likely that your application will *look good* on both platforms.

If you design your application first for the HPC, your dialogs will almost certainly be too big for the PPC, assuming they'll even fit into the screen at all. Similarly, if you design your application first for the PPC, the dialogs will look cramped and small on an HPC. Here again, you probably won't even be able to see the bottom one-eighth or so of the dialog, as it will be hidden by the Taskbar. An example of this is shown in Figure 2.3.

FIGURE 2.3:

A PPC application running on an HPC device



Clearly, even the simplest of programs will require some changes to the UI. The good news is that, in most cases, the changes needed are small and easy to implement.

In the case of the notepad application developed earlier in this chapter, for example, one way you can ensure that the user interface looks good on both device types is to automatically size the window to the screen size. You can do this simply by creating the application's main window with the default size and positioning options set:

```
hwndMain = CreateWindow(szAppName, szTitle,
                        WS_VISIBLE,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL );
```

Windows CE sets the default size to be the full screen area. Then, by creating any child window controls—in the case of the notepad application, the edit control—based on the client area of the main window, you can ensure that the notepad application will look good on both the PPC and HPC device types.

Sometimes, however, just ensuring that windows and controls are the right size isn't enough. The DBView sample that Microsoft provides is a perfect example of this. Figures 2.4 and 2.5 show DBView running on an HPC and PPC device, respectively.

FIGURE 2.4:

DBView running on an HPC device

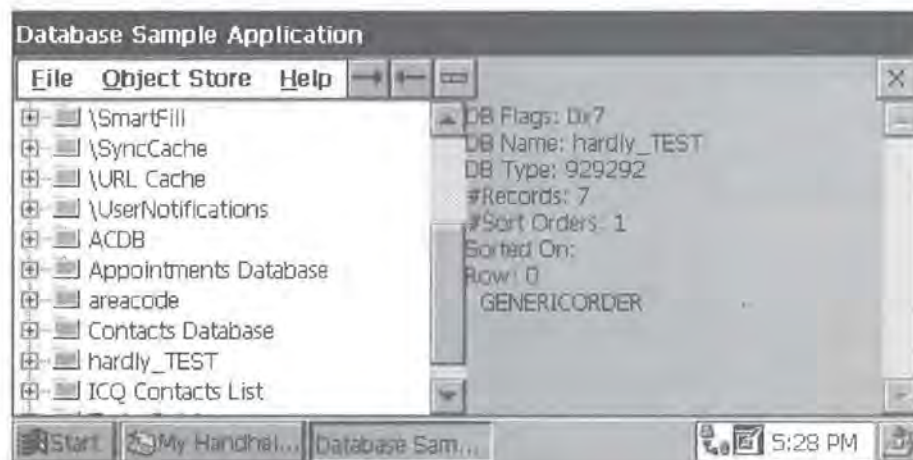
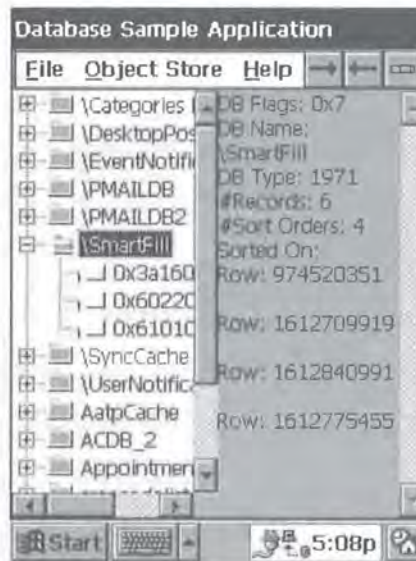


FIGURE 2.5:

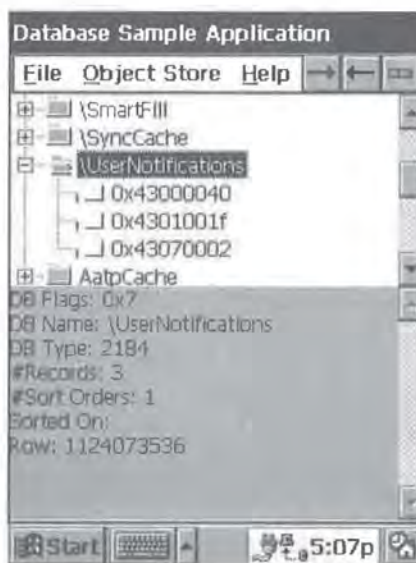
DBView running on a PPC device



As you can see, DBView will compile and run on a PPC or HPC device without any modifications. However, a TreeView control with a width of about 60 pixels, as shown in Figure 2.5, is not exactly user-friendly. Most of the items in the TreeView cannot be read without using the scrollbar. Modifying the UI would clearly help. Figure 2.6 shows a possible modification that will definitely make the program easier to use.

FIGURE 2.6:

DBView modified for the PPC platform



The TreeView and edit controls were originally created so that they'd be alongside each other, as illustrated by this code:

```
GetClientRect(hwnd, &rcClient);
//...
nCommandHeight = CommandBar_Height(hwndCB);
//...
hwndTV = CreateWindowEx(0, WC_TREEVIEW, TEXT("DB View Control"),
    WS_VISIBLE | WS_CHILD | TVS_HASLINES |
    TVS_LINESATROOT | TVS_HASBUTTONS, 0, nCommandHeight,
    rcClient.right/2, rcClient.bottom-nCommandHeight, hwnd,
    (HMENU) IDC_TREEVIEW, g_hInstance, NULL);

hEdit = CreateWindow(TEXT("EDIT"), TEXT("Property Information"),
    WS_VISIBLE | WS_CHILD | ES_MULTILINE | ES_READONLY |
    WS_VSCROLL,
    rcClient.right/2, nCommandHeight,
    rcClient.right/2, rcClient.bottom-nCommandHeight, hwnd,
    (HMENU) IDC_EDITCONTROL, g_hInstance, NULL);
```

In the modified version, the TreeView control is positioned above the edit control, as shown here:

```
GetClientRect(hwnd, &rcClient);
//...
nCommandHeight = CommandBar_Height(hwndCB);
//...
hwndTV = CreateWindowEx(0, WC_TREEVIEW, TEXT("DB View Control"),
    WS_VISIBLE | WS_CHILD | TVS_HASLINES |
    TVS_LINESATROOT | TVS_HASBUTTONS, 0, nCommandHeight,
    rcClient.right, rcClient.bottom/2-nCommandHeight, hwnd,
    (HMENU) IDC_TREEVIEW, g_hInstance, NULL);

hEdit = CreateWindow(TEXT("EDIT"), TEXT("Property Information"),
    WS_VISIBLE | WS_CHILD | ES_MULTILINE | ES_READONLY |
    WS_VSCROLL,
    0, rcClient.bottom/2,
    rcClient.right, rcClient.bottom/2, hwnd,
    (HMENU) IDC_EDITCONTROL, g_hInstance, NULL);
```

In many cases, of course, the changes required will be more involved than those required by DBView's main window, and it might be necessary to maintain device- or platform-specific copies of each important dialog used in your application.

For example, if you look again at DBView, you will find that it has a dialog box it uses to allow the user to set which fields in a given table are indexed, and which are not. You'll also find that this dialog box will definitely not work on a PPC device, because it's over 290 pixels wide. Figures 2.7 and 2.8 show the original Modify Indexes dialog box, running on both HPC and PPC devices, respectively.

FIGURE 2.7:

The original Modify Indexes dialog box running on an HPC



Clearly, the original dialog box is not appropriate for both platforms. This is a perfect example of needing two copies of the same dialog box. Just as you did with the TreeView and edit controls of the main window, you'll need to reorganize the controls on the Modify Indexes dialog box. However, don't actually modify the original dialog. Instead, follow this simple three-step procedure for creating a copy of the original dialog:

1. Create a new dialog and name it something related to the original.
2. Mark and copy all of the controls from the original dialog into the new one. (This ensures that all of the control identifiers will remain the same. You can then essentially swap one dialog for another with a minimum of work.)
3. Resize and rearrange the controls until they're usable on the desired form factor.

When you do this for the Modify Indexes dialog, you end up again rearranging the controls so that they are vertically oriented. The result looks something like Figure 2.9.

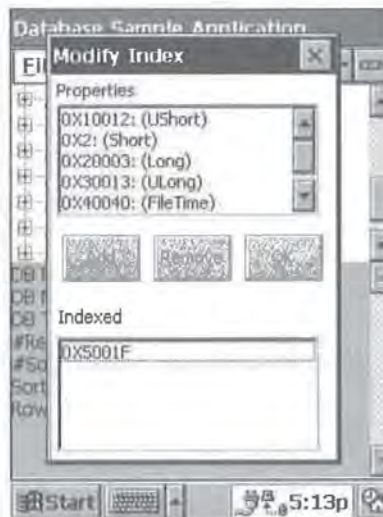
The three-step procedure outlined above allows you to make all of your programs useable and give them a professional look, regardless of what device they're running on.

FIGURE 2.8:

The original Modify Indexes dialog box running on a PPC

**FIGURE 2.9:**

The PPC-fixed version of the Modify Indexes dialog box



Maintaining a Single Codebase Is Nearly Impossible

While all of the form factors are running the same operating system, there are subtle differences between each form factor that make it very difficult to maintain a single codebase for your project.

NOTE

The term *codebase* refers to all of the source code, resource files, etc., that are used to create a given project.

Advantages of a Single Codebase

A single codebase can help prevent a situation like this:

You find a bug in, say, the HPC version source code. That same code is also in the PPC version source. Now you've got to make the same fix to both the HPC and the PPC source. When you then run your program on the PPC, you find that the change you've made breaks something else. Naturally, this also affects both versions.

The long and short of it is that you can drive yourself crazy keeping multiple codebases up-to-date and (relatively) bug-free. Therefore, the preferred way is to maintain only one codebase.

Typically, when you want to maintain one codebase that compiles for multiple platforms, you can rely on the availability of compiler defines. *Compiler defines* are global constants available at compile time that exist largely for the purpose of maintaining a single codebase. For example, you might want to perform some operation differently under the Windows CE emulator than you do on the device. For such a situation, you can use the `_WIN32_WCE_EMULATION` compiler define:

```
#if defined(_WIN32_WCE_EMULATION)
    //Do some emulator-specific code here
    MessageBox(hwnd, TEXT("You are in the emulator"), TEXT("em x86"),
        MB_OK);
#endif
```

The magic of compiler defines is that they actually control what blocks of code are compiled into the final executable. The above call to `MessageBox` would not be compiled into an application that was built for any platform other than the x86 emulator.

Visual C++ for Windows CE implements the Windows CE-related compiler defines listed in Table 2.4.

TABLE 2.4: Windows CE-Related Compiler Defines

Compiler Define	Question It Answers
<code>_WIN32_WCE</code>	Are you targeting a CE-based platform?
<code>_WIN32_CE_EMULATION</code>	Are you targeting the x86-based emulator?
<code>_MIPS_</code>	Are you targeting a MIPS-based device?
<code>_SH3_</code>	Are you targeting an SH3-based device?

You may notice that something important is missing from this list, however; namely, there are no compiler defines that you can use to determine whether your code is targeting an HPC, a PPC, an Auto PC, or some custom device type.

TIP

The AutoPC SDK implements the `#APC` compiler define to help determine whether you're compiling for an AutoPC device.

So, how can you determine what device type you're running on? As far as conditional compilation is concerned, there is no automatic way to do this, but there are two possible solutions to this problem:

1. Create your own conditional defines.
2. Create a runtime platform detector.

Creating Your Own Conditional Defines

The way that Microsoft recommends managing a single codebase is to create your own `#define` to indicate the platform you're targeting. In other words, if you're building for the Palm-size PC platform, add the following line to the relevant file or files:

```
#define _PPC
```

You then build your project for the Palm-size PC platform, remove or comment out the `#define`, and build for any other platforms.

TIP

You can also create a `#define` by simply adding it to the “Preprocessor defines” edit control on the C/C++ tab of Visual C++’s Project Settings dialog box.

While this does work, it requires you to remember that you’ve made this `#define` when you build for other platforms. This can be a bit tricky, so it’s best to use this method only when necessary. If you can find a way to implement your platform-specific code without conditional compiles, it will be simpler for you in the long run.

Creating a Runtime Platform Detector

A *runtime platform detector* is a function that attempts to guess the current platform while the program is running. This is useful for all platform operations that don’t require major changes to the source code. For instance, in the DBView example above, switching from one version of a dialog box to another is something that can be handled at runtime by a simple platform detector.

Microsoft recommends that you determine the device type by calling `SystemParametersInfo()` and passing `SPI_GETPLATFORMTYPE` as the requested information:

```
SystemParametersInfo(SPI_GETPLATFORMTYPE , 80, (PVOID)plat, 0);
```

The only problem with this is that not all of the OEMs support this item, so you have to use the screen dimensions to guess the platform type. This method is actually fairly reliable, although, eventually, there may be too much variety in the types of devices running CE to depend on it.

The screen-size method works by calling `GetSystemMetrics()` to get the height (pass in the `SM_CYSCREEN` constant) and the width (pass in the `SM_CXSCREEN` constant) of the device’s screen:

```
int GuessPlatform()
{
    int iHeight, iWidth;

    iHeight = GetSystemMetrics(SM_CYSCREEN);
    iWidth = GetSystemMetrics(SM_CXSCREEN);
```

Once you have these dimensions, you can make a few simple comparisons to determine the device type. First, check to see if the screen is long and narrow (i.e.,

is the height greater than the width?). If it is, assume you're running on a Palm-size PC:

```
if (iHeight > iWidth)
{
    return(PLAT_PPC);
}
```

On the other hand, if the device has a screen that's short and wide, you must perform some additional tests to narrow down your choices. First, check if the device has a screen like the HPC Pro devices (assumed to be 640 × 480).

```
if ((iHeight == HPCPRO_HEIGHT) && (iWidth == HPCPRO_WIDTH))
{
    return(PLAT_HPCPRO);
}
```

Next, check to see if the height of the screen is equal to the height of an Auto PC's screen, assumed to be 16 pixels:

```
else if (rcClient.bottom == APC_HEIGHT)
{
    return(PLAT_APC);
}
```

As the default condition, if the screen size is not that of an HPC Pro and not that of an Auto PC, then assume the device is a standard HPC:

```
else
{
    return(PLAT_HPC);
}
```

Finally, as a catchall condition, if the height and width of the screen are equal, return a value indicating an unknown type:

```
else
{
    return(PLAT_OTHER);
}
```

When it's all put together, the function looks like this:

```
int GuessPlatform()
{
    int iHeight, iWidth;
```



```
iHeight = GetSystemMetrics(SM_CYSCREEN);
iWidth = GetSystemMetrics(SM_CXSCREEN);

if (iHeight > iWidth)
{
    return(PLAT_PPC);
}
else if (iHeight < iWidth)
{
    if ((iHeight == HPCPRO_HEIGHT) && (iWidth == HPCPRO_WIDTH))
    {
        return(PLAT_HPCPRO);
    }
    else if (iHeight == APC_HEIGHT)
    {
        return(PLAT_APC);
    }
    else
    {
        return(PLAT_HPC);
    }
}
else
{
    return(PLAT_OTHER);
}
}
```

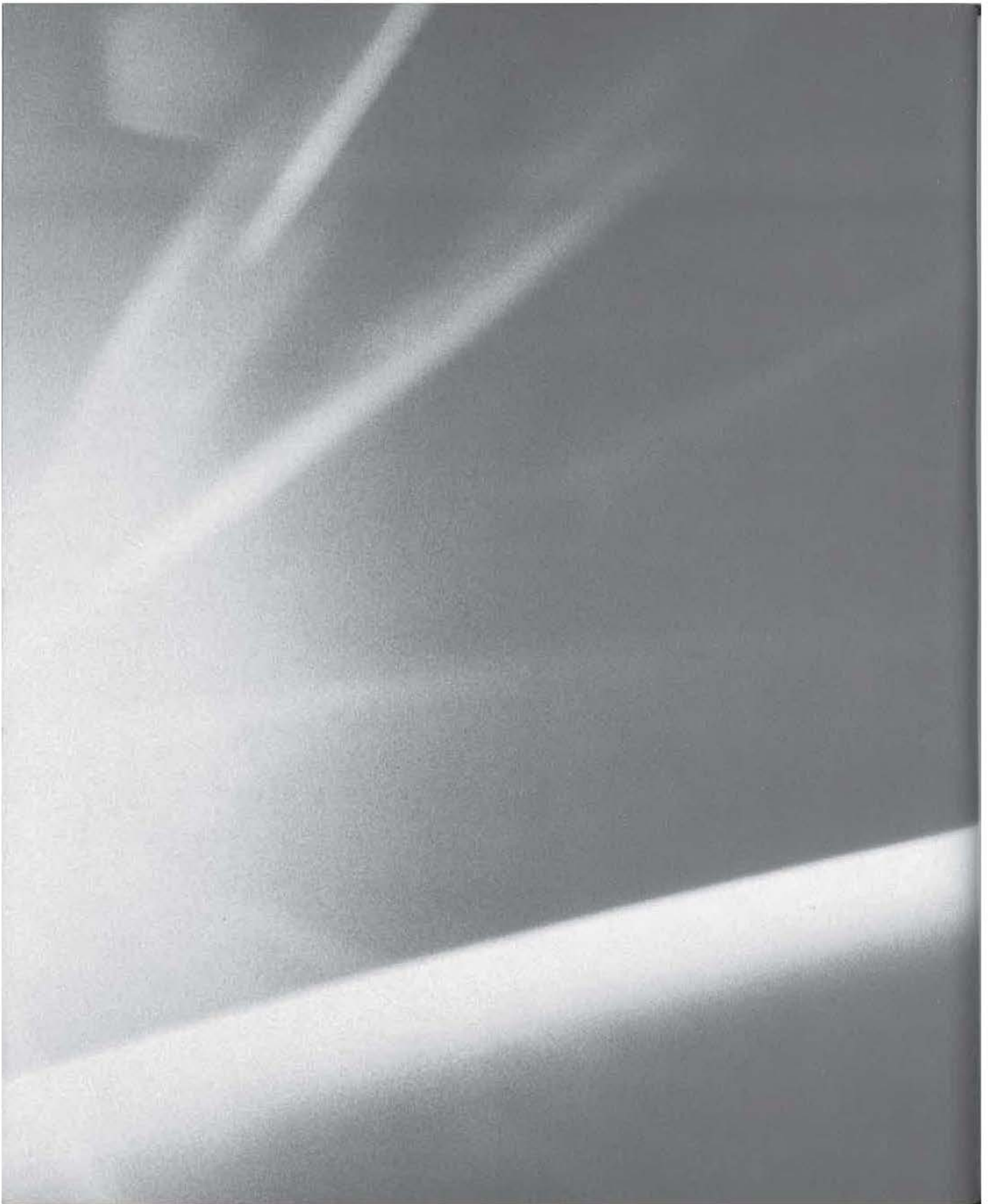
Although this runtime platform detector may look easier to implement than the “roll your own” conditional defines, there are some situations that the runtime detector can’t handle, and there will be times when you must compile and link different pieces of code into the program depending on the target platform.

In your own applications, you will probably find that you need to employ a combination of the two techniques to successfully deal with any platform-dependant issues that arise.

Summary

In this chapter, you explored some of the issues that you are faced with when programming for Windows CE. Specifically, you learned the ins and outs of the Unicode character set and how it affects CE programming. You also learned how to tailor your code for the low-memory environment of Windows CE. And, finally, you saw how to deal with the different form factors with a minimum of effort.

Although some of these issues might seem difficult to navigate, it only takes a little bit of practice before they become second nature.



CHAPTER

THREE

3

To C or Not to C?

- `calloc()`
- `MoveToEx90/LineTo()`
- `WM_RBUTTONDOWN`
- Recreating the `fopen()/FILE*`-based functions

This chapter will give the experienced C/C++ programmer a heads up as to what's missing in the C/C++ language for Windows CE. Most importantly, there are a number of programming constructs and functions that have been removed. For instance, there is no `try..catch` construct. Similarly, many standard functions such as `fprintf()` are gone.

The long and short of all this is that whether you're writing a program from scratch or porting an existing application, you'll definitely run into some roadblocks. In this chapter, we'll take a look at what's missing, how to deal with it, and what's still there. The most basic of C library functions do exist for Windows CE, and many API calls also still exist. However, a quick glance at the different include (.h) file directories (the standard Visual C++ files and the Windows CE files) reveals just how much of the C/C++ runtime library and Windows API is missing: there are roughly 350 standard include files, but there are less than 120 CE include files. Ouch! Some of the more important functions/constructs that are not available to you when coding for CE are listed below:

- `calloc()`
- `MoveToEx()/LineTo()`
- `WM_RBUTTONDOWN`
- `try..catch`
- `FILE*`, `fopen()`, `fprintf()`, etc.

When you need to use one of these functions, there are three approaches you can take:

1. Search for a substitute function provided by the CE runtime libraries or perhaps the CE API.
2. Consider changing the logic of your program to work around the missing functionality.
3. Write your own function to perform the same task as the function that's missing.

NOTE

CE 2.11 does support the full `stdio.h` library of functions. However, this is of very little practical use to those of us who need to target all of the other devices already on the market.

Let's see how these choices can be applied in a practical way to our list of missing functions.

Finding Substitute Functions

Whenever possible, the best solution to dealing with a missing API function is to find a substitute function that will perform the same operations as the original. In this section, we'll take a look at how this applies to three common operations: `calloc()`, `MoveToEx()` / `LineTo()`, and `WM_RBUTTONDOWN`.

calloc()

As you probably know, `calloc()` is one of the more important functions in the C/C++ language. It is one of the primary ways that you can allocate and initialize a block of memory. Unfortunately, it does not exist in CE's runtime library.

Thankfully, `LocalAlloc()`, another allocation function, *does* exist under CE. The catch is that `calloc()` allocated and initialized the memory, whereas `LocalAlloc()` requires you to explicitly request that it initialize the memory by specifying the `LMEM_ZEROINIT` flag, like this:

```
ptr = LocalAlloc(LMEM_ZEROINIT, size);
```

MoveToEx() / LineTo()

While perhaps not as important as `calloc()`, `MoveToEx()` and `LineTo()` are important Windows API calls for drawing lines onto an HDC. The two functions are usually used in a pair, as follows:

```
RECT rt;    GetClientRect(hWnd, &rt);
MoveToEx(hdc, 0, 0, NULL);
LineTo(hdc, rt.right, rt.bottom);
```

Unfortunately, they too do not exist under Windows CE.

NOTE

Although `MoveToEx()/LineTo()` do not exist in the Windows CE API, they do exist in MFC for CE.

A quick search through the API documentation, though, reveals that `PolyLine()`, another API function, is the recommended substitute. Using `PolyLine()`, you can draw exactly the same line that the previous code sample produces, as shown below. (This code can be found on the CD in the directory for this chapter as `MoveToEx_LineTo.txt`.)

```
POINT Points[2];
RECT rt;
GetClientRect(hWnd, &rt);
Points[0].x = 0;
Points[0].y = 0;
Points[1].x = rt.right;
Points[1].y = rt.bottom;
Polyline(hdc, &Points[0], 2);
```

WM_RBUTTONDOWN

`WM_RBUTTONDOWN` is the message that Windows sends your program every time the user right-clicks the mouse. Of course, since most CE devices (all of the PPC and HPC class devices, in fact) do not have any mouse at all, there is also no right mouse button, and thus no right-click message. But there are still times when you'd like to be able to offer a pop-up (i.e., right-click) menu. Under Windows CE, a right-click is simulated with a combination of pressing the Alt key and tapping the stylus to the screen, also known as `WM_LBUTTONDOWN`.

TIP

For a demonstration of the Alt-tap behavior, just try pressing the Alt key while clicking on your HPC device's Desktop.

The way to support pop-up menus and right-click functionality is to handle the stylus tap (which translates to a `WM_LBUTTONDOWN` message), while checking to

see whether the Alt key is depressed. Under Windows 98/NT, you would use code that relies on `WM_RBUTTONDOWN`:

```
        case WM_RBUTTONDOWN:
        {
            MessageBox(hwnd, TEXT("Show your pop-up menu"), TEXT("wow!"),
MB_OK);
            break;    }
```

Under Windows CE, however, your code would instead use `WM_LBUTTONDOWN` and then test whether the Alt key (`VK_MENU` virtual key code) was depressed (as shown in the file `WM_RBUTTONDOWN.txt`).

WM_RBUTTONDOWN.txt

```
        case WM_LBUTTONDOWN:
        {
            if (GetAsyncKeyState(VK_MENU))
            {
                MessageBox(hwnd, TEXT("Show your popup menu"),
TEXT("wow!"), MB_OK);
            }
            break;
        }
```

NOTE

This technique does not apply to Palm-size PCs, since programs running on a PPC device cannot support the Alt-click combination at all. The Input Panel (keyboard) of a PPC device does not have an Alt key.

Changing the Program's Logic: *try..catch* and Exceptions

While some of the missing API functions do have suitable replacements, this isn't always the case. Some constructs have simply been removed entirely, and there are no substitutes available. In those situations, you're forced to alter the actual logic of your programs to work around this fact.

When it comes to having to restructure your program's logic, the worst offender is the `try . . catch` construct. This construct is a wonderfully flexible way for your program to be notified of an error (exception) created in a block of code. Your program can then gracefully handle it in the appropriate manner.

But, as you may have guessed, it does not exist under CE. And, because `try . . catch` is something that the compiler itself, as opposed to a set of functions, must implement, there isn't really a good way to recreate it yourself. Therefore, your only choice is to rework your existing code so that you add enhanced error checking and to make sure that you've thoroughly tested your application before shipping it.

NOTE

This only applies to the C++-based exception-handling methods. All of the Win32 API-based exception handling still exists. However, there are some subtle differences between the two that may make it difficult to port from one to the other.

Writing Your Own Functions

At the beginning of this chapter, we listed five functions that are missing in CE: `Calloc()`, `MoveToEx()/LineTo()`, `WM_RBUTTONDOWN`, `try . . catch`, and `FILE*` (along with `fopen()`, `fprintf()`, etc.). In this list, there is one item that is particularly important, probably because you've come to count on its existence more than on any of the others. Specifically, it's the `FILE */fopen()/fprintf()` set of functions contained in `stdio.h`. These are among the most basic file-handling functions of the C language, and almost every C programmer is well versed in their use. However, Microsoft did not implement them on CE. Instead, Microsoft implemented only the following three Win32 API file-access functions:

- `CreateFile()`
- `ReadFile()`
- `WriteFile()`

If you've ever looked at these functions, you know that they don't offer the same functionality of formatted output as, say, `fprintf()` does. That means those of us who prefer the FILE*-based functions are left with only two choices:

1. Struggle with the `CreateFile()` family of functions. This means that you have to do extensive work on any FILE*-based code that you want to port to CE. Most people don't like this option for the following reasons:
 - If you were porting legacy code, you'd have to find all of the calls to the FILE*-based functions and convert them over by hand, one at a time.
 - Even if you're not porting legacy code, many longtime C programmers feel more comfortable with the FILE*-based functions and are probably not anxious to learn an entirely new system for file access.
2. Create wrapper functions around the `CreateFile()` family of functions that will elegantly simulate and recreate the FILE* family of functions in such a way that you can use both sets of functions interchangeably. This also means that you will have a much easier time of porting your existing FILE*-based code to CE.

NOTE

Wrapper functions are functions used to insulate us from or *wrap* functions with complex parameters. For instance, the `CreateProcess()` function has 10 parameters, but 5 of them are always NULL on Windows CE. Therefore, it might make sense to create a wrapper function for `CreateProcess()` (say, `MyCreateProcess()`) that would have 5 parameters. These parameters, in turn, would call `CreateProcess()`, passing in the 5 values it received, as well as the 5 NULLs.

Naturally, option number two is the one to choose. The up-front investment you must put in to re-create these functions will pale in comparison to the amount of time and energy you would waste trying to port your existing code or learn the API's file-access functions.

FILE*

All of the FILE*-based functions identify an open, active file via a *file pointer*. All of the API's file access functions identify an open, active file via a *file handle*.

The Microsoft Developer Network Glossaries define a file pointer as a pointer to "the next byte to be read or the location to receive the next byte written in a file."

A file handle, in contrast, is “a unique identifier that Windows assigns to a file when the file is opened or created. A file handle is valid until the file is closed.”

Your first job then is to try and figure out how to implement the `FILE*` pointer type, while still remaining compatible with the `CreateFile()`-based functions of the API. There are two reasons you’d want your file-access routines to remain compatible with the API’s routines:

- Even though you’re going to be creating some sophisticated file-access functions, you’ll still need to be able to use the `CreateFile()` functions to do the core work.
- By maintaining compatibility between our routines and those of the API, you’ll be able to seamlessly mix and match functions from one group with those from another group.

Although there may be other ways to do this, the most elegant way is to first `#define` the `FILE` type to be equivalent to, or in this case, another name for, standard Win API `HANDLE`s:

```
#define FILE HANDLE
```

Then, declare a simulated file pointer just as you would if you were writing a Desktop application:

```
FILE* fp;
```

This means that you’ve made `FILE *` equivalent to `HANDLE *`.

Clearly, though, your new file pointer is not the same file pointer you would have used when writing a Desktop application. However, since you’re rewriting those routines from scratch, and since you just want them to *appear* to be the same as the original functions, this substitution is perfectly acceptable.

What that means is that you can design your `FILE*`-based functions to take this faked file pointer and then simply dereference (i.e., use the contents of the address being pointed to) that file pointer to retrieve the true file handle. Or, using C- pseudocode, we can illustrate it like this:

```
FILE* fp;  
//...  
fsomefileaccess(fp);  
//...
```

Then in the implementation of `fsomefileaccess`, we would have:

```
//...
```

```
ReadFile(*fp, ...); //API call : ReadFile
//...
```

Although this doesn't look like much, it meets both of your goals perfectly. You can use your file access functions with your fake file pointer, and you can use all of the API file access functions by dereferencing your file pointer.

Now let's move on to the functions themselves.

NOTE

All of the code for the FILE*-related routines can be found on the CD in the files `cestdio.c` and `cestdio.h`.

fopen()

In the standard C runtime library (RTL), `fopen()` is defined as follows:

```
FILE *fopen( const char *filename, const char *mode );
```

`fopen()` opens the file specified by `filename`, and returns either a file pointer or `NULL` (if the request fails). The `mode` parameter is a string that specifies the type of file access:

Mode	What It Means
r	For reading
r+	For reading and writing
w	For writing
w+	For reading and writing
a	For appending
a+	For appending and reading

The original runtime library (RTL) `fopen()` function supports some additional values for the `mode` parameter, but we've decided not to implement them here since they don't mean anything in Windows CE. For instance, `t` (for text mode) is a `mode` value that we do not support because the underlying CE API functions do not support text mode file access. Because the `fopen()` function takes only character strings as parameters, the first question you must address when creating your own `fopen()` is whether your `fopen()` function should require Unicode strings or ANSI strings.

While it is true that the original `fopen()` takes ANSI strings, CE is a Unicode-based operating system, and most, if not all, of the CE API functions that take string parameters require Unicode strings. Therefore, for the sake of consistency with the CE API, use Unicode strings for your `fopen()` function. The CE `fopen()`, then, will be defined as follows:

```
FILE * fopen(LPWSTR filename, LPWSTR mode);
```

As you can see, the `char *` parameters have been replaced with `LPWSTR` parameters. `LPWSTR` stands for “long pointer to wide (Unicode) string.”

Next, let’s turn our attention to the implementation of our `fopen()` function.

```
LocalAlloc(LMEM_ZEROINIT, LocalFree(
```

The first thing that `fopen()` does is allocate memory for your simulated file pointer, as follows:

```
h = (HANDLE *)LocalAlloc(LMEM_ZEROINIT, 1 * sizeof(HANDLE));
```

This is the magic that makes the entire file-pointer simulation possible, as you’ll see when you call the CE API’s `CreateFile` function.

Just as your `fopen()` has “flags” that can be set via the mode parameter, `CreateFile()` also has a set of flags that must be set to indicate the type of file access desired. The next block of code examines the mode string and converts the value(s) passed in that string into the corresponding values for the `CreateFile()` flags.

First, there’s a test to determine whether or not a “modifier” (i.e., the “+” character) is specified in the mode flags:

```
if(wstrlen(mode) > 1)
{
    modifier = mode[1];
}
```

Then, use a `switch..case` statement to actually set the `CreateFile` flags based on the mode string. In the `switch..case` statement, the first character of the mode string is examined to see what type of file access is requested:

```
case (TCHAR)'r':
```

NOTE

`TCHAR` is a character type which, in the case of Windows CE and any Unicode-based operating system, equates to a 2-byte or *wide* character. (See Chapter 2 for more information.)

Then, based on this value, the appropriate `CreateFile` flags are set:

```
flag = GENERIC_READ;
opentype = OPEN_EXISTING;
```

This is repeated for each of the possible values and modifiers. The complete `switch..case` statement follows:

```
switch((TCHAR)mode[0])
{
    case (TCHAR)'r':
        flag = GENERIC_READ;
        opentype = OPEN_EXISTING;
        if(modifier == (TCHAR)'+')
        {
            flag = GENERIC_READ | GENERIC_WRITE;
        }
        break;
    case (TCHAR)'w':
        flag = GENERIC_WRITE;
        opentype = CREATE_ALWAYS;
        if(modifier == (TCHAR)'+')
        {
            flag = GENERIC_READ | GENERIC_WRITE;
            opentype = CREATE_ALWAYS;
        }
        break;
    case (TCHAR)'a':
        flag = GENERIC_WRITE;
        opentype = OPEN_ALWAYS;
        if(modifier == (TCHAR)'+')
        {
            flag = GENERIC_READ | GENERIC_WRITE;
            opentype = CREATE_ALWAYS;
        }
        break;
    default:
        break;
}
```

Once you're out of the `switch...case` statement, it's time to call to `CreateFile()`, passing in the flags you've just configured:

```
hFiletmp = CreateFile(filename, flag, FILE_SHARE_READ,
    NULL, opentype, FILE_ATTRIBUTE_NORMAL, NULL);
```

The first thing you must do now is to test the return value to determine whether or not `CreateFile` was successful. If it was—and this is where the file pointer magic comes in—you then take the file *handle* returned by `CreateFile` and assign that value to the contents of the file *pointer* that will be returned by your `fopen()` function!

```
if(hFiletmp != INVALID_HANDLE_VALUE)
{
    *h = hFiletmp;
}
```

If the call to `CreateFile()` was not successful, however, you must free the memory you allocated for the file pointer and return a `NULL` to indicate failure:

```
else
{
    LocalFree(h);
    return(NULL);
}
```

Finally, if everything went smoothly, return the file pointer:

```
return(h);
```

That's how to simulate `fopen()`. The complete function is shown below:

```
FILE * fopen(LPWSTR filename, LPWSTR mode)
{
    HANDLE hFiletmp;
    HANDLE *h = NULL;
    DWORD opentype = OPEN_ALWAYS;
    DWORD flag = GENERIC_READ | GENERIC_WRITE;
    TCHAR modifier = (TCHAR)'\\0';

    // part of magic to simulate familiar FILE handle
    h = (HANDLE *)LocalAlloc(LMEM_ZEROINIT, 1 * sizeof(HANDLE));

    // use mode flags
```

```

if(wstrlen(mode) > 1)
{
    modifier = mode[1];
}
switch((TCHAR)mode[0])
{
    case (TCHAR)'r':
        flag = GENERIC_READ;
        opentype = OPEN_EXISTING;
        if(modifier == (TCHAR)'+')
        {
            flag = GENERIC_READ | GENERIC_WRITE;
        }
        break;
    case (TCHAR)'w':
        flag = GENERIC_WRITE;
        opentype = CREATE_ALWAYS;
        if(modifier == (TCHAR)'+')
        {
            flag = GENERIC_READ | GENERIC_WRITE;
            opentype = CREATE_ALWAYS;
        }
        break;
    case (TCHAR)'a':
        flag = GENERIC_WRITE;
        opentype = OPEN_ALWAYS;
        if(modifier == (TCHAR)'+')
        {
            flag = GENERIC_READ | GENERIC_WRITE;
            opentype = CREATE_ALWAYS;
        }
        break;
    default:
        break;
}
// actually make the file per params.
hFiletmp = CreateFile(filename, flag, FILE_SHARE_READ,
NULL, opentype, FILE_ATTRIBUTE_NORMAL, NULL);

// if we have a valid handle give it, else we return NULL
if(hFiletmp != INVALID_HANDLE_VALUE)
{

```



```
        *h = hFiletmp;
    }
    else
    {
        LocalFree(h);
        return(NULL);
    }
    return(h);
}
```

Let's now take a look at `fclose()`, which is considerably simpler.

fclose()

The original `fclose()` is defined as follows:

```
int fclose( FILE *stream );
```

As you can see, `fclose()` takes the single `FILE*` pointer parameter and returns an integer: 0 to indicate success or EOF to indicate an error. Because no strings are involved and because file pointers have been simulated, the CE `fclose()` can follow the existing definition exactly.

The definition done, let's examine the implementation of our `fclose()`. The first step is to attempt to close the file handle you got from your initial call to `CreateFile()` from within `fopen()`. Do this by calling `CloseHandle()`, a fairly generic API function for closing almost any type of open or active handle, whether it's for a file, a process, or other handle type.

Because `CloseHandle()` returns a `BOOL`, you can easily test the result to determine what type of value you should return from your `fclose()`. So, if `CloseHandle()` fails, free your file pointer—you'd have to do this anyway—and return EOF to indicate an error:

```
if (!CloseHandle(*fp))
{
    LocalFree(fp);
    return(EOF);
}
```

In order to implement `fclose()` properly, you must define the value of EOF because EOF is defined in a header file that does not exist under CE. Furthermore, the `CreateFile()`-based functions do not return such a value themselves, so

until now there was no need for EOF to be defined. Therefore, you must define it yourself, as follows:

```
#define EOF      -1
```

It's not terribly complex, but it is important, just the same.

If the call to `CloseHandle()` was a success, however, free your file pointer—again, something you'd have to do anyway—and return 0 to indicate success:

```
else
{
    LocalFree(fp);
    return(0);
}
```

That's all there is to recreating `fclose()`. The complete function is shown below:

```
int fclose(FILE * fp)
{
    if (!CloseHandle(*fp))
    {
        LocalFree(fp);
        return(EOF);
    }
    else
    {
        LocalFree(fp);
        return(0);
    }
}
```

Next, let's examine how to re-create some of the simpler `FILE*`-based functions, and we'll work our way up to the more complex ones.

fgetc()

`fgetc()` is a function that reads one character from the file and returns that character as an integer value. The original `fgetc()` is defined as follows:

```
int fgetc( FILE *stream );
```

For our purposes, this definition will work just fine.

However, the original `fgetc()` returns an integer. You'll be returning a Unicode character, but an integer will work just fine. That's because an integer is 4 bytes long, whereas a Unicode character is only 2 bytes long. Therefore, you can use the existing definition for `fgetc()` without change.

The heart of this function is the actual call to `ReadFile()`, which reads a single TCHAR from the file specified by the file handle supplied:

```
ret = ReadFile(*fp, charbuf, sizeof(TCHAR), &num, NULL);
```

Then, as you might expect, a check of the value returned by `ReadFile` determines what your `fgetc()` will return. If `ReadFile()` was not successful, you'll return EOF; otherwise, you'll return the TCHAR read by `ReadFile()`:

```
    if(ret == FALSE)
    {
        return(EOF);
    }
    return(charbuf[0]);

int fgetc( FILE *fp )
{
    BOOL ret = FALSE;
    DWORD num = 0;
    TCHAR charbuf[2];

    ret = ReadFile(*fp, charbuf, sizeof(TCHAR), &num, NULL);
    if(ret == FALSE)
    {
        return(EOF);
    }
    return(charbuf[0]);
}
```

Now that we've looked at reading a single character, let's take a look at writing a single character.

fputc()

`fputc()` is the other half of `fgetc`; it writes a single character to the file and returns the character written. The original `fputc()` is defined as follows:

```
int fputc( int c, FILE *stream );
```

Your `fputc()` will be defined slightly differently in that you'll want to ensure that the first parameter passed—in the case of the original `fputc()`, an integer—is explicitly typed as a `TCHAR`. This is mostly for consistency's sake. If you feel strongly that you'd like to change it back to an integer, you are welcome to do so. Your definition of `fputc()`, then, looks like this:

```
int fputc(TCHAR value, FILE * fp);
```

To implement it, initialize the data that you'll actually be passing to `WriteFile()`:

```
TCHAR msg[2];
msg[0] = (TCHAR)value;
msg[1] = (TCHAR)'\0';
```

Next, call `WriteFile()` and, as always, check the return value to determine whether to return `EOF` (failure) or the character written (success):

```
if (!WriteFile(*fp, msg, sizeof(TCHAR), &num, NULL))
{
    return(EOF);
}
else
{
    return(value);
}
```

Again, it's fairly simple logic. The implementation looks like this:

```
int fputc(TCHAR value, FILE * fp)
{
    DWORD num = 0;
    TCHAR msg[2];
    msg[0] = (TCHAR)value;
    msg[1] = (TCHAR)'\0';
    // data
    if (!WriteFile(*fp, msg, sizeof(TCHAR), &num, NULL))
    {
        return(EOF);
    }
    else
    {
        return(value);
    }
}
```

Now, let's move on to writing out full strings, instead of just a single character at a time.

fgets()

`fgets()` is a function that reads a string from the file specified. It will read up to and including the first newline character or it will read until it has read `n` characters from the file. `fgets()` then returns either the string read or EOF to indicate an error. The RTL `fgets()` is defined as follows:

```
char *fgets( char *string, int n, FILE *stream );
```

As you've probably guessed, your definition will be slightly different, in that you'll need to change all of the `char *` pointers to Unicode strings (`LPWSTR`). Your definition of `fgets`, then, will look like this:

```
LPWSTR fgets(LPWSTR msg, int n, FILE *fp );
```

As you can see, it's very close to the original `fgets()`.

In its implementation, this function basically reads one character at a time, via a `do...while` loop:

```
do
{
    ret = ReadFile(*fp, charbuf, sizeof(TCHAR), &num, NULL);
```

If `ReadFile()` fails for any reason, it breaks out of the loop:

```
    if(ret == FALSE)
    {
        return(EOF);
    }
```

The next block is slightly more complex. Here, you read and append every non-NULL character you read to your return value. However, there's an extra `if` statement embedded in this block of code that will keep you from reading past a newline character:

```
    if(charbuf[0] != (TCHAR)'\0')
    {
        msg[count] = charbuf[0];
        count++;
        if (charbuf[0] == (TCHAR)'\n')
            break;
    }
```

This is consistent with the original `fgets()`, which also would read until a newline character was encountered, or until it had read `n` characters. This brings you to the end of the `do . . while` loop, where it's time to test to ensure that you do not read more than `n` characters:

```
    } while( count <= n );
```

Next, test to make sure that you did, in fact, read something from the file. If you didn't, return `EOF` to indicate an error:

```
    if (count == 0) //no characters read, error
    {
        return(EOF);
    }
```

If you did manage to read some data from the file, however, return the string that was read:

```
    msg[count] = (TCHAR)'\0';
    return(msg);
```

You're done! The completed function follows:

```
LPWSTR fgets(LPWSTR msg, int n, FILE *fp )
{
    BOOL ret = FALSE;
    DWORD num = 0;
    TCHAR charbuf[2];
    int count = 0;

    do
    {
        ret = ReadFile(*fp, charbuf, sizeof(TCHAR), &num, NULL);
        if(ret == FALSE)
        {
            return(EOF);
        }
        if(charbuf[0] != (TCHAR)'\0')
        {
            msg[count] = charbuf[0];
            count++;
            if (charbuf[0] == (TCHAR)'\n')
                break;
        }
    } while( count <= n );
```

```

        if (count == 0) //no characters read, error
        {
            return(EOF);
        }
        msg[count] = (TCHAR)'\0';
        return(msg);
    }

```

Now let's take a look at `fputs()`, the companion function to `fgets()`.

fputs()

`fputs()` writes a string up to (but not including) its NULL character to the file specified. If successful, it returns a positive number; otherwise, it returns EOF. The original RTL `fputs()` is defined as follows:

```
int fputs( const char *string, FILE *stream );
```

Your `fputs`, of course, will change the `char *` parameter to an `LPWSTR`, as with the previous functions:

```
int fputs(LPWSTR msg, FILE * fp);
```

The implementation of your `fputs()` is fairly straightforward. Call `WriteFile()`, and based on the result, return either EOF or the number of bytes written to the file:

```

int fputs(LPWSTR msg, FILE * fp)
{
    DWORD num = 0;

    if (!WriteFile(*fp, msg, wcslen(msg), &num, NULL))
    {
        return(EOF);
    }
    else
    {
        return(num);
    }
}

```

We'll now look at `fread()` and `fwrite()`.

wstrlen()—More on Unicode

As you look at the code for the `fputs()` function, you may notice one function that's not defined anywhere in the Microsoft documentation: `wstrlen()`. `wstrlen()` is a function created strictly for the purpose of dealing with Unicode and file access under CE. Simply put, it will return the length, in bytes—not characters—of a Unicode string.

The existing Unicode string length function of CE, `wcslen()`, returns string length in terms of the number of characters in the string. However, `WriteFile()` and other API file-access functions require the number of bytes. You'll recall that Unicode strings are 2 bytes long, hence the difference, making the number of bytes is twice the number of characters. Because of this difference, you need a function that will do the conversion.

`wstrlen`

is defined as follows:

```
int wstrlen(LPWSTR wstr)
{
    return(wcslen((LPWSTR)wstr) * sizeof(TCHAR));
}
```

You'll find that `wstrlen()` is used in a number of the functions developed in this chapter.

fread()

The original RTL `fread()` is defined as follows:

```
size_t fread( void *buffer, size_t size, size_t count, FILE *stream );
```

`fread()` reads `count` items (each one of size `size`) from the file and returns the number of items read or, if there's an error, returns 0 to indicate no items were read. The `size` parameter, then, specifies the number of bytes each item occupies. In other words, to calculate the total number of bytes read, you could multiply `size` by `count`.

Your `fread()` looks and acts exactly the same as the original:

```
int fread(void *buffer, int size, int count, FILE *fp )
{
    BOOL ret = FALSE;
    DWORD num = 0;
```



```

        ret = ReadFile(*fp, buffer, size * count, &num, NULL);
        return(num / size);
    }

```

Here, all you have to do is call `ReadFile`, calculate the total number of bytes to be written (`size * count`), and return the number of items (`num / size`) actually written. If there is an error of any kind, `num` will still be 0, so `fread()` will correctly return 0 to indicate an error.

Not surprisingly, `fwrite()` works in much the same manner.

fwrite()

The original RTL `fwrite()` is defined as follows:

```

size_t fwrite( const void *buffer, size_t size, size_t count, FILE
               *stream );

```

`fwrite()` works exactly like `fread()`, except that it writes data instead of reading it. `fwrite()` writes `count` items (each one of size `size`) from the file and returns the number of items written. The `size` parameter, then, specifies the number of bytes each item occupies. In other words, to calculate the total number of bytes to be written, you could multiply `size` by `count`. `fwrite()` returns the number of items written or, if there's an error, returns 0 to indicate no items were written.

Your `fwrite()` will be defined as follows:

```

int fwrite(const void *buffer, int size, int count, FILE *fp )
{
    DWORD num = 0;
    WriteFile(*fp, buffer, size * count, &num, NULL);
    return(num / size);
}

```

As you can see, `fwrite()` calls `WriteFile()`, calculates the total number of bytes (`size * count`), and returns the number of items read (`num / size`). Again, just as with `fread()`, if there is an error of any kind, `num` will still be 0, so `fwrite()` will correctly return 0 to indicate an error.

Next, we come to the pair of functions that you will probably use more than any other, `fprintf()` and `fscanf()`. These functions are also more complex than those that you've already implemented in this chapter.

But What If I Don't Want to Use Unicode?

On the CD for this book, there are ANSI-based versions of all of the Unicode files implemented here. They are named according to the Win32 API standard for indicating an ANSI-based function; namely, they have a capital letter *A* appended to their name. For instance, the Unicode version of the function `fprintf()` is simply `fprinf()`, while the ANSI-based function is named `fprintfA()`.

These ANSI functions will be of particular interest to anyone who is porting significant volumes of legacy code that explicitly uses `char *` or `LPSTR`, both of which are ANSI string types.

For more on helping your application to be Unicode-aware, see Chapter 2.

fprintf()

The original RTL `fprintf()` is defined as follows:

```
int fprintf( FILE *stream, const char *format [, argument ]...);
```

According to the Microsoft documentation, `fprintf()` “formats and prints a series of characters and values” to the file specified. “Each function argument (if any) is converted and output according to the corresponding format specification in format.” The format string uses the same format-specifiers as `printf()` and `wsprintf()`.

TIP

The three dots in the function definition mean that the function takes a variable and an unknown number of arguments. It is up to the implementers of the function to parse the variable argument list.

The only difference between your `fprintf()` and the original `fprintf()` is that yours will, of course, change the `char *` parameter (the format string) to an `LPWSTR` Unicode string.

As you can see in the code below, the real trick to implementing `fprintf()` is in the variable argument list:

```
int fprintf(FILE * fp, LPWSTR format, ...)  
{
```

```

DWORD num = 0;
TCHAR msg[512];
unsigned short *pmsg = NULL;
int len = 0;
BOOL bResult;

va_list marker;
va_start( marker, format );

pmsg = (TCHAR *)&msg[0];
wvsprintf(pmsg, format, marker);
// get string length for WriteFile call.
len = wcslen((LPWSTR)pmsg);
bResult = WriteFile(*fp, pmsg, len, &num, NULL);

va_end( marker );
if (bResult)
{
    return(num);
}
else
{
    return(EOF);
}
}

```

Because this function is a bit complex, let's take each part and examine it.

First, you must set up the variable argument list:

```

va_list marker;
va_start( marker, format );

```

Next, declare the argument list with the `va_list` macro. Then, call `va_start()`, passing the variable argument list and the parameter that appears just before the variable argument list in the function declaration.

Next, use the little-known variant of `wsprintf` to help you do your work:

```

pmsg = (TCHAR *)&msg[0];
wvsprintf(pmsg, format, marker);

```

`wvsprintf()` differs from `wsprintf()` in that it takes the entire variable argument list as a single variable (i.e., one declared with the `va_list` macro) instead

of the actual arguments themselves. This means that you don't have to do any parsing of the format string and the variable argument list yourself!

Then, call `WriteFile()`, passing it the string you just got back from `wvsprintf()`:

```
len = wcslen((LPWSTR)pmsg);
bResult = WriteFile(*fp, pmsg, len, &num, NULL);
```

Now that you're finished with the variable argument list, use the `va_end()` macro to indicate that:

```
va_end( marker );
```

Finally, check the value you got back from `WriteFile()` and indicate whether or not your operation has been successful:

```
if (bResult)
{
    return(num);
}
else
{
    return(EOF);
}
```

And you're done. On to `fscanf()`, the next function on our list.

fscanf()

The Microsoft Developer Network documentation defines `fscanf()` as:

```
int fscanf( FILE *stream, const char *format [, argument ]... );
```

"The `fscanf()` function reads data from the current position of stream into the locations given by argument (if any). Each argument must be a pointer to a variable of a type that corresponds to a type specifier in *format*."

In other words, `fscanf()` is used to read formatted data from a file. Just as with `fprintf()`, the format string specifies the data types of the various items being read. And, just as with the other functions you've worked on, the only parameter that needs to change is *format*, which you can implement as an `LPWSTR` instead of a `char *`.

`fscanf()` is the longest (in terms of lines of code) of our functions, so it's easiest to break it into steps. Begin with the same variable argument handling that you used in `fprintf()`:

```
va_list ap;
va_start(ap, format);
```

Here you declare the argument list "ap" with the `va_list` macro. Then, call `va_start()`, passing the variable argument list and the parameter that appears just before the variable argument list in the function declaration.

The next step is to set up a `while` loop and `switch...case` statement that will iterate through all of the characters in the format string:

```
while (*format)
{
    count = 0;
    switch(*format++)
    {
```

TIP

The `*format++` in the `switch` statement increments the `*format` string pointer (LPWSTR) so that it automatically points to the next character in the format string.

This is how you know the data type of the items you're reading from the formatted input file. It is also how you know the expected data types of the items in the variable argument list, as you'll see from the `case` portion of your `switch` statement. Here's the first condition, which tests for, and then reads, a string from the file:

```
case (TCHAR)'s': // string
    s = va_arg(ap, TCHAR *);
```

The second line extracts the next variable argument from the list and uses the variable "s" to act as that variable. At the same time, the variable argument list is advanced by the size of a `TCHAR` pointer.

The next portion of this `case` is a `do...while` loop which reads a `TCHAR` from the file and, if it's not `NULL`, appends it to the result s:

```
do
{
    ret = ReadFile(*fp, charbuf, sizeof(TCHAR),
&num, NULL);
```

```

        if(ret == FALSE)
        {
            break;
        }
        if(charbuf[0] != (TCHAR)'\0')
        {
            s[count] = charbuf[0];
            count++;
        }
    } while( (charbuf[0] != (TCHAR)'\n') && (charbuf[0]
!= (TCHAR)'\r') && (charbuf[0] != (TCHAR)' ') && (charbuf[0] !=
(TCHAR)'\0'));
    s[count] = (TCHAR)'\0';
    break;

```

You'll also notice here that the end of a string is defined by the `while` condition of the `do...while` loop, repeated here for clarity:

```

while( (charbuf[0] != (TCHAR)'\n') && (charbuf[0] != (TCHAR)'\r') &&
(charbuf[0] != (TCHAR)' ') && (charbuf[0] != (TCHAR)'\0'));

```

What this means is that "whitespace" for our `fscanf()` is defined as any character that is:

- '\n' a linefeed
- '\r' a carriage return
- ' ' a space
- '\0' a NULL

As you can see, all of the conditions in our `switch...case` statement follow the same guidelines for determining whitespace.

The next case condition tests for long integers and integers and treats them in the same manner:

```

case (TCHAR)'d':           // int
case (TCHAR)'l':           // int
    d = va_arg(ap, int *);

```

Here again, you are extracting the next item from the list of variable arguments and assigning it to the variable `d`, an integer. The variable argument list is then advanced by the size of an integer pointer. Unlike the string processing code,

however, this case condition will first attempt to read any leading blanks from the file before reading the numeric digits:

```
do
    {
        ret = ReadFile(*fp, charbuf, sizeof(TCHAR),
&num, NULL);
        if(ret == FALSE)
        {
            break;
        }
    } while(charbuf[0] == (TCHAR)' ');
```

Then, because you had to read one nonblank character (i.e., a numeric digit) in order to break out of the loop, you must save that extra character:

```
tmp[count] = charbuf[0];
count++;
```

Now that you've stripped out any leading blanks, continue to read digits until you encounter whitespace:

```
do
    {
        ret = ReadFile(*fp, charbuf, sizeof(TCHAR),
&num, NULL);
        if(ret == FALSE)
        {
            break;
        }
        if(charbuf[0] != (TCHAR)'\\0')
        {
            tmp[count] = charbuf[0];
            count++;
        }
    } while( (charbuf[0] != (TCHAR)'\\n') && (charbuf[0]
!= (TCHAR)'\\r') && (charbuf[0] != (TCHAR)' ') && (charbuf[0] !=
(TCHAR)'\\0'));
```

Finally, append a NULL to the end of the string of digits you've collected and convert it to an integer:

```
tmp[count] = (TCHAR)'\\0';
*d = _wtoi(tmp);
break;
```

Floating-point values are handled in much the same way, by grabbing the next argument off the variable argument list and reading the leading blanks:

```

        case (TCHAR)'f':          // float
            f = va_arg(ap, float *);
            do
            {
                ret = ReadFile(*fp, charbuf, sizeof(TCHAR),
&num, NULL);

                if(ret == FALSE)
                {
                    break;
                }
            } while(charbuf[0] == (TCHAR)' ');
            tmp[count] = charbuf[0];
            count++;

```

Then, read a digit at a time, until you encounter whitespace:

```

            do
            {
                ret = ReadFile(*fp, charbuf, sizeof(TCHAR),
&num, NULL);

                if(ret == FALSE)
                {
                    break;
                }
                if(charbuf[0] != (TCHAR)'\0')
                {
                    tmp[count] = charbuf[0];
                    count++;
                }
            } while( (charbuf[0] != (TCHAR)'\n') && (charbuf[0]
!= (TCHAR)'\r') && (charbuf[0] != (TCHAR)' ') && (charbuf[0] !=
(TCHAR)'\0'));

```

To finish off, append a NULL to the string of digits you've collected and convert it to a floating-point value using `swscanf()`:

```

            tmp[count] = (TCHAR)'\0';
            swscanf(tmp, TEXT("%F"), f);
            break;

```


NOTE

`swscanf()` works just like `fscanf()`, except that it reads and parses formatted data from a string, as opposed to a file.

The last time you will use `switch..case` is for single characters. Again, start by grabbing the next argument from the variable argument list:

```
case (TCHAR)'c':           // char
    c = va_arg(ap, TCHAR *);
```

Then, because you're only reading a single char, don't scan for whitespace or leading blanks. Instead, just read a single character from the file:

```
ret = ReadFile(*fp, charbuf, sizeof(TCHAR), &num,
NULL);
```

If the read is not successful for any reason, simply continue to the next iteration of the `while` loop:

```
if(ret == FALSE)
{
    continue;
}
```

Otherwise, assign the result of our parsing to the appropriate variable:

```
*c = charbuf[0];
```

Since that's the last of the data types you're checking for, close the `switch..case`, the `while` loop, and, finally, the function itself.

Before you're done, however, you must properly reset the variable argument list pointer:

```
va_end(ap);
```

And that's all there is to re-creating `fscanf()`.

Here is the code in its entirety:

```
int fscanf(FILE * fp, LPWSTR format, ...)
{
    TCHAR *c, *s;
    int *d;
    float *f;
    DWORD num = 0;
    TCHAR charbuf[2];
    TCHAR tmp[80];
```

```

int count = 0;
BOOL ret = FALSE;

va_list ap;
va_start(ap, format);
while (*format)
{
    count = 0;
    switch(*format++)
    {
        case (TCHAR)'s':           // string
            s = va_arg(ap, TCHAR *);
            do
            {
                ret = ReadFile(*fp, charbuf, sizeof(TCHAR),
&num, NULL);

                if(ret == FALSE)
                {
                    break;
                }
                if(charbuf[0] != (TCHAR)'\\0')
                {
                    s[count] = charbuf[0];
                    count++;
                }
            } while( (charbuf[0] != (TCHAR)'\\n') && (charbuf[0]
!= (TCHAR)'\\r') && (charbuf[0] != (TCHAR)' ') && (charbuf[0] !=
(TCHAR)'\\0'));
            s[count] = (TCHAR)'\\0';
            break;
        case (TCHAR)'d':           // int
        case (TCHAR)'}':           // int
            d = va_arg(ap, int *);
            do
            {
                ret = ReadFile(*fp, charbuf, sizeof(TCHAR),
&num, NULL);

                if(ret == FALSE)
                {
                    break;
                }
            } while(charbuf[0] == (TCHAR)' ');
            tmp[count] = charbuf[0];

```

```

count++;
do
{
    ret = ReadFile(*fp, charbuf, sizeof(TCHAR),
&num, NULL);
    if(ret == FALSE)
    {
        break;
    }
    if(charbuf[0] != (TCHAR)'\\0')
    {
        tmp[count] = charbuf[0];
        count++;
    }
} while( (charbuf[0] != (TCHAR)'\\n') && (charbuf[0]
!= (TCHAR)'\\r') && (charbuf[0] != (TCHAR)' ') && (charbuf[0] !=
(TCHAR)'\\0'));
tmp[count] = (TCHAR)'\\0';
*d = _wtoi(tmp);
break;
case (TCHAR)'f': // float
f = va_arg(ap, float *);
do
{
    ret = ReadFile(*fp, charbuf, sizeof(TCHAR),
&num, NULL);
    if(ret == FALSE)
    {
        break;
    }
} while(charbuf[0] == (TCHAR)' ');
tmp[count] = charbuf[0];
count++;
do
{
    ret = ReadFile(*fp, charbuf, sizeof(TCHAR),
&num, NULL);
    if(ret == FALSE)
    {
        break;
    }
}
if(charbuf[0] != (TCHAR)'\\0')
{

```

```

        tmp[count] = charbuf[0];
        count++;
    }
    } while( (charbuf[0] != (TCHAR)'\n') && (charbuf[0]
!= (TCHAR)'\r') && (charbuf[0] != (TCHAR)' ') && (charbuf[0] !=
(TCHAR)'\0'));
        tmp[count] = (TCHAR)'\0';
        swscanf(tmp, TEXT("%f"), f);
        break;
    case (TCHAR)'c':          // char
        c = va_arg(ap, TCHAR *);
        ret = ReadFile(*fp, charbuf, sizeof(TCHAR), &num,
NULL);
        if(ret == FALSE)
        {
            continue;
        }
        *c = charbuf[0];
        break;
    default:
        break;
    }
}
va_end(ap);
return(1);
}

```

Differences between Our fscanf() and the Original fscanf()

As you may have noticed, this is not a 100 percent perfect recreation of `fscanf()`. For instance, there are some format specifiers that this `fscanf()` does not understand:

- "%0x..." hex value specifiers
- "%03d" leading zero specifiers when reading integers into string variables
- Literals embedded in the format string

These limitations aside, however, it is practical for moderate use and can be easily extended or customized to meet specific requirements.

fseek()

The last function in our list of FILE*-based functions to re-create on CE is `fseek()`. `fseek()` is used to change the current position of the file. The original RTL `fseek()` is defined as follows:

```
int fseek( FILE *stream, long offset, int origin );
```

where `stream` is the file pointer, `offset` is the number of bytes to move (positive or negative value), and `origin` specifies the starting point. `origin` can be one of three values, predefined as constants:

- `SEEK_CUR` offset is relative to current position in file
- `SEEK_END` offset is relative to end of file
- `SEEK_SET` offset is relative to beginning of file

Your `fseek()` will follow this definition exactly. Here, however, you're really just translating these constants into constants used by the API's `SetFilePointer` function.

`SetFilePointer`—a member of the `CreateFile`-family of functions—does essentially the same thing as `fseek()` but requires rewriting existing code. `fseek()`, therefore, is easier to use.

First, map the constants from one function to the constants from another function, using a `switch...case` statement:

```
switch(origin)
{
    case SEEK_CUR:
    default:
        how = FILE_CURRENT;
        break;
    case SEEK_END:
        how = FILE_END;
        break;
    case SEEK_SET:
        how = FILE_BEGIN;
        break;
}
```

That done, call `SetFilePointer` and then check the result type for an error:

```
ret = SetFilePointer(fp, offset, NULL, how);
if(ret == 0xFFFFFFFF) // error
    return(1);
return(0);
```

And that's it!

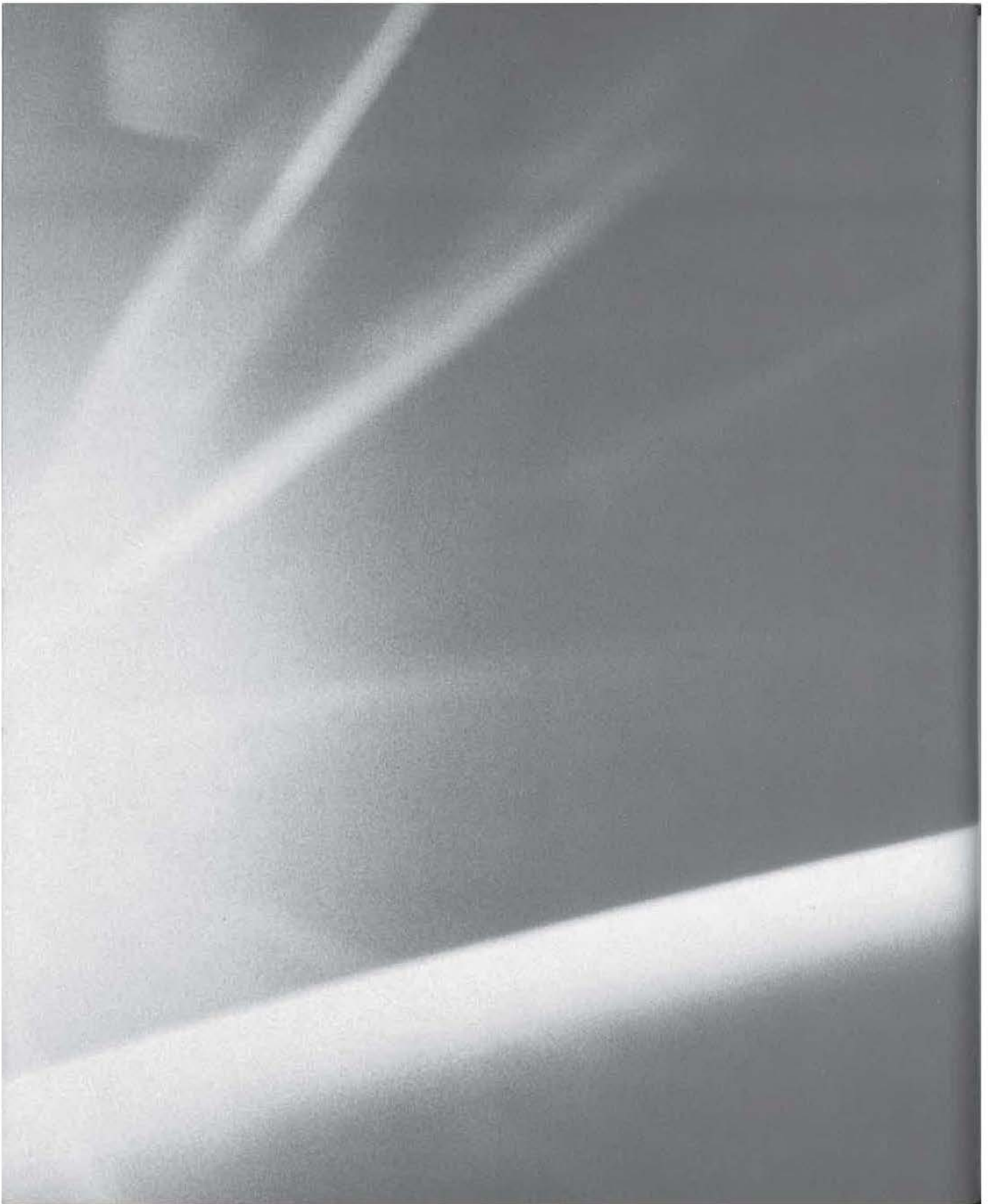
Here is the entire `fseek()` function, as you'll implement it:

```
int fseek( FILE *fp, long offset, int origin )
{
    DWORD how;
    DWORD ret;
    switch(origin)
    {
        case SEEK_CUR:
        default:
            how = FILE_CURRENT;
            break;
        case SEEK_END:
            how = FILE_END;
            break;
        case SEEK_SET:
            how = FILE_BEGIN;
            break;
    }
    ret = SetFilePointer(fp, offset, NULL, how);
    if(ret == 0xFFFFFFFF) // error
        return(1);
    return(0);
}
```

As you can see, it's fairly simple and straightforward. At this point, with a few functions, some API calls, and a clever `#define`, you have re-created the key `stdio.h` file-access functions.

Summary

In this chapter, we reviewed some of what's missing in the C/C++ language for Windows CE. You learned that, although some favorite constructs and functions are missing from Windows CE, there are ways to work around this. You can find substitute functions, change your program's logic, or re-create the missing functions yourself. In addition to reviewing some of the items that were removed, you also implemented your own FILE*-based functions!



CHAPTER

FOUR

4

CE's Structured Storage

- The Registry
- The Windows CE Database Engine
- CE's Database Engine vs. Familiar Database Engines
- The CE Database Engine API

Windows CE was designed to power information appliances and, when you think about it, an information appliance without a way to store information is pretty useless. That's why Microsoft provides two kinds of structured storage as part of the CE operating system itself. These two kinds of structured storage are

- The Windows CE registry
- The Windows CE Database Engine

The Windows CE registry is used for the same purpose as the registry in Windows 98/NT: to store user settings and small amounts of user data. As you'll see later in this chapter, there are a few precautions you must take when using the CE registry because, in contrast to the registry of Windows 98/NT, the CE registry has a reduced scope and purpose.

The registry as a form of structured storage has some characteristics in common with a database system, but it really isn't a true database system. Instead, it is a hierarchical data store that allows us to immediately access any specific piece of information. In addition, it provides the functionality for us to search the entire data store for a value. But, as you'll also see, that's about all you can do with the registry.

Windows CE provides an additional, better form of structured storage: it comes with its own database engine. However, the CE Database Engine is significantly different from most of the database engines we're used to. The tables created with the CE Database Engine are closer to text files than to relational structures used by most database systems. Further, the CE Database Engine does not support SQL and does not come with the usual array of database management software. Naturally, these factors will impact your choices when it comes to storing and organizing your data.

Despite its limitations, though, the CE Database Engine does have enough functionality to be very useful for light database work. For instance, just as with the registry, you can search a CE database for a specific value. Likewise, you can immediately access any specific piece of data. Furthermore, the CE Database Engine even allows you to use indexes (called *sort orders*) to order the records of a table. On the whole, the CE databases provide a convenient, no-frills, low-effort, zero-cost way to store and organize your data.

In this chapter, we'll examine the trade-offs of using the registry, when to use it, and when not to use it. In addition, we'll look at when to use the registry vs. when to use the Database Engine. Then we'll examine the trade-offs of using the

Database Engine, when to use it, and when not to use it. We'll finish up with a sample application that shows off the features of the CE Database Engine.

The Registry

In this section, we'll be examining the basics of working with the registry under Windows CE. Data in the registry is organized in a hierarchical fashion, with four major roots, called keys, as outlined in Table 4.1.

TABLE 4.1: The Roots of the Windows CE Registry Hierarchy

Key Name	Type of Information Stored under Key
HKEY_LOCAL_MACHINE	Hardware information and program settings that apply to the machine as a whole
HKEY_CURRENT_USER	Program settings relating to the preferences of the currently logged-in user
HKEY_USERS	Program settings stored by user name
HKEY_CLASSES_ROOT	ActiveX classes and file types registered with the system

Each of these roots or keys has an unlimited number of branches called subkeys. Each of these subkeys can then have subkeys of their own, and so on, and so on. It is these keys that form the hierarchical structure of the registry.

HKEY_CURRENT_USER and HKEY_USERS **Under Windows CE**

Traditionally, under Windows 98/NT, the HKEY_USERS key retains the settings for all of the users, organized by user name. So, when a user logs into the system, their individual settings are loaded into the HKEY_CURRENT_USER key. Because Windows CE doesn't support multiple users, the HKEY_USERS key exists largely for the purpose of backward compatibility with Desktop Windows applications ported to the CE platform. You can continue to use both the HKEY_USERS and HKEY_CURRENT_USER keys just as you would under 98/NT; however, only HKEY_CURRENT_USER will have any real use under CE.

Proper Uses of the CE Registry

Under Windows CE, there are really only two main uses of the registry:

- Storing system data
- Storing user-specific information

Storing System Data

The operating system itself maintains certain system data in the registry. Typically, this consists of file associations, properties of ActiveX controls, locale settings, owner information, control panel settings, and other information necessary to maintain the state of the system.

Acceptable kinds of information you might write to the registry include your application-specific file associations, persistent state information, and user option settings. Other types of data written to the registry may include COM object registration (i.e., ActiveX controls and input methods) and references to other installed application or data files.

Storing User-Specific Information

The second acceptable use of the registry is to store user-specific information such as color preferences, MRU (most recently used) file lists, default file locations, etc. For example, Microsoft's Pocket Internet Explorer stores its user-specific information under the key `HKEY_CLASSES_ROOT\Software\Apps\PocketIE`. If you examine the various settings found under this key, you'll find ones like the default search engine, the proxy server address, and even the maximum cache size. The point is that even Microsoft only stores the bare minimum information needed.

Improper Uses of the CE Registry

The basic rule to follow when deciding whether or not to write a specific piece of data out to the CE registry is: "Is this absolutely critical and necessary to the operation of our program?" If you can't answer "yes" to that question, you shouldn't be writing that data to the registry at all. There are two reasons for this:

- Limited storage space
- Decreased system performance

The issue of limited storage space is obvious. CE devices don't have a lot of memory, so you must be conservative in your use of what memory is available. Not quite as obvious is the issue of decreased system performance. This is because every application, including the OS itself, needs to access the registry on an almost constant basis. For example, the simple act of minimizing a window under Windows 98/NT generates *at least six reads* from the registry! The problem here is that if you load the registry down with a lot of unnecessary data, it takes longer to access any specific piece of data. Simply put, if you add lots of data to the registry, its overall size increases. If its overall size increases, it takes longer to either search the registry or to access a specific piece of data. The more bloated the registry becomes, the longer it takes to access the information you're after because the registry-related functions have to wade through data you don't care about.

Unless you feel that there's a compelling reason to do so, there are several types of information that you should avoid writing to the registry:

- Application version information
- BLOB data of any kind
- Redundant information that can be dynamically generated

MFC, Palm-size PCs, and the Registry

One of the problems with MFC on a Palm-size PC is that the usual MFC classes that are used to work with the registry are Active Template Library (ATL) classes. Unfortunately, the ATL is not supported on PPC devices. Therefore, it's probably best to use straight API-based functions for working with the CE registry under MFC. This will ensure that your code will compile for as many platforms as possible. On the CD for this book, there is a sample password application that demonstrates how to use the registry API from within an MFC application. Basically, the application prevents anyone from doing anything on the device until the correct password is entered. The user can also change the password if desired. As you can guess, the application uses the registry to store and retrieve the password.

Application Version Information

Typically, under Windows 98/NT, when an application is installed, it writes some application version information to the registry. For instance, when Windows itself is installed or upgraded, Microsoft writes the version number to the registry under the key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Version`. While this is fine to do on a Desktop machine, it would be considered very poor design to write such information to the CE registry because of the limited storage space of a CE device.

Under Windows 98/NT, storage space is theoretically not an issue, so the extra few bytes or so that it takes to write this setting to the registry don't count a great deal. However, under CE if every application vendor wrote such information to the registry, your free space would rapidly disappear. A quick review of the CE registry reveals that almost no vendors at all write this kind of information to the registry. Of course, there are situations in which your application may legitimately need to store this type of data; however, it is best to be as conservative as possible.

BLOB Data of Any Kind

BLOB (Binary Large Object) data is free-form binary data of a certain length that can represent anything from the user's address to a bitmap image. Under CE, just as with Windows 98/NT, it's possible to store BLOB data in the registry. However, it's not necessarily a good idea. The main issue here is, once again, storage space. As a developer, you need to ensure that you do not take up too much storage space, especially in the registry, which is an area that most users do not have the tools or skills to edit directly.

Redundant Information

Information in this category is anything that can be calculated or otherwise retrieved from another source instead of being stored. For instance, you wouldn't want to store the time stamp of a file in the registry because you can always retrieve that information by querying the file itself.

The Registry vs. the Database Engine

When it comes to choosing the registry over the database engine, the main issue is storage space. While it's true that there are some uses for which a database is naturally more suited, there is another issue that makes databases very attractive as a means of structured storage. By default, all databases are compressed. That means that even if you store the same number of characters to the registry and a database, you'll be saving the user some storage space simply by choosing the database method.

The Windows CE Database Engine

In this section, we'll be looking at the CE database engine and what it can and can't do. First, we'll look at the differences and similarities between the CE database engine and other database engines. After that, we'll move on to a sample application.

Differences between CE's Database Engine and Familiar Database Engines

When it comes to CE's database engine, there are a few points you need to know up front. Typically, the most challenging part of getting started with a CE database is getting used to the differences between the CE database engine and other database engines.

These differences come in two forms:

- The engine's organization and naming of data
- Features offered by the engine itself

Organization and Naming of Data

There are three important facts about the organization and naming of data by the CE database engine:

- CE's "databases" are really just tables.
- CE's database engine is not relational.
- Fields are record-specific and not table-specific.

CE's Databases Are Really Just Tables The term *database* is misused under Windows CE. That's because a CE database is what most of us would call a table—an object containing columns and rows of data. Typically, of course, a database is a *collection* of related tables, each table having columns and rows.

CE's Database Engine Is Not Relational The CE Database Engine is not a relational database system. There are no foreign keys or master-detail relationships that the engine will create for us. Instead, it is a system closer in logical appearance to an early version of the DBF file format; that is, a CE database is basically a well-organized text file.

Fields Are Record-Specific and Not Table-Specific The fields of a CE database are record-specific and not table-specific. In other words, it's entirely possible for one record to have three fields and the next record to have five fields. Fields, in fact, are considered to be a record's properties, undefined until the record's contents are actually written out to the database. Consider a standard table of the type you're used to:

ID	NAME	COMPUTER TYPE	DEPARTMENT
34	Jane S.	PII	Engineering
9A	Joe R.	486	Sales
44	Ted W.	586	Sales

A CE Table, on the other hand, might look something like this:

Record 1:

ID	NAME	COMPUTER TYPE	DEPARTMENT
34	Jane S.	PII	Engineering

Record 2:

ID	NAME	COMPUTER TYPE	DEPARTMENT	SALARY	PICTURE
9A	Joe R.	486	Sales	\$44,000.00	(BLOB Data)

Record 3:

ID	NAME	COMPUTER TYPE	DEPARTMENT	SALARY
44	Ted W.	586	Sales	\$42,500.00

Record 4:

ID	NAME	COMPUTER TYPE	DEPARTMENT
Z7	Betty Y.	K-6	Engineering

Differences in Features Offered by the Engine Itself

The features offered by the CE Database Engine itself make up the additional differences between the CE Database Engine and more traditional database engines. For instance, CE's Database Engine does not support SQL (Structured Query Language) queries. This means that you must use the API and your own code in order to perform searches of a database.

Similarly, the CE Database Engine does not offer the usual package of management tools and utilities that most engines do. Most database engines, for instance,

come with some kind of Enterprise Manager application, which provides a graphical way to manage your databases and tables.

Now that you've seen all of the differences between CE's Database Engine and other engines, let's take a look at the similarities.

Similarities between CE's Database Engine and Familiar Database Engines

Despite the differences between CE's Database Engine and other database engines, there are a number of similarities. For instance, CE's Database Engine allows you to do two of the most common database operations:

- Indexing
- Searching

Even when you're performing these more common database operations, there are still some unusual aspects you need to deal with before diving in.

Indexing

Just as most database engines do, CE's Database Engine allows you to create indexes (or "sort orders") for your tables. While these indexes/sort orders allow you to sort the records of a CE database, they do have a few limitations. Before you can understand these limitations and how they'll affect you, however, you need to first understand how databases are created and then opened.

To create a CE database, call `CeCreateDatabase()`. `CeCreateDatabase()` is defined as follows:

```
CEOID CeCreateDatabase(LPWSTR lpszName, DWORD dwDbType, WORD wNum-  
SortOrder, SORTORDERSPEC * rgSortSpecs);
```

The first parameter, `lpszName`, is simply the name you're going to assign to the database once it's created.

The second parameter, `dwDbType`, is a numeric value that your application uses to uniquely identify one of its databases from another.

NOTE

The `dwDbType` value does not have to be globally unique; only your application uses this value.

The third parameter, `wNumSortOrder`, tells the CE Database Engine how many sort orders you'll be creating.

The fourth parameter, `rgSortSpecs`, is a pointer to an array of `SORTORDERSPEC` structures, which serve to define the actual sort orders. The `SORTORDERSPEC` structure has two members:

```
PEGPROPID propid;
DWORD dwFlags;
```

`PEGPROPID` is a long integer, composed of two `WORD`s. The low-order `WORD` specifies the field type using one of the eight possible constants defined by Microsoft, as outlined in Table 4.2.

TABLE 4.2: Field Types of the CE Database Engine

Field Type	Description
<code>CEVT_LPWSTR</code>	The field is a Unicode string, NULL terminated.
<code>CEVT_FILETIME</code>	The field is a <code>FILETIME</code> type.
<code>CEVT_I2</code>	The field is a 16-bit signed int type.
<code>CEVT_I4</code>	The field is a 32-bit signed int type.
<code>CEVT_UI2</code>	The field is a 16-bit unsigned int type.
<code>CEVT_UI4</code>	The field is a 32-bit unsigned int type.
<code>CEVT_R8</code>	The field is a double (floating point) type.
<code>CEVT_BOOL</code>	The field is a Boolean type.
<code>CEVT_BLOB</code>	The field is a BLOB type.

WARNING

The `CEVT_BLOB` type, which is included here for completeness, cannot be indexed. All other field types can be indexed.

WARNING

The CEVT_R8 and CEVT_BOOL types are new to CE 2.11 and are not supported in previous versions. To store a floating point value under CE 2.0x, you must convert the value to a string and store it that way. Similarly, to store a Boolean value under CE 2.0x, you should probably convert it to an integer and store it that way.

The high-order WORD is an application-defined value. It is a numeric value used by the application to uniquely distinguish one field from another. Just as with the `dwDbType` parameter of `CeCreateDatabase()`, the high-order WORD here is only intended to be unique within the scope of the application. In other words, it's perfectly acceptable to have one application that used the value `100` for the first field of its database and a second application that also used `100` for the first field of its database.

The second part of the `SORTORDERSPEC` structure is the `dwFlags` parameter, which is a bitmask value specifying the way in which the records should be ordered. A complete listing of the possible values and their descriptions is shown in Table 4.3. As indicated in the table, these values can be combined to further customize the sort.

TABLE 4.3: Sort Types Available

Value	Description
<code>CEDB_SORT_GENERICORDER</code>	The default setting, wherein records are sorted in ascending order. (The value <code>0</code> will appear before <code>9</code> .) You cannot use the OR operator on this value and <code>CEDB_SORT_DESCENDING</code> because they conflict with each other.
<code>CEDB_SORT_DESCENDING</code>	The records are sorted in descending order. (The value <code>9</code> will appear before <code>0</code> .) You cannot use the OR operator on this value and <code>CEDB_SORT_GENERICORDER</code> because they conflict with each other.
<code>CEDB_SORT_CASEINSENSITIVE</code>	If the field being sorted is a character string, the records are sorted without regard to case. (The value <code>a</code> has the same rank as the value <code>A</code> .) You can use the OR operator on this value together with any of the other flags.
<code>CEDB_SORT_UNKNOWNFIRST</code>	This flag exists to handle the condition of records that do not contain the field being sorted. As you saw earlier, it's entirely possible for some records to possess a field that is not shared by other records. By default, records that do not possess the field being sorted will be sorted to appear at the end of the database. However, if you use <code>CEDB_SORT_UNKNOWNFIRST</code> , these records will appear at the beginning of the database. You can safely use the OR operator on this value together with any of the other flags.

NOTE The CE0ID (CE Object Identifier) return type of `CeCreateDatabase()` is another globally unique value. CE0IDs are like ID numbers given to all CE system objects, such as files and databases, which CE uses to keep track of them. Even records of a CE database get their own CE0ID number.

Now that you know everything that goes into creating a database, let's look at some code that actually does the work of creating a database. This example is from an application we'll be developing throughout the rest of this chapter. Here, we'll create a database to hold a list of area codes and the states they correspond to.

NOTE In the source code for this project, each database-related operation is encapsulated in its own function, named for the operation being performed. For instance, the database creation code you're looking at now is contained in a function called `AreaCodeDatabaseCreate()`.

Your first task when creating a database is to choose the unique value to identify this database type; this will be used for the `dwDbaseType` parameter of `CeCreateDatabase()`:

```
#define DB_ID      1911
```

You must also define a constant for the name of the database:

```
#define AREACODEDBNAME TEXT("ACDB")
```

Next, choose values to identify the field types; these values will be used first in your `SORTORDERSPEC` array:

```
#define PROP_AREACODE  100
#define PROP_STATE     200
```

TIP It's best to define the database type identifier, database name, and field identifiers as constants because you'll be using them throughout your application.

The next step is to create and then fill in the values for your `SORTORDERSPEC` array. Your array will be two `SORTORDERSPECs` wide, one sort order for the area code field and one sort order for the state field:

```
SORTORDERSPEC soSortOrders[2];
```

Then fill in the values of the SORTORDERSPEC structures, starting with the SORTORDERSPEC for the area code field, then the state field:

```
soSortOrders[0].propid = MAKELONG(CEVT_LPWSTR, PROP_AREACODE);
soSortOrders[0].dwFlags= CEDB_SORT_GENERICORDER;
soSortOrders[1].propid = MAKELONG(CEVT_LPWSTR, PROP_STATE);
soSortOrders[1].dwFlags= CEDB_SORT_GENERICORDER;
```

Now that you've prepared everything, call CeCreateDatabase(), passing in the database name, the database type ID, the number of sort orders, and a pointer to the sort order array:

```
m_obj = CeCreateDatabase(AREACODEDBNAME, DB_ID, 2, &soSortOrders);
```

You can then test the result and perform some action based on the value of m_obj. In this case, you'll be returning a TRUE or FALSE result from a function:

```
if (!m_obj) //a NULL result from CeCreateDatabase indicates failure
{
    return(FALSE);
}
else
{
    return(TRUE);
}
}
```

When you're all done, your completed AreaCodeDatabaseCreate() function looks like this:

```
BOOL AreaCodeDatabaseCreate()
{
    //fill in the properties of our sort orders before we create the
    database
    soSortOrders[0].propid = MAKELONG(CEVT_LPWSTR, PROP_AREACODE);
    soSortOrders[0].dwFlags= CEDB_SORT_GENERICORDER; //Ascending
    (default) order
    soSortOrders[1].propid = MAKELONG(CEVT_LPWSTR, PROP_STATE);
    soSortOrders[1].dwFlags= CEDB_SORT_GENERICORDER; //Ascending
    (default) order

    m_obj = CeCreateDatabase(AREACODEDBNAME, DB_ID, 2, &soSortOrders);

    if (!m_obj) //a NULL result from CeCreateDatabase indicates failure
    {
```

```

        return(FALSE);
    }
    else
    {
        return(TRUE);
    }
}

```

Let's now proceed to the `AreaCodeDatabaseOpen()` function. To open a database, use the API function `CeOpenDatabase()`, which is defined as follows:

```

HANDLE CeOpenDatabase(PCEOID poId, LPWSTR lpszName, CEPROPID propId,
    DWORD dwFlags, HWND hwndNotify);

```

The first parameter, `poId`, is a pointer to a CEOID. Since you probably won't know the CEOID most of the time (you'll probably only know the database name) you can set this value to 0. For example:

```

CEOID oid = 0;
CeOpenDatabase(&oid, ...);

```

CE's Database Engine will then write the value of the database's CEOID to the variable passed in as `poId`.

WARNING

Because CE's Database Engine will actually be writing a value to the `poId` parameter, you should never pass 0 as a constant value.

The second parameter, `lpszName`, is the name of the database that you assigned when you created the database. In the case of your area code application, use the same value you used before, `AREACODEDBNAME`.

The third parameter, `propId`, specifies the sort order you'd like to use. It must be either 0 (for no sort order) or a long integer, just like the `propId` value of your `SORTORDERSPECS`. Therefore, your value for the `propId` parameter will be `MAKELONG(CEVT_LPWSTR, PROP_AREACODE)`.

The fourth parameter, `dwFlags`, specifies what, if anything, will be done to the current record pointer each time you read a record. The choices are

- `CEDB_AUTOINCREMENT`
- 0 (no operation)

With the `CEDB_AUTOINCREMENT` value, the CE Database Engine will automatically set the current record pointer to point to the next record in the database each time you read a record. With the value of 0, the CE Database Engine won't do anything to the current record pointer. For your purposes, the auto-increment option will be just fine.

The fifth parameter, `hwndNotify`, is used if you want to receive notification messages when another program is accessing your database at the same time you are accessing it. You don't need this for your application, so pass in a `NULL` value.

TIP

The `hwndNotify` parameter can be used to set up some kind of crude record-locking mechanisms so that the two programs accessing the same data will not overwrite each other's data.

Your actual call to `CeOpenDatabase()`, then, looks like this:

```
m_Handle = CeOpenDatabase(&oid, AREACODEDBNAME,
    MAKELONG(CEVT_LPWSTR, PROP_AREACODE), CEDB_AUTOINCREMENT, NULL);
```

And your entire `AreaCodeDatabase()` function looks like this:

```
BOOL AreaCodeDatabaseOpen()
{
    CEOID oid = 0;
    CEPROPID propid = 0;
    HWND hwndNotify = NULL;
    m_Handle = CeOpenDatabase(&oid, AREACODEDBNAME, MAKELONG(CEVT_LPWSTR,
PROP_AREACODE), CEDB_AUTOINCREMENT, NULL);
    if(m_Handle == INVALID_HANDLE_VALUE)
    {
        return(FALSE);
    }
    else
    {
        return(TRUE);
    }
}
```

Now that you have a pretty good background on what goes into creating and opening a database and how sort orders are created and used, we can return to our original discussion of database sort orders. As you just learned, these sort

orders have a few limitations that may affect how you use them. These limitations are:

- A sort order can only sort records based on one of the database's fields.
- Only one sort order per database can be marked as active.
- A database can have a maximum of four sort orders.
- The sort orders should be created when the table is created.

Let's take a look at these limitations and how to ensure that these sort orders will still work for you.

A Sort Order Can Only Sort Records Based on One of the Database's Fields Typically, with other database engines it would be possible to create an index that ordered records based on multiple fields. However, as you've just seen, you are limited to a *one-field-per-sort-order* rule when you create a sort order.

Only One Sort Order per Database Can Be Marked as Active Unlike ordinary database engines, where an index exists more for the purpose of maintaining a certain relationship between records, CE's sort orders are analogous to a SQL "view" that displays the records of the database in the order specified. Sort orders do not change the physical order of the records in any way. The catch to this is that you can only use one sort order at a time. If you want to change sort orders, you must close the database and then reopen it, specifying a different sort order.

With a CE database, each time you open the database, you have the opportunity to select one of your sort orders (or no sort order at all). One way to think about sort orders is that they are comparable to the columns in a ListView control in the Report View style (pictured in Figure 4.1). By clicking on any of the column headings, you can sort the data according to the values in that column. In much the same way, by selecting one of the sort orders, you will sort the data according to the data in the column that the sort order points to.

FIGURE 4.1:

A ListView control,
Report View style

Name	Size	Type	Modified	Attributes
WMIPSDbg		File Folder	1/29/99 6:37 PM	
WMIPSRel		File Folder	2/2/99 12:07 AM	
Acdb.001	21KB	001 File	12/29/98 9:32 PM	A
Acdb.apa	36KB	APS File	2/2/99 12:07 AM	A
ACDB.c	13KB	C File	1/29/99 4:32 AM	A
ACDB.dsp	22KB	DSP File	1/29/99 6:37 PM	A
ACDB.dsw	1KB	DSW File	1/29/99 6:37 PM	A
ACDB.exe	19KB	Application	1/29/99 4:43 AM	A
ACDB.ncb	73KB	NCB File	2/1/99 4:47 PM	A
ACDB.opt	54KB	DPT File	2/2/99 12:07 AM	A
ACDB.plg	2KB	PLG File	2/2/99 12:07 AM	A
ACDB.rc	4KB	RC File	2/2/99 12:07 AM	A
areacode.txt	6KB	Text Docu...	1/23/99 3:37 PM	A
db.c	16KB	C File	1/29/99 4:38 AM	A
db.h	3KB	H File	1/29/99 3:43 AM	A
icon1.ico	1KB	Icon	12/1/98 5:54 PM	A
resource.h	2KB	H File	1/29/99 3:27 AM	A

TIP

If you don't select a sort order, the records will appear in the order in which they were entered into the database.

A Database Can Have a Maximum of Four Sort Orders An additional limitation of sort orders is that you can only create a maximum of four indexes per table. This probably won't affect most of your applications, but it's worth knowing about before you get too far into your application.

The Sort Orders Should Be Created When the Table Is Created The final limitation of the CE sort orders is that it is best to create them when the table is created. This is because the CE Database Engine operates more efficiently when building the indexes as the records are entered, as opposed to creating an index after the database has been populated with records.

In version 2.11 of Windows CE, however, Microsoft did add a feature that makes it possible to add and delete indexes after you've created and populated the table. As is usually the case, there are trade-offs to this option. First, there is the fact that it only works on CE version 2.11, which limits your code to HPC/Pro devices (or upgraded HPCs). The second trade-off is that building new sort orders is a very resource-intensive task that can take several minutes to complete. Therefore, it's always best to create the sort orders at the same time you create the table.

Searching

Despite its lack of support for SQL, CE's database engine does allow you to search the databases. The catch here is that the engine does not allow wildcard matching as SQL does. Instead, the CE Database Engine provides one function, `CeSeekDatabase()`, which serves as a way to both search the database and manipulate the current record pointer. `CeSeekDatabase()` is defined as follows:

```
CEOID CeSeekDatabase(HANDLE hDatabase, DWORD dwSeekType, DWORD dwValue, LPDWORD lpdwIndex);
```

The first parameter, `hDatabase`, is a handle to an open database.

TIP

The second parameter, `dwSeekType`, specifies the type of seek or search operation you want to perform. In this way, you tell the CE Database Engine whether you're searching for a value or merely moving the current record pointer. When you do a search, a sort order must be active. That's because CE's database engine only searches on the sort order field. For the purpose of this discussion, *searching* means that you are attempting to find a value in the database matching the one supplied; *seeking* means simply moving the current record pointer to some relative or absolute position in the database.

The third parameter, `dwValue`, specifies the value you're searching for or is a numeric value specifying how the current record pointer should be moved.

The fourth parameter, `lpdwIndex`, is a pointer to a `WORD` that the CE Database Engine will fill in with the number of records from the start of the database to the record that was found.

There are eight different types of search or seek operations you can perform. When searching, you can look for a record that has a

- Certain CEID
- Value smaller than the value you're searching for
- Value greater than the value you're searching for
- Value equal to the value you're searching for

When seeking, you can move the current record pointer to the record that is

- The next record having a value equal to the value you just searched for (used with the last search operation above)
- A given number of records from the first record in the database
- A given number of records from the last record in the database
- A given number of records from the current record

A Record That Has a Certain CEOID This is a search operation that attempts to find a record that has a CEOID matching the one you specify in the `dwValue` parameter. To perform this type of search, pass in the value `CEDB_SEEK_CEOID` to `CeSeekDatabase()`:

```
CeSeekDatabase(m_Handle, CEDB_SEEK_CEOID, oidValue, &dwIndex);
```

While this search is very efficient, it's unlikely that you'll know the CEOID for your records. Therefore, you probably won't be performing this type of search very often.

A Record That Has Smaller Value than the Value You're Searching For

This is a search operation that finds the largest value that is smaller than the search value. So, if you had a database with the following records:

```
ID
42
43
47
56
```

and you were searching for the value 44 and specifying the `CEDB_SEEK_VALUE-SMALLER` flag, the result of your search would point to the record with the ID value of 43. The code to perform this search looks like this:

```
CeSeekDatabase(m_Handle, CEDB_SEEK_VALUESMALLER, 44, &dwIndex);
```

A Record That Has a Value Greater than the Value You're Searching For

This is a search operation that finds the smallest value that is greater than the search value. So, if you had a database with these records:

ID
42
43
47
56

and you were searching for the value 44 and specifying the `CEDB_SEEK_VALUEGREATER` flag, the result of your search would point to the record with the ID value of 47. The code to perform this search looks like this:

```
CeSeekDatabase(m_Handle, CEDB_SEEK_VALUEGREATER, 43, &dwIndex);
```

A Record That Has a Value Equal to the Value You're Searching For

This is a search operation that finds the first value that's equal to the search value. So, if you had a database with these records:

ID
42
43
47
56

and you were searching for the value 43 and specifying the `CEDB_SEEK_VALUEFIRSTEQUAL` flag, the result of your search would point to the record with the ID value of 43. Similarly, if you were searching for the value 44 or some other value that doesn't exist in the table, the result of your search would point to the end of the table. The code to perform this kind of search looks like this:

```
CeSeekDatabase(m_Handle, CEDB_SEEK_VALUEFIRSTEQUAL, 43, &dwIndex);
```

The Record That Is the Next Record Equal to the Search Criteria This is a seek operation that is used in conjunction with the search operation you just looked at. Simply put, if you have a database with these records:

```
ID
42
43
43
57
```

and if you just searched for the value 43 specifying the `CEDB_SEEK_VALUEFIRST-EQUAL` flag, the result of your search would point to the first record with the ID value of 43. If you then called `CeSeekDatabase()`, specifying the `CEDB_SEEK_VALUENEXTEQUAL` flag, your database cursor would be positioned at the second record with the ID value of 43. The code to perform this type of operation looks like this:

```
CeSeekDatabase(m_Handle, CEDB_SEEK_VALUENEXTEQUAL, 43, &dwIndex);
```

The Record That Is a Given Number of Records from the Beginning of the Database This seek operation positions the current record pointer to a record that is `dwValue` records from the first record in the database. One use of this function is to position the cursor to the beginning of the database by specifying 0 as the `dwValue` parameter. The code to perform this type of operation looks like this:

```
CeSeekDatabase(m_Handle, CEDB_SEEK_BEGINNING, 0, &dwIndex);
```

The Record That Is a Given Number of Records from the End of the Database This seek operation positions the current record pointer to a record that is `dwValue` records from the last record in the database. One use of this function is to position the cursor at the last record of the database by specifying 0 as the `dwValue` parameter. The code to perform this type of operation looks like this:

```
CeSeekDatabase(m_Handle, CEDB_SEEK_END, 0, &dwIndex);
```

The Record That Is a Given Number of Records from the Current Record This seek operation positions the current record pointer to a record that is `dwValue` records from the current record in the database. For example, the code to advance the cursor by one record looks something like this:

```
CeSeekDatabase(m_Handle, CEDB_SEEK_CURRENT, 1, &dwIndex);
```

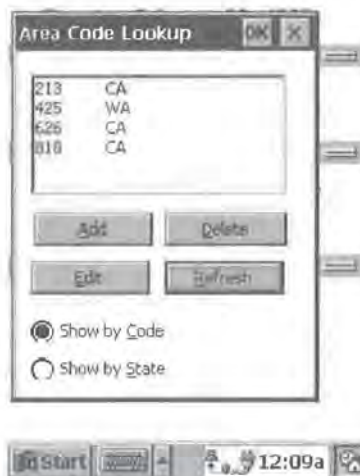
The CE Database Engine API

Although we've previewed it a little bit, let's now concentrate on creating a database application that will show off some of the lessons you've just learned. The application you'll create will be an area code lookup utility, which will enable the user to either:

- Find an area code if they know the state *or*
- Find a state if they know the area code

For the sake of simplicity, we'll make it a dialog-based application. The final dialog will look something like Figure 4.2.

FIGURE 4.3:
The Edit/Add Record dialog



The idea here is that the large list box will display the area codes and states. The user will have the option via two radio buttons to sort the list by area code or by state. They'll also be able to add or delete records using the buttons along the bottom of the form. When the user chooses to add a record, a separate dialog, shown in Figure 4.3, will appear, prompting them for an area code and a state.

FIGURE 4.3:

The Edit/Add Record dialog



Your first task will be to create a library of database-related functions, as you'll be performing many of these operations several times throughout the running of the program. You'll create a wrapper function for each one of the database operations your application will perform.

NOTE

We'll examine only the highlights of the application's code here. The full code is provided on the CD-ROM, of course.

At this point, the only routines you still need to develop are

- `AreaCodeDatabaseRead()` for reading records from the database
- `AreaCodeDatabaseInsert()` for adding records to the database
- `AreaCodeDatabaseDeleteRecord()` for deleting records from the database
- `AreaCodeDatabaseFinish()` for performing any cleanup operations

AreaCodeDatabaseRead()

Reading a record from a CE database is accomplished via `CeReadRecordProps()`, which is defined as follows:

```
CEOID CeReadRecordProps(HANDLE hDbase, DWORD dwFlags, LPWORD lpcPropID,
    CEPROPID * rgPropID, LPBYTE * lplpBuffer, LPDWORD lpcbBuffer);
```

The first parameter, `hDbase`, is a handle to an open database.

The second parameter, `dwFlags`, tells the CE Database Engine whether or not it can reallocate memory if the memory pointed to by the fifth parameter is too small. To allow the database engine to reallocate the memory, pass in `CEDB_ALLOWREALLOC`; to insist that it *not* reallocate the memory, pass in 0.

WARNING

Although it's possible to pass in 0 for this parameter, you'll almost always want to let the system reallocate memory as needed. Otherwise, your function call could fail, and you'll have to reallocate the buffer yourself.

The third parameter, `lpcPropID`, is a pointer to a long integer specifying the number of properties to be read. If you set this value to `NULL`, the variable you pass as `lpcPropID` will receive an integer value specifying the number of properties read. Recall that database fields are referred to as *properties* of a record.

The fourth parameter, `rgPropID`, is a pointer to an array of `CEPROPID` structures specifying which properties are to be read. If you set this parameter to `NULL`, all properties will be read.

NOTE

This is the same type of structure used to specify the active sort order when calling `CeOpenDatabase()`.

The fifth parameter, `lplpBuffer`, is a pointer to a pointer to an array of `CEPROPVAL` structures. This is where the properties you requested will actually be returned. `CEPROPVAL` is a somewhat complicated structure that has four members:

- `CEPROPID propid`
- `WORD wLenData`
- `WORD wFlags`
- `CEVALUNION val`

The `propId` member is just like all the other `CEPROPIDs` you've seen in this chapter: a long integer comprised of two `WORDs` that tell you about the field you're looking at. `wLenData` is officially unused, and `wFlags` will only be one of two values:

- `CEDB_PROPNOTFOUND` if you're reading a record and the requested property doesn't exist
- `CEDB_PROPDELETE` if you're removing a property from a record

That leaves just one member of the `CEPROPVAL` structure: `val`, a `CEVALUNION`, which is a union type that changes its contents based on the type of the field itself. This is a bit confusing at first, but it will become clear as soon as you begin to look at the code.

The final parameter of `CeReadRecordProps()` is `lpcbBuffer`, which is a pointer to an integer that will receive the size of the buffer that is the fourth parameter.

To look at `AreaCodeDatabaseRead()` from a coding perspective, what you have is a `WORD` used to store the number of properties read, a `CEPROPVAL` pointer (your buffer), and the number of bytes in the reallocated buffer:

```
BOOL AreaCodeDatabaseRead()
{
    WORD cProps;

    CEPROPVAL * pBuf;

    DWORD cbBuf;
```

Then, you call `CeReadRecordProps()`:

```
    if (!CeReadRecordProps(m_Handle, // handle to the database
                          CEDB_ALLOWREALLOC, // allow pBuf allocation
                          &cProps, // return count of properties
                          NULL, // retrieve all properties
                          (LPBYTE *)&pBuf, // buffer to return prop data
                          &cbBuf)) // count of bytes in pBuf
```

Then, you return a `BOOL` value based on the result of the call:

```
{
    return FALSE;
}
```

```

        else
        {
            return TRUE;
        }
    }
}

```

NOTE

You may not have encountered an error, even if `CeReadRecordProps()` returns a 0. It's up to the calling procedure to check `GetLastError()` to determine whether an error occurred or whether you merely reached the end of the database.

AreaCodeDatabaseInsert()

Your next task is inserting or adding records to your database. The function that does this is called `CeWriteRecordProps()` and is defined as follows:

```

CEOID CeWriteRecordProps(HANDLE hDbase, CEOID oidRecord, WORD cPropID,
    CEPROPVAL * rgPropVal);

```

The first parameter, `hDbase`, is a handle to an open database. The second parameter, `oidRecord`, is the CEOID of an existing record. If this parameter contains a value, the record being written will actually overwrite the existing record. If this parameter is 0, the record being written will be added as a new record. The third parameter, `cPropID`, is the number of properties you're writing. Finally, the `rgPropVal` parameter is a pointer to an array of `CEPROPVAL` structures, just as you saw above with `CeReadRecordProps()`.

In order to write a new record to your database, then, you need a function like this:

```

BOOL AreaCodeDatabaseInsert(LPTSTR szCode, LPTSTR szState)
{
    CEPROPVAL pvPropVals[2];

```

In this case, you are receiving the values for the area code property and the state property. You use these values passed in to your function as the values of the properties you'll write to the database:

```

    pvPropVals[0].propid = MAKELONG(CEVT_LPWSTR, PROP_AREACODE);
    pvPropVals[0].wFlags = 0;
    pvPropVals[0].val.lpwstr = szCode;
    pvPropVals[1].propid = MAKELONG(CEVT_LPWSTR, PROP_STATE);
    pvPropVals[1].wFlags = 0;
    pvPropVals[1].val.lpwstr = szState;

```


Once you've initialized your CEPROPVAL array, you can then call `CeWriteRecordProps()`:

```
if (!CeWriteRecordProps(m_Handle, 0, 2, &pvPropVals))
```

You then return a value based on the result of `CeWriteRecordProps()`:

```
    return(FALSE);
else
    return(TRUE);
}
```

AreaCodeDatabaseDelete()

To delete a record, you use a function called `CeDeleteRecord()`. So, in the context of this application, to delete a record from your database, you would have a function like the following, where you pass in a string containing the CEOID of the record to be deleted:

```
BOOL AreaCodeDatabaseDeleteRecord(LPWSTR data)
{
```

The next step is to convert the string to a numeric value:

```
    CEOID oidRecord;
    oidRecord = (CEOID)_wtoi(data);
```

Once that's done, all you have to do is call `CeDeleteRecord()`:

```
    ret = CeDeleteRecord(m_Handle, oidRecord);
    return(ret);
}
```

And that's all there is to deleting records from a database.

AreaCodeDatabaseFinish()

The only action you really have to take here is to remember to close the database and free any memory still in use. To close a database, call `CloseHandle()` and pass in the handle to the open database:

```
void AreaCodeDatabaseFinish()
{
    CloseHandle(m_Handle);
}
```

Summary

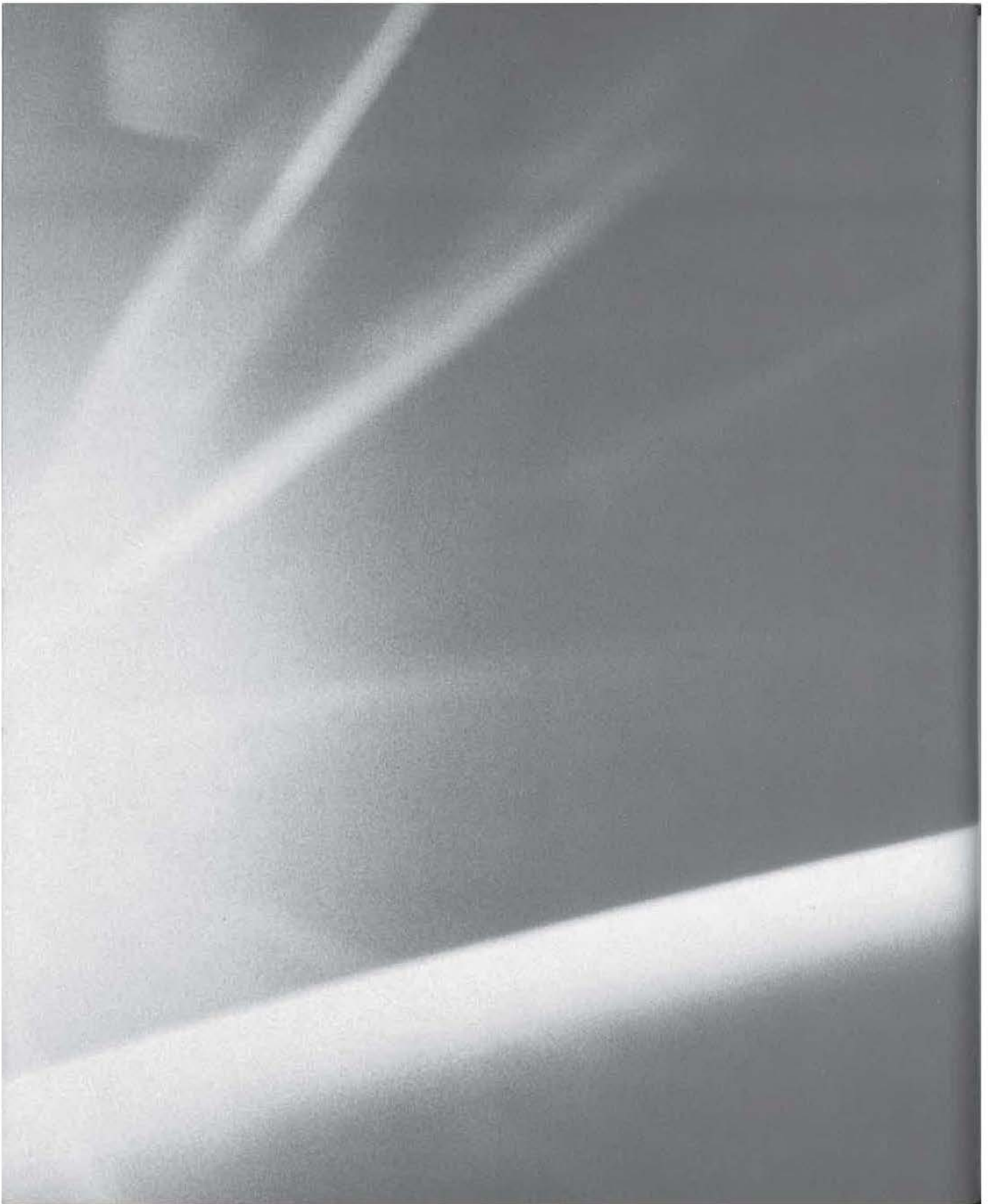
In this chapter, we looked at the basics of CE's structured storage types. There's a lot of information to master here, and it can be quite dizzying at first. With a little bit of practicing on the examples provided here, however, you're sure to have it down in no time.

In Chapter 12, we'll look at some alternatives to API-based database access. Specifically, we'll be looking at MFC's approach to databases, as well as some database engines provided by third-party vendors.

PART II

Mastering the Developer's Tools

- CHAPTER 5: CE Toolkit for Visual C++
- CHAPTER 6: Yes, It's Possible—MFC on CE
- CHAPTER 7: Real MFC Applications Ported to CE
- CHAPTER 8: Visual Basic Toolkit for CE
- CHAPTER 9: A Real VB Application Converted to CE



CHAPTER

FIVE

5

CE Toolkit for Visual C++

- Using VC++ to Develop for Windows CE
- The Windows CE Toolkit for VC++
- The Platform SDKs
- MFC versus SDK-Style Coding

The purpose of this chapter is to show off the Windows CE Toolkit for Visual C++ (VC++). After reading this chapter, you will have a thorough understanding of the most important features of the toolkit and will know how to integrate the toolkit into VC++ as smoothly as possible. You'll also have a sense of what sort of common problems crop up from time to time when using the toolkit. (There are a few, but not too many.)

Along the way, you'll develop a Wizard for VC++ to help you prototype CE applications with a minimum of code and effort by creating small, dialog-based applications! Next, we'll examine the tradeoffs of coding in Microsoft Foundation Classes (MFC) vs. SDK-style coding. Of course, we'll be picking up MFC again in a later chapter. Finally, we'll present a simple SDK-style application, written specifically for the PPC devices, which will illustrate the differences—if any—between “regular” (i.e., desktop) applications and CE applications.

Using VC++ to Develop for Windows CE

When we use Visual C++ to build programs for Windows CE, we're really using two products at once.

The first product is the Microsoft Windows CE Toolkit for Visual C++. The toolkit consists of the core files needed to build and debug executables for the various Windows CE platforms.

The second product is the Software Development Kit (SDK) for the platform we're targeting. The SDK consists of additional tools to help prototype and debug our applications, such as the Windows CE emulators.

In this section, we'll look at the components that make up both the toolkit and the emulator and show you what you really need to know about each.

The Windows CE Toolkit for VC++

As noted, the toolkit is the core that makes it possible for Visual C++ to build executables for the Windows CE operating system, including such tools as

- The compilers
- The Platform Manager
- The Project Wizards

The Compilers

Because CE is an operating system that runs on a variety of chip types, several compilers are included in the toolkit, one for each chip. While the fact that you have more than one compiler might seem unusual to many Windows developers, the compilers are generally fairly transparent. However, there are times when it can be useful to know a bit about them.

The major issue that you'll experience from time to time is that of the compilers not being able to find the correct path or location of certain `.h` or `.lib` files that it needs in order to build and link your project successfully. This issue may appear trivial, but it's not! In fact, questions relating to directories and linking are among the most commonly asked by those new to CE programming.

If you're using the toolkit for VC++ 5, the paths are

BIN

`%WCESDK%\wce\emu\hpc\windows`

`%WCETOOLKIT%\Bin`

`%WCESDK%\wce\bin`

INCLUDE

`%WCETOOLKIT%\Include\WCE200`

`%WCETOOLKIT%\MFC\Include\WCE200`

`%WCETOOLKIT%\ATL\Include`

LIB

`%WCETOOLKIT%\LIB\wce200\%CHIPTYPE%`

`%WCETOOLKIT%\MFC\LIB\wce200\%CHIPTYPE%`

For VC++ 6, the paths are

BIN

```
%WCETOOLS%\Bin
```

INCLUDE

```
%WCETOOLS%\WCE200%\PLATFORM%\Include
```

```
%WCETOOLS%\MFC\ WCE200%\PLATFORM%\Include
```

```
%WCETOOLS%\WCE200\ATL%\PLATFORM%\Include
```

INCLUDE

```
%WCETOOLS%\WCE200%\PLATFORM%\Lib\%CHIPTYPE%
```

```
%WCETOOLS%\MFC\ WCE200%\PLATFORM%\ Lib\%CHIPTYPE%
```

```
%WCETOOLS%\WCE200\ATL%\PLATFORM%\ Lib\%CHIPTYPE%
```

NOTE

There are special assemblers for each of the chips.

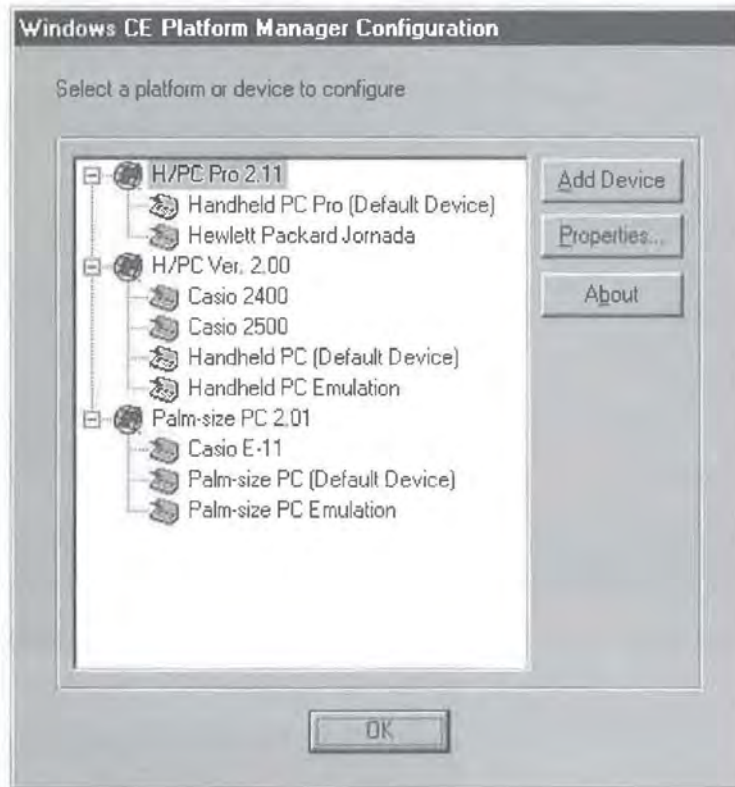
The Platform Manager

The Platform Manager is a new tool provided in the toolkit for VC++ 6 that makes it much easier to specify the target device and the preferred connection method for EXE file uploading and debugging purposes. As shown in Figure 5.1, the Platform Manager can maintain a listing of all your CE devices, organized by platform type.

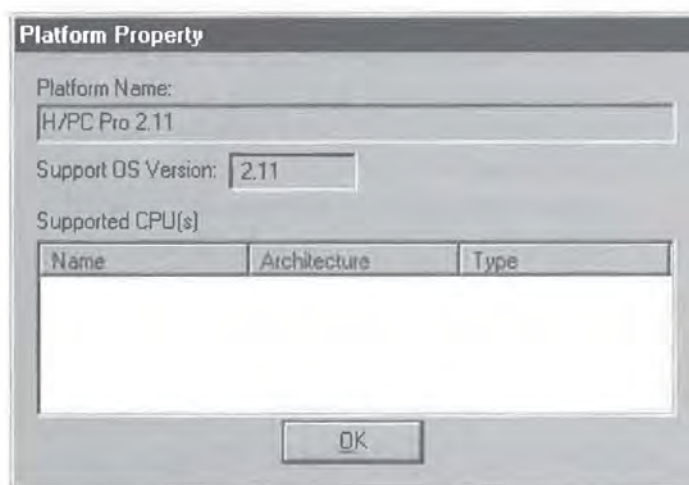
By right-clicking on one of the platform types (e.g., Palm-size PC 2.01) and selecting the Property menu, you can view information about the platform type, including which version of CE is supported and which chip types and architectures are supported. The Platform Property dialog is shown in Figure 5.2.

FIGURE 5.1:

The main view of Platform Manager

**FIGURE 5.2:**

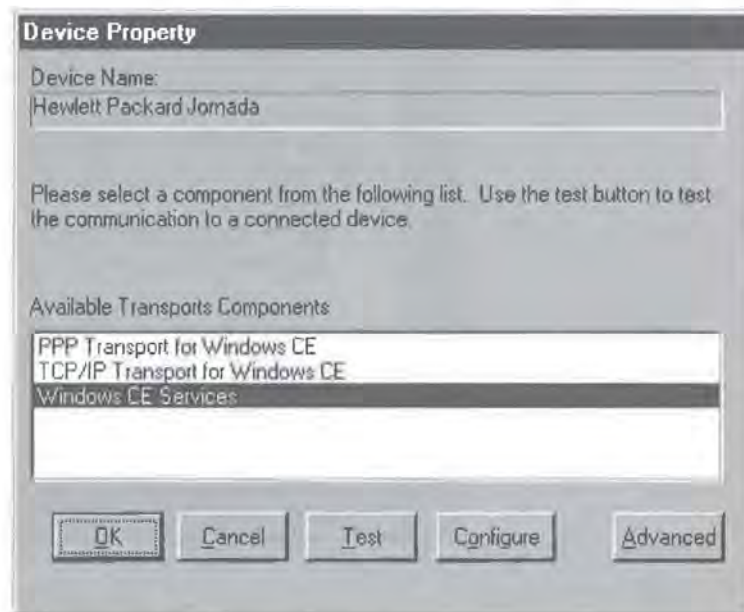
The Platform Property dialog



Additionally, Platform Manager allows you to set the properties for each device registered with the system. By right-clicking on a device and selecting Property, you can tell Platform Manager how you would like the CE development tools to connect to your CE devices. There are three possible ways to connect to a CE device, as shown in Figure 5.3: PPP Transport for Windows CE, TCP/IP Transport for Windows CE, or Windows CE Services.

FIGURE 5.3:

The Device Property dialog



Of the three, Windows CE Services is probably the most common means of connecting to a device. CE Services is the standard, serial-line connection that is usually used for Active Syncing devices with the Desktop. The drawback to debugging over a serial line is that it's very slow. In fact, you'll find that it takes so long to execute a single line of source code while debugging over the serial port that you can actually count the seconds. Obviously, this can be very frustrating.

That's why the debugging method that most developers will probably want to use is the TCP/IP Transport for Windows CE. Basically, this allows you to set up your CE devices and all of the CE-related tools so that they use debugging over Ethernet. The immediate advantage of this is that it is many times faster than debugging over the serial connection.

Ethernet Debugging

When it comes to debugging your applications, the Ethernet TCP/IP Transport is about as fast as it gets. It's so useful and such a time saver, in fact, that it's really worth the investment to get your CE device connected to your LAN.

As mentioned in Chapter 1, there are two options available to get your CE device on your LAN. The first is a wireless LAN solution from Proxim, Inc. of Mountain View, California (<http://www.proxim.com>). Proxim offers a product called the RangeLan2 Bridge, which serves as a kind of wireless hub or access point to the network. The other part of this solution is a PC Card (PCMCIA Card) that you use with the CE Device itself.

Your second option for Ethernet debugging is a solution provided by Socket Communications of Newark, California (<http://www.socket.com.com>). Socket Communications offers a PC Card (PCMCIA Card) or Compact Flash Card that provides an instant (wired) connection to the network. Both the PC Card and the Compact Flash solution are easy to configure. However, the Compact Flash version has the added benefit of taking advantage of a slot you're probably not using, and it will run on Palm-size PC devices, too.

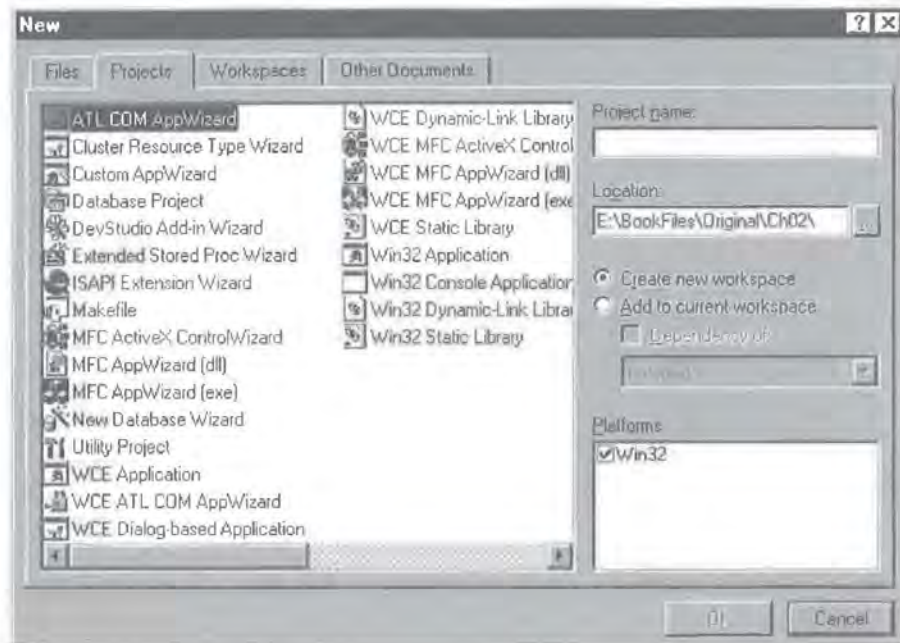
The Project Wizards

The Project Wizards are the add-ins to VC++ that appear whenever you choose File > New and highlight the Projects tab, as shown in Figure 5.4. Although there are over 20 Wizards in all, there are 7 Wizards that help to speed up Windows CE development. They are

- WCE Application
- WCE ATL COM AppWizard
- WCE Dynamic-Link Library
- WCE MFC ActiveX Control
- WCE MFC AppWizard (DLL)
- WCE MFC AppWizard (EXE)
- WCE Static Library

However, of these seven Wizards, you'll probably only use two of them on a regular basis, the WCE Application Wizard and EXE version of the WCE MFC AppWizard.

FIGURE 5.4:
The Project Wizards



WCE Application Wizard Using the WCE Application Wizard is the simplest way to create a full-fledged SDK-style Windows CE application. In less than three steps, the WCE Application Wizard will create

- An empty project, with no source files whatsoever
- A Simple Windows CE Application, which merely has an empty `WinMain()`
- A “Hello World”-style application, complete with a simple window, `WndProc()` function, and an icon, as shown in Figure 5.5

WCE MFC AppWizard The WCE MFC AppWizard is the one you’ll use for every CE-based MFC Application you create. It’s very similar to the standard MFC AppWizard that you’ve probably used to create Desktop-based MFC applications. However, there are a few noticeable differences between the CE version and the Desktop version.

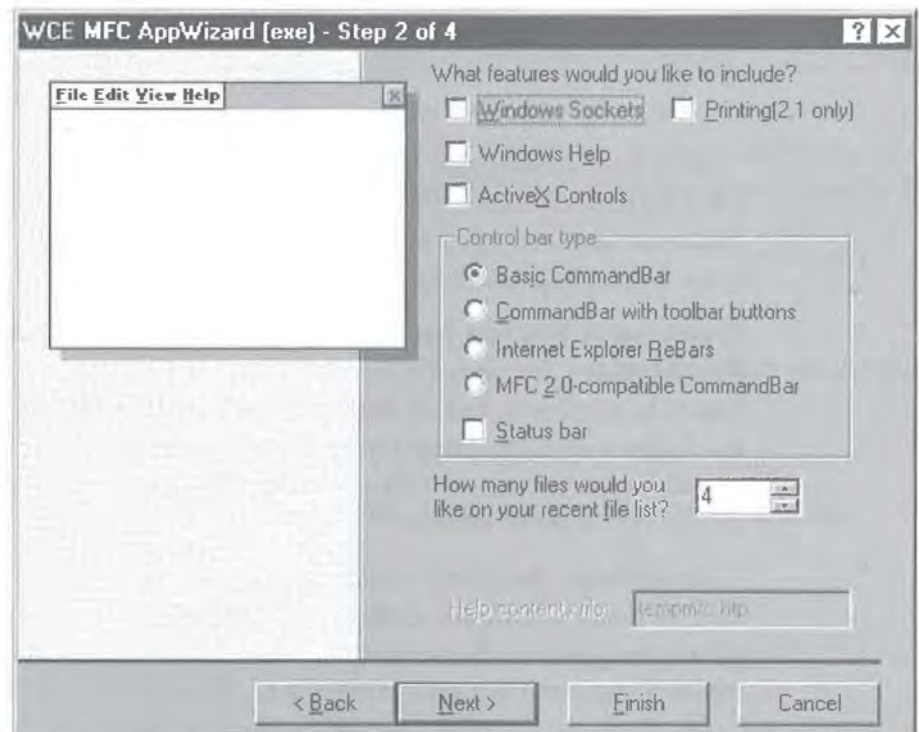
The biggest potential problem to watch out for relates to choosing the correct implementation of `CommandBars` for your application. On the second page of the Wizard (shown in Figure 5.6), you are asked to choose the type of `CommandBar` you’d like for your application.

FIGURE 3.5:

The "Hello World" application

**FIGURE 5.6:**

The WCE MFC AppWizard



The Wizard presents you with four options:

- Basic CommandBar
- CommandBar with toolbar buttons
- Internet Explorer ReBars
- MFC 2.0-compatible CommandBar

The problem is that unless you explicitly choose the fourth option, you will only be able to compile for CE 2.01 or CE 2.11, which are the versions of CE that PPC and HPC/Pro devices are running; all of the HPC devices are running CE 2! And, while it's true that some OEMs will be offering customers the ability to upgrade to CE 2.11, not all of them will. This means that if you don't choose the MFC 2-compatible CommandBar, you will be ignoring a potentially huge market for your software.

Of course, everything has its tradeoffs. When you do choose the 2.0-compatible CommandBar, you lose the option of having a status bar and support for printing, both of which are only available for CE 2.01 and 2.11. The end result of this is that you either have to explicitly target only PPC and HPC/Pro devices, or you have to write your own printing and status bar code.

As of this writing, there are no commercially available HPC devices running CE 2.11; only the HPC/Pro devices run CE 2.11. The Palm-size PCs run CE 2.01. However, as mentioned in Chapter 1, some HPC manufacturers do offer a software or ROM upgrade from CE 2 to CE 2.11.

Creating A Wizard for Dialog-Based Applications—A Sample

Application In previous versions of the toolkit, the array of Wizards was not nearly so complete. In fact, there was no way to quickly prototype an application at all. Even with the current toolkit's extensive list of Wizards, there may be a variety of custom applications or frameworks that you'd also like to be able to use for quick development.

For instance, there's no way to quickly and easily create a dialog-based application. In many situations, dialog-based applications represent a very handy tool in prototyping or rapidly developing an application that would take much longer to create if you used any other method.

To remedy this situation, let's create our own Wizard for VC++ in 10 easy steps, first creating a simple, dialog-based application and the Wizard based on the application.

1. To create the dialog-based application, create a new WCE Application with the Simple Windows CE Application option specified.
2. After VC++ creates the source files and places you in the project, create a simple, generic dialog, as shown in Figure 5.7.

FIGURE 5.7:

The generic dialog



3. Your next task is to create a simple `DialogProc()`. So that your project is generic enough to be useful for a variety of applications, make it handle two messages:

```
WM_INITDIALOG
WM_COMMAND
```

(You may also want to handle `WM_CTLCLORDLG` if you find that your dialog paints strangely on some devices.)

When you're finished, you have a fairly basic-looking `DialogProc()`:

```
BOOL CALLBACK DlgProc (HWND hwnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_INITDIALOG:
        {
            return TRUE;
        }
    }
}
```



```

        case WM_CTLCOLORDLG:
        {
            SetBkColor((HDC)wParam, LTGRAY_BRUSH);
            return FALSE; //for occasional strange painting
        }
        case WM_COMMAND:
        {
            switch(LOWORD(wParam))
            {
                case IDCANCEL:
                case IDOK:
                {
                    EndDialog(hwnd, 0);
                    return TRUE;
                }
            }
        }
        default:
            return FALSE;
    }
}

```

4. The only thing left to do is to edit your `WinMain()` function to launch the dialog:

```

    DialogBox ((HANDLE) hInstance, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
        (DLGPROC)DlgProc);
    return 0 ;

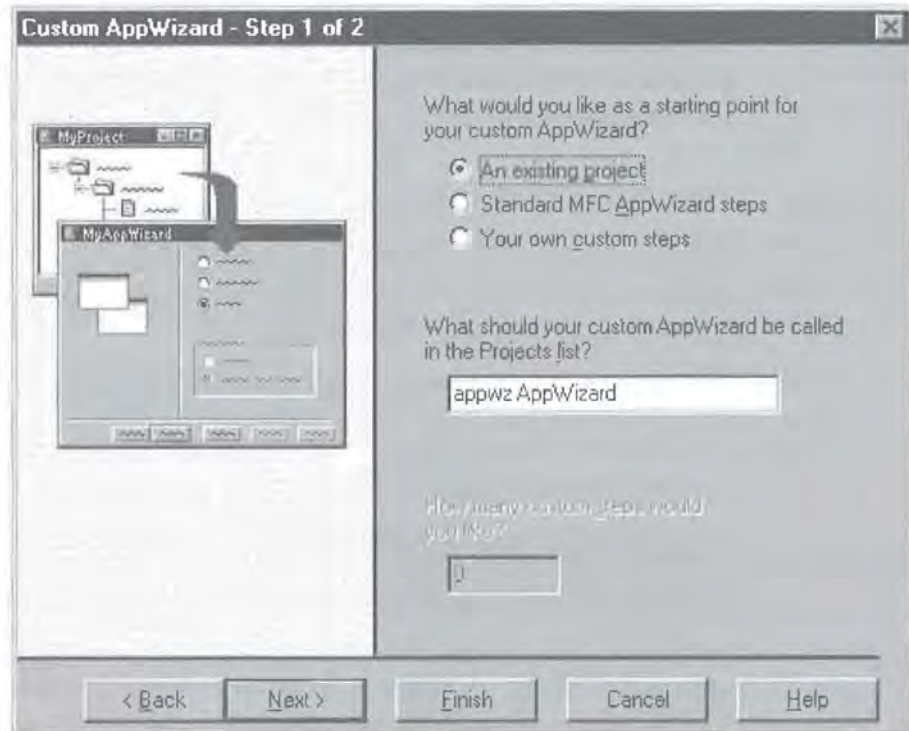
```

You are now done creating your template application.

5. Now to create the Wizard. Choose `File>New` in VC++. Then choose the `Projects` tab, and select `CustomAppWizard` from the list. Enter a name for your new Wizard such as **WCE Dialog-based Application**. Click `Next`.
6. When you've done this, you'll be presented with a list of options, as shown in Figure 5. 8.
7. Choose `An Existing Project` as the starting point for your Wizard, and click the `Next` button.
8. In the dialog that appears, enter the name of the application you created in steps 1–4. Click `Next`. You'll then be asked to confirm your options; click `Done` to finish.

FIGURE 5.8:

The Custom App Wizard dialog



9. When VC++ places you into the project, NEWPROJ.INF should read something like the following:

```
=:ICON1.ICO    icon1.ico
RESOURCE.H    resource.h
=SIMPLE.C      simple.c
ROOT.RC       $$root$.rc
```

Change this so that the line referencing `simple.c`—or the name of your application's main C/C++ file—now reads

```
=SIMPLE.C      $$root$.c
```

This ensures that your main C/C++ file will be renamed to reflect whatever project name is entered into your Wizard when someone creates a new project.

10. Next, compile your Wizard—and it's ready to use!

The Platform SDKs

The platform SDKs can be regarded as two separate packages: First, there are the generic tools that will work for any CE platform. Second, there are the platform-specific tools, such as the emulators.

The Generic Tools

The major part of every SDK is the supplementary tools and utilities designed to make it easier to target and debug the Windows CE platforms. Some of the tools currently shipping with the SDKs are

- Remote File Viewer, a Desktop-based Explorer for CE Devices
- Remote Heap Viewer, which allows you to view information about Heap allocation on your device
- Remote Process Viewer, a kind of Desktop-based task manager for CE
- Remote Registry Editor, which allows you to edit the registry of your CE device
- Remote Spy++, which allows you to view class information and messages for the various windows, both hidden and active, on your CE device
- Remote Zoomin, a tool for taking screenshots of programs running on your device

These tools are included in all of the SDKs and should work with any CE devices, regardless of form factor. However, in each toolkit there is a unique tool that can help you develop for a CE-based form factor, even if you do not own a device.

TIP

All SDK tools are simple RAPI applications, which means that you can easily create your own CE tools for debugging! For more on RAPI and creating RAPI applications, see Chapter 10.

The Emulators

Each of the SDKs comes with a Desktop-based emulator for the platform the SDK is targeting. For example, the AutoPC SDK comes with an emulator for the AutoPC.

The emulators can be a wonderful tool for prototyping and debugging your application before you begin testing it on a device.

There are, however a few drawbacks to using the emulators. The first disadvantage is that they're not true emulators. You must compile a special version of your program just for the emulator.

Second, the emulators only run on Windows NT; they will not run under Windows 98. This is because the emulators, like Windows CE itself, are Unicode-based, while Windows 98 is ANSI-based. It is true that 98 does have stub code for some Unicode operations, but it is not enough to support the CE emulators.

WARNING

Because the emulators and many of the SDK tools do not run under Windows 95/98, it is recommended that you use Windows NT as your development platform. Although the emulators do not work under 95/98, two of the SDK tools will run just fine: Remote Zoomin and Heap Viewer.

The typical CE application development cycle, then, is to first compile for the emulator of your choice (yes, you must actually do a special compilation just for the emulator). Next, test and debug your application in the emulator as thoroughly as possible. Finally, when you believe that your application is ready, upload it to the CE device and repeat the process. Of course, since testing and debugging on a CE device may be quite time-consuming, being able to debug on the Desktop is a serious advantage.

The third real disadvantage of using the emulators is that they tend to be more forgiving of bad code than the devices are. What that means is that even though your code works in the emulator, you may still have some work to do before it will work on the device. Simply put, the emulators are not a 100 percent-reliable measure of how your code will work on a device. For instance, in the HPC emulator, the following code either crashes or runs and displays the desired result:

```
case WM_INITDIALOG:
{
    LPSYSTEMTIME lpSystemTime; //system time struct
    TCHAR chTime[254]; //string to display system time
    GetSystemTime(SystemTime);
    wsprintf((LPTSTR)chTime, TEXT("Today's date: %d/%d/%d"),
lpSystemTime->wMonth, lpSystemTime->wDay, lpSystemTime->wYear);
    SetWindowText(hwnd, TEXT("Hello WinCE!"));
}
```



```

        SendDlgItemMessage(hwnd, IDC_DATETEXT, WM_SETTEXT, 0,
(LPPARAM) (LPTSTR)chTime);
        return TRUE;
    }

```

At first glance, it seems like acceptable code, but look closely. The `lpSystemTime` structure is never initialized to point to a valid memory location before it is passed to `GetSystemTime()`. When this same code is executed on an HPC device, the program runs and behaves relatively normally, with the exception that there is an area of the dialog box that, instead of showing the date, shows the word *Static*.

The corrected code looks like this:

```

    case WM_INITDIALOG:
    {
        SYSTEMTIME SystemTime; //system time struct
        TCHAR chTime[254]; //string to display system time
        GetSystemTime(&SystemTime);
        wsprintf((LPTSTR)chTime, TEXT("Today's date: %d/%d/%d"),
SystemTime.wMonth, SystemTime.wDay, SystemTime.wYear);
        SetWindowText(hwnd, TEXT("Hello WinCE!"));
        SendDlgItemMessage(hwnd, IDC_DATETEXT, WM_SETTEXT, 0,
(LPPARAM) (LPTSTR)chTime);
        return TRUE;
    }

```

Here, the *address* of a `SYSTEMTIME` variable (not just an `LPSYSTEMTIME` variable, as before) is passed to `GetSystemTime()`, which will return the correct values on a device or in the emulator.

MFC vs. SDK-Style Coding

There's ongoing debate about using MFC or SDK-style code. And, in the case of Windows CE, both SDK-style programmers and MFC programmers have had to give up a little. As demonstrated in Chapters 3 and 6, many API calls are missing from the CE API and some MFC classes have been modified or dropped completely on CE. In the end, however, it comes down to two simple issues: storage space and ease of development.

Storage Space

If you create the simplest possible SDK-style application and the simplest possible MFC application, you'll get the following results:

MFC or SDK	EXE Size	Additional DLLs	Total Storage Space
SDK	8.5K	(none)	8.5K
MFC	10K	356K	366K

When developers first started writing for CE, this 350K difference was a big reason that so much SDK-style development was done. However, CE devices and their uses are changing in a couple of ways that make MFC more attractive.

First, many applications are being developed as vertical market applications, which means that the device will likely be running just that one application. In that case, it obviously doesn't matter how much storage space your program occupies, because you have the entire device to yourself.

The second changing aspect of CE devices is that OEMs are now starting to include the MFC runtimes in the ROM of the devices themselves. This means that it will no longer be necessary for you to ship a 350K DLL with your applications.

Of course, not all devices have the DLL in ROM, so this is still an issue for the time being. And, SDK-style programmers will correctly point out that they can still write tighter, more compact code than that which MFC can produce.

Ease of Development

Ease of development is more of an individual decision. If you happen to have an existing program that you're looking to port to Windows CE, you're almost certainly going to choose whichever style of coding the existing program was originally written in.

Only a few MFC classes have not been implemented on CE, so it is typically easier to port existing MFC code than it is to port existing SDK-style code. This is especially true as large numbers of the API and C-runtime library functions have not been implemented for CE. However, just as with the issue of storage space, some of these objections are also fading away. For instance, with the release of

CE 2.11, Microsoft significantly enhanced the available C-runtime libraries, making it easier to port existing code and to write new code.

The end result of this is that the line dividing MFC and SDK-style CE developers is becoming almost invisible. With the release of CE 2.11 and future releases, you'll likely see both styles of coding become equally popular, with no clear benefit to either one. In the next chapters, we'll examine MFC and SDK-style programming in more detail. We'll look at the benefits of each and tell you how to work around what's missing from each when it comes to Windows CE.

A Sample SDK-Style Application

At this point, let's take a look at a sample SDK-style application for CE and examine how it differs from a standard Win32 application. For this demonstration, let's make a simple Task Manager-style application for the Palm-size PC. Currently, the only way to see all of the programs running on your PPC device and close a program that's running requires you to go to Start > Settings > System, then click the Task Manager tab, as shown in Figure 5.9. In our sample application, we will change this so that our Task Manager will allow the user to close any application with one click.

FIGURE 5.9:

The PPC device
Task Manager



Using the Dialog-based App Wizard that you just created, you will now create a simple, dialog-based application from which you'll begin your Task Manager. Your Task Manager will have three purposes:

- To display a list of running tasks
- To allow the user to switch to any of the tasks
- To allow the user to close any of the tasks

We will cover each of these purposes separately in the following sections.

Displaying a List of Running Tasks

The first step in this process is to add a list box to your dialog box that will show the list of running tasks once they've been retrieved from the system. Using the dialog editor of VC++, go ahead and add a list box by selecting the ListBox icon from the toolbar and dragging in onto your form. Then, right-click the list box, select Properties, and rename its identifier to IDC_LISTTASKS. Since you'll also need an End Task and a Switch To button, go ahead and add them to the form now. Give the End Task button an identifier of IDC_ENDTASK and the Switch To button an identifier of IDC_SWITCHTO.

Now, on to the list of tasks. Filling the list box with the list of windows on the system is quite easy. First, make sure to clear out any existing contents of the list box:

```
SendMessage(hwndCtrl, LB_RESETCONTENT, 0, 0);
```

Then, begin looping through all of the windows on the system with successive calls to `GetWindow()`:

```
hWndAWindow = GetWindow(hwnd, GW_HWNDFIRST);  
while (hWndAWindow != 0)  
{
```

Be careful to filter out your own application, any hidden windows, any child windows, and, of course, the Desktop itself:

```
if ((hWndAWindow != hwnd) &&  
    IsWindowVisible(hWndAWindow) &&  
    (GetWindow(hWndAWindow, GW_OWNER) == 0) &&  
    (GetWindowText(hWndAWindow, szBuff, MAX_PATH) != 0)  
&&  
    (lstrcmpi (szBuff, TEXT("Desktop")) != 0))
```


If the window you find passes these tests, simply add its caption to your list box and retrieve the next window handle:

```

        SendMessage(hwndCtrl, LB_ADDSTRING, 0, (LPARAM)
(LPCTSTR) szBuff);
        hWndAWindow = GetWindow(hWndAWindow, GW_HWNDNEXT);
    }

```

When it's all put together, it looks like this:

```

    SendMessage(hwndCtrl, LB_RESETCONTENT, 0, 0);
    hWndAWindow = GetWindow(hWnd, GW_HWNDFIRST);
    while (hWndAWindow != 0)
    {
        if ((hWndAWindow != hWnd) && IsWindowVisible(hWndAWin-
dow) &&
            (GetWindow(hWndAWindow, GW_OWNER) == 0) &&
            (GetWindowText(hWndAWindow, szBuff, MAX_PATH) != 0)
            &&
            (!strcmpi (szBuff, TEXT("Desktop")) != 0))
            SendMessage(hwndCtrl, LB_ADDSTRING, 0, (LPARAM)
(LPCTSTR) szBuff);
        hWndAWindow = GetWindow(hWndAWindow, GW_HWNDNEXT);
    }

```

A Missing API Call and Some Recycled Code

With a few changes, the above code can double as a replacement for `GetDesktopWindow()`, one of the API calls that's missing from the CE API. `GetDesktopWindow()` is used to get the window handle of the desktop. This has many uses, including retrieving the device context (DC) of the desktop so that you can paint directly on it. However, although it is used quite commonly in desktop PC programming, `GetDesktopWindow()` is not supported under Windows CE.

If you look at the code we just wrote, though, you'll note that we're making sure not to add the Desktop window to our task list. Therefore, it stands to reason that there is a window calling itself the Desktop. If you know this, you can get its handle. For instance, if you modified your existing block of code from the example above so that you are explicitly looking for that Desktop window, you can simulate `GetDesktopWindow()`.

Continued on next page

The modified code for a `GetDesktopWindow()`-style routine has the same `GetWindow()` loop as the above example:

```
hWndAWindow = GetWindow(hwnd, GW_HWNDFIRST);
while (hWndAWindow != 0)
{
```

Again, it retrieves the caption of each window:

```
    GetWindowText(hWndAWindow, szBuff, MAX_PATH);
```

This time, however, it only quits if it finds one whose caption matches the word *Desktop*:

```
    if (!strcmpi (szBuff, TEXT("Desktop"))) == 0)
    {
        break;
    }
```

Otherwise, it continues to retrieve the next window:

```
        hWndAWindow = GetWindow(hWndAWindow, GW_HWND-
NEXT);
    }
```

When assembled, the entire block of code looks like this:

```
hWndAWindow = GetWindow(hwnd, GW_HWNDFIRST);
while (hWndAWindow != 0)
{
    GetWindowText(hWndAWindow, szBuff, MAX_PATH);
    if (!strcmpi (szBuff, TEXT("Desktop"))) == 0)
    {
        break;
    }
    hWndAWindow = GetWindow(hWndAWindow, GW_HWND-
NEXT);
}
```

With just a few lines of mostly recycled and borrowed code, you've just managed to simulate one of the missing API calls. Of course, you might not be so lucky all the time—for more on working around missing API calls, see Chapter 3.

Switching to One of the Tasks

Switching to one of the tasks in the task window is extremely simple. All you have to do is to add a handler for the `WM_COMMAND` message, since the user will be clicking a button in order to make this work. First, make sure that the user actually did select an item from the list:

```

        cbCnt = (int)SendMessage(GetDlgItem(hwnd,
IDC_LISTTASKS), LB_GETCURSEL, 0, 0L);
        if ((cbCnt < 0))
        {
            MessageBox(hwnd, TEXT("You must first
select a task."), TEXT("Error: No task"), MB_OK | MB_ICONSTOP);
            return TRUE;
        }

```

If they did select an item, retrieve the text of the currently selected item:

```

        SendMessage(GetDlgItem(hwnd, IDC_LISTTASKS),
        LB_GETTEXT,
        (LPARAM)SendMessage(GetDlgItem(hwnd,
IDC_LISTTASKS), LB_GETCURSEL, 0, 0L),
        (LPARAM)(LPSTR)szBuff);

```

Then, make sure the window still exists by calling `FindWindow()`:

```

        if (FindWindow(NULL, szBuff) != NULL)
        {

```

If it does, call `SetForegroundWindow()`:

```

            hWndAWindow = FindWindow(NULL, szBuff);
            SetForegroundWindow(hWndAWindow);

```

This done, the only job remaining is to add the logic for closing a running application.

Closing an Application

You may have noticed that in the last code example, which switches to an application, the `if (FindWindow(NULL, szBuff) != NULL)` block was never actually finished. This is because with just another few lines of code, you can make the same routine serve to close an application as well. All you have to add to the above code is:

```

        if (LOWORD(wParam) == IDC_ENDTASK)
        {

```

```

    SendMessage(hWndAWindow, WM_CLOSE, 0,
0);
    }

```

Then, add one additional line to close the Task Manager:

```
SendMessage(hWnd, WM_CLOSE, 0, 0);
```

NOTE

Close the Task Manager after the user has switched to another program or closed a program to be consistent with the behavior of the Windows 98 Task Manager, which does the same thing.

At first, this handling of two button events with the same WM_COMMAND message handler might seem a bit confusing, so let's take a look at the entire message handler block of code:

```

    case WM_COMMAND:
    {
        switch(LOWORD(wParam))
        {
            case IDCANCEL:
            case IDOK:
            {
                EndDialog(hWnd, 0);
                return TRUE;
            }
            case IDC_ENDTASK:
            case IDC_SWITCHTO:
            {
                int    cbCnt;
                TCHAR  szBuff[MAX_PATH];
                HWND   hWndAWindow;
                cbCnt = (int)SendMessage(GetDlgItem(hWnd,
IDC_LISTTASKS), LB_GETCURSEL, 0, 0L);
                if ((cbCnt < 0))
                {
                    MessageBox(hWnd, TEXT("You must first
select a task."), TEXT("Error: No task"), MB_OK | MB_ICONSTOP);
                    return TRUE;
                }
                SendMessage(GetDlgItem(hWnd, IDC_LISTTASKS),
LB_GETTEXT,

```



```

                                (LPARAM)LPSTR)szBuff);
IDC_LISTTASKS), LB_GETCURSEL, 0, 0L),
                                (LPARAM)LPSTR)szBuff);
window still exists!
                                if (FindWindow(NULL, szBuff) != NULL) //the
                                {
                                    hWndAWindow = FindWindow(NULL, szBuff);
                                    SetForegroundWindow(hWndAWindow);
                                    if (LOWORD(wParam) == IDC_ENDTASK)
                                    {
                                        SendMessage(hWndAWindow, WM_CLOSE, 0,
0);
                                    }
                                    SendMessage(hWnd, WM_CLOSE, 0, 0);
                                    return TRUE;
                                }
                                else //if you didn't find the app, it's time to
refresh the list of tasks
                                    SendMessage(hWnd, WM_USER + 200, 0, 0);
                                break ;
                            }
                        }

```

As you can see, it makes sense to handle both the IDC_ENDTASK and the IDC_SWITCHTO button clicks in the same handler because, with the exception of the very last `if` statement, they need to do exactly the same thing. Both of them need to make sure the user selected an item from the list, and both of them need to make sure that if they did select an item, that item is still a valid window. By combining the two button handlers into one, you can make the application much more efficient.

One Last Snag: Keeping the List of Tasks Current

If you look closely at the entire WM_COMMAND message handler, you'll notice that there's one extra line of code at the end that we haven't yet discussed. It reads

```
SendMessage(hWnd, WM_USER + 200, 0, 0);
```

This simple line of code exists to solve a common Task Manager problem: it ensures that the list of tasks in the list box is current. In other words, you don't want the application to display a task that's no longer running. Normally, you might handle something like this by setting up a timer, but here that's not necessarily the most efficient way to accomplish this.

Instead, take the code that fills up your list box and move it to a custom message (say, `WM_USER+200`) handler, like this:

```
case WM_USER + 200:
{
    //fill list box code get moved here...
}
```

Then, ensure that the Task Manager application sends itself this `WM_USER + 200` message on any one of the following three occasions:

- When the program starts up
- When the program receives a `WM_CTLCLORDLG` message (the closest thing to `WM_PAINT` when working with dialogs)
- Whenever the user selects a task from the list that no longer exists

The first case can be handled by amending the `WM_INITDIALOG` handler to read

```
case WM_INITDIALOG:
{
    SendMessage(hwnd, WM_USER + 200, 0, 0);
    return TRUE;
}
```

The second case can be implemented similarly with regard to the `WM_CTLCLORDIALOG` message:

```
case WM_CTLCLORDLG:
{
    SendMessage(hwnd, WM_USER + 200, 0, 0);
    return TRUE;
}
```

The third item, as you saw already, is implemented in the `WM_COMMAND` handler by trapping the condition of the selected item not being a valid window handle. This is shown in the last line of this block of code:

```
if (FindWindow(NULL, szBuff) != NULL) //the
window still exists!
{
    hWndAWindow = FindWindow(NULL, szBuff);
    SetForegroundWindow(hWndAWindow);
    if (LOWORD(wParam) == IDC_ENDTASK)
    {
```

```
    SendMessage(hWndAWindow, WM_CLOSE, 0, 0);  
    }  
    SendMessage(hWnd, WM_CLOSE, 0, 0);  
    return TRUE;  
    }  
    else //if you didn't find the app, it's time to  
    refresh the list of tasks  
        SendMessage(hWnd, WM_USER + 200, 0, 0);
```

When you're finished, you'll have a professional-looking application that will make the Palm-size PC devices much easier to use.

TIP

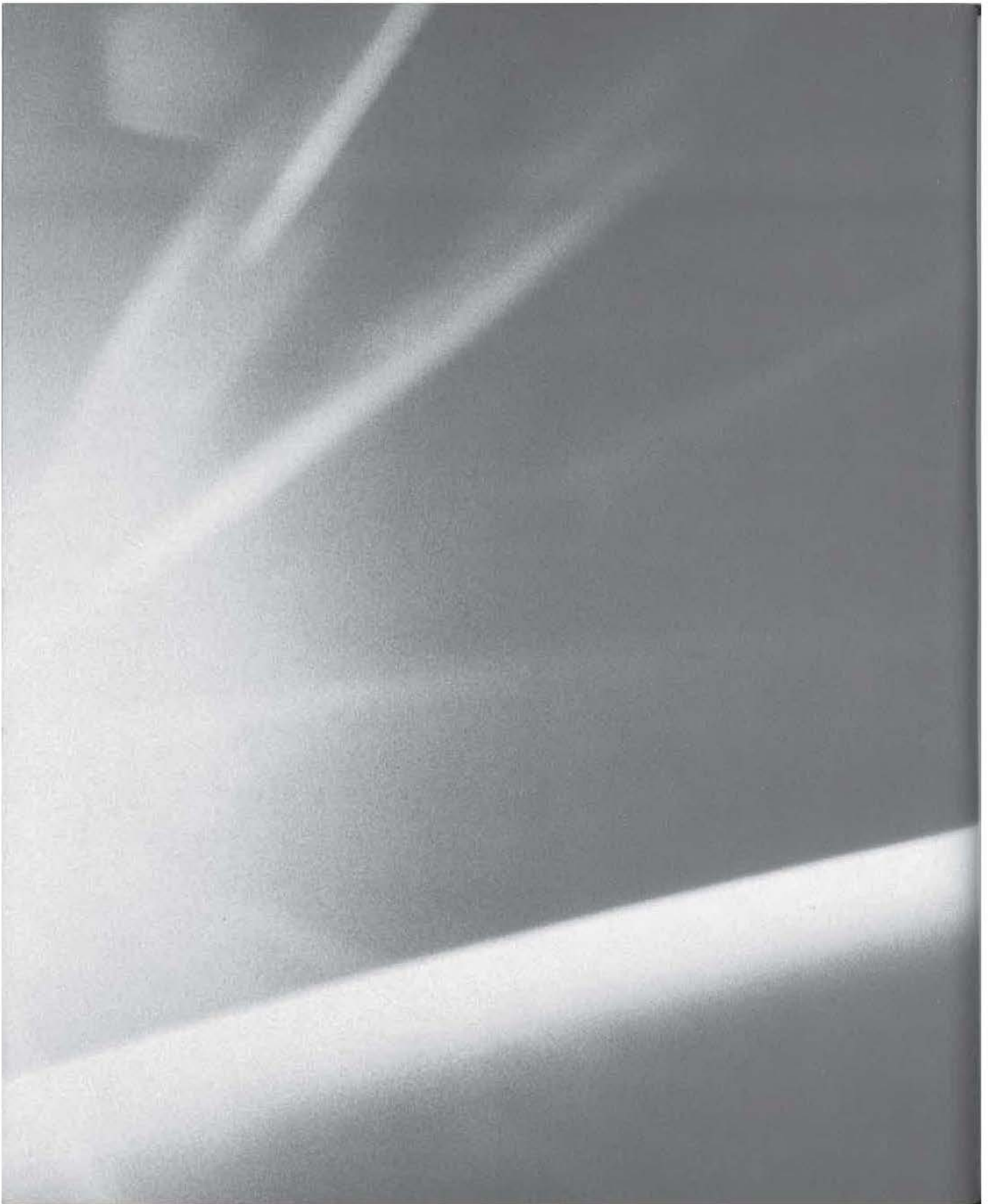
There's absolutely no reason you can't run your Task Manager on an HPC device, too.

As you can see, SDK-style coding for CE doesn't look all that different from SDK-style coding for desktop PCs. The main difference between SDK-style coding and any Win32 code you may have written is the use of the TEXT macro to account for Unicode strings. Of course, you can and *should* use this macro when coding for 98 or NT, but most programmers still regard it as optional.

Summary

In this chapter, we learned the basics of using Visual C++ to build applications for Windows CE. We also examined some of the advantages of both MFC and SDK-style coding. Finally, we made a simple Task Manager application to make our Palm-size PCs more user-friendly.

In the next chapters, we'll be exploring more MFC on CE as well as Visual Basic on CE. MFC has several "new" classes that exist only on the CE platform, and we'll be looking at each of them and how to use them. Similarly, Visual Basic on CE has a number of unique tools and features that we'll learn to master.



CHAPTER

SIX

6

Yes, It's Possible—MFC on CE!

- New Classes for CE
- Modified Classes
- Classes That Lost Functionality
- Classes That Gained Functionality
- Missing Classes

In chapter 5, we looked at some of the differences between coding in an SDK-style versus using MFC. In this chapter, we'll explore the differences between MFC on the Desktop and MFC on a CE device. While most of MFC remains unchanged, there are a few changes we need to know about. Generally, these changes fall into three categories:

- New classes and how to use them
- Modified classes and how to deal with them
- Deleted classes and how to live without them

We'll be taking a look at each of these areas and how they'll affect our applications.

New Classes for CE

Because some of the functionality offered by the Windows CE API was so different from anything in the desktop world, Microsoft found it necessary to add five new classes to the MFC class library. They are:

- `CCeSocket`
- `CCeDBEnum`
- `CCeDBDatabase`
- `CCeDBProp`
- `CCeDBRecord`

The first of these classes, `CCeSocket`, handles the limited Winsock operations supported by Windows CE. The remaining four classes allow MFC programs to use the Windows CE database engine.

CCeSocket

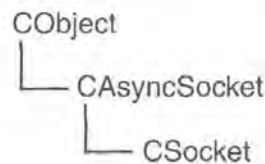
The `CCeSocket` is a very unusual class. Mostly, that's because all of its documentation advises us that `CAsyncSocket` is not supported under Windows CE and that we should use `CCeSocket` instead. However, this is only partially true.

In fact, we should use `CCeSocket` for all Winsock-based communication under Windows CE. That's because CE doesn't support true asynchronous communication. Instead, `CCeSocket` simulates asynchronous events by creating two threads that watch for socket-related operations and events. When an event occurs, the threads pass the notification to the correct message handler. That's the part of the statement about `CAsyncSocket` and `CCeSocket` that's true.

The part that isn't true is that there is, in fact, a `CAsyncSocket` on Windows CE. It's just that it's buried about three levels deep and exists only as a base class upon which the other two socket classes of Windows CE (`CSocket` and `CCeSocket`) are built.

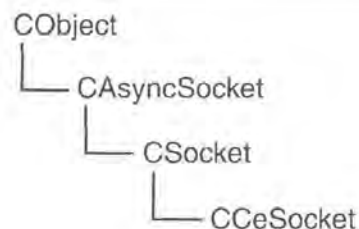
Under Windows 98/NT, the object hierarchy for socket classes looks something like the diagram in Figure 6.1.

FIGURE 6.1:
Socket classes object hierarchy under Windows 98/NT



Under Windows CE, there is one extra class, as shown in Figure 6.2.

FIGURE 6.2:
Socket classes object hierarchy under Windows CE



TIP

For more on communication with Windows CE, see Chapter 11.

TIP

MFC for CE also defines a custom Windows message, `WM_SOCKET_NOTIFY`, for use with the `CCeSocket` class. However, since this message appears to be used only internally, it should not affect our development at all.

The remaining four classes in our list are CE database classes and exist strictly for the purpose of encapsulating the Windows CE Database Engine. In this section, we'll take a look at each class and how to use it.

CCeDBEnum

`CCeDBEnum` is a class that wraps the `CeFindFirstDatabase()/CeFindNextDatabase()` functions. When you create a `CCeDBEnum` object—just as with the `CeFindFirstDatabase()` function—you can pass in a database-type identifier to look for a specific type of database, or you can pass in 0 to tell the object to enumerate all databases. You can then retrieve the CEOID of each database matching your criteria by calling the class's `Next()` method.

In code, retrieving a listing of all databases might look something like this:

```
CCeDBEnum DBEnum;
DBEnum = new CCeDBEnum(0);
while (idCEOID = DBEnum.Next())
{
    //...do some processing here
}
//...
```

CCeDBProp

The `CCeDBProp` class encapsulates the `CEPROPVAL` structure used to set and retrieve values for properties (fields) of a database record. Table 6.1 shows the methods of `CCeDBProp` and the members of the `CEPROPVAL` structure to which they correspond.

TABLE 6.1: CCeDBProp Methods and Corresponding CEPROPVAL Members

Method	CEPROPVAL members	Description
GetBlob()	myCEPROPVAL.val.blob	Retrieves the CEBLOB value of the property
GetFiletime()	myCEPROPVAL.val.filetime	Retrieves the FILETIME value of the property
GetIdent()	HIWORD(myCEPROPVAL.propid)	Retrieves the identifier of the property
GetLong()	myCEPROPVAL.val.lVal	Retrieves the long int value of the property
GetShort()	myCEPROPVAL.val.iVal	Retrieves the int value of the property
GetSortFlags()	myCEPROPVAL.wFlags	Retrieves the flags specifying how the property is sorted
GetString()	myCEPROPVAL.val.lpwstr	Retrieves the wide string value of the property
GetType()	LOWORD(myCEPROPVAL.propid)	Retrieves the type of the property
GetULong()	myCEPROPVAL.val.uVal	Retrieves the unsigned long int value of the property
GetUShort()	myCEPROPVAL.val.uVal	Retrieves the unsigned int value of the property
SetBlob()	myCEPROPVAL.val.blob	Retrieves the CEBLOB value of the property
SetFiletime()	myCEPROPVAL.val.filetime	Sets the FILETIME value of the property
SetIdent()	HIWORD(myCEPROPVAL.propid)	Sets the identifier of the property
SetLong()	myCEPROPVAL.val.lVal	Sets the long int value of the property
SetShort()	myCEPROPVAL.val.uVal	Sets the unsigned int value of the property
SetSortFlags()	myCEPROPVAL.wFlags	Sets the flags specifying how the property is sorted
SetString()	myCEPROPVAL.val.lpwstr	Sets the wide string value of the property
SetType()	LOWORD(myCEPROPVAL.propid)	Sets the type of the property
SetULong()	myCEPROPVAL.val.uVal	Sets the unsigned long int value of the property
SetUShort()	myCEPROPVAL.val.uVal	Sets the unsigned int value of the property

The one point you must be aware of here is that these methods to set and retrieve a value of a certain type do not perform any conversions. If you attempt to read a numeric value from a property that contains a string, you will likely get invalid data and cause an error. In other words, you must always check the type of the property before reading its data.

Although there are lots of ways to accomplish this, the most generic—and, it would seem, most popular—is through the use of a `switch..case` block based on the value returned by the `GetType()` method:

```
switch(MyCeDBProp.GetType())
{
    case CEVT_I2:
        wsprintf(szBuf, TEXT("%d"), MyCeDBProp.GetShort());
        break;
    case CEVT_UI2:
        wsprintf(szBuf, TEXT("%d"), MyCeDBProp.GetUShort());
        break;
    case CEVT_I4:
        wsprintf(szBuf, TEXT("%d"), MyCeDBProp.GetLong());
        break;
    case CEVT_UI4:
        wsprintf(szBuf, TEXT("%d"), MyCeDBProp.GetULong());
        break;
    case CEVT_FILETIME:
        wsprintf(szBuf, TEXT("FILETIME DATA"));
        break;
    case CEVT_LPWSTR:
        wsprintf(szBuf, MyCeDBProp.GetString());
        break;
    case CEVT_BLOB:
        wsprintf(szBuf, TEXT("BLOB_DATA"));
        break;
}
```

This `switch..case` block is very similar to the one we'll create later in Chapter 10. One of the reasons for this is that although the MFC classes for accessing data do provide some abstraction from the API, they do not provide the same level of abstraction as, say, `CString` provides.

CCeDBRecord

The CCEDBRecord class serves mostly as a way to group a set of CCEDBProp objects together so that they may be treated as a record. In reality, this class is little more than a wrapper for a COBArray object. This means that the methods of CCEDBRecord do not have any API equivalents, and they have absolutely no effect on the underlying data. They exist only to provide a friendly, logical way to associate CCEDBProp objects. Table 6.2 lists the CCEDBRecord methods and reveals what each method actually does.

TABLE 6.2: CCEDBRecord Methods and What They Do

Method	Description
AddProp()	Calls the COBArray.Add() method
AddProps()	Calls the COBArray.Add() method for each property passed in
DeleteAllProps()	Removes all elements from the COBArray and frees all used memory
DeleteProp()	Removes specified property from COBArray
GetNumProps()	Returns number of elements in COBArray
GetPropFromIdent()	Finds specified property and returns CCEDBProp object
GetPropFromIndex()	Returns n th CCEDBProp from COBArray

CCeDBDatabase

The CCEDBDatabase class encapsulates all of the API calls dealing with database creation and seeking and accessing records. In short, CCEDBDatabase makes it possible for us to treat a CE database as an object. Table 6.3 lists the methods of the CCEDBDatabase class and shows the API function that each method corresponds to.

NOTE

In Table 6.3, only the key values have been filled in for each API function.

TABLE 6.3: CCEDBDatabase Methods and Corresponding API Functions

CCeDBDatabase Method	API function	Description
AddRecord()	CeWriteRecordProps(..., 0, ..., ...)	Writes the specified CCEDBRecord to the database. Used for appending a new record to the database.
Close()	CloseHandle()	Closes the database.
Delete()	CeDeleteDatabase(idCEOID)	Deletes the database.
DeleteCurrRecord()	CeDeleteRecord(hDatabase, idCEOID)	Deletes the record with the specified CEOID from the database.
DeleteCurrRecordProps()	cepvPropVal.wFlags = CEDB_PROPDELETE	Equivalent to setting the wFlags of a CEPROPVAL to CEDB_PROPDELETE. Deletes the specified property or properties from the current record.
Exists()	CeOpenDatabase()	Attempts to open specified database; if not found, function returns FALSE.
GetCurrIndex()	CeSeekDatabase(..., CEDB_SEEK_CURRENT, ..., &dwIndex)	Calls CeSeekDatabase() to retrieve the index value returned in the fourth parameter.
GetCurrRecord()	CeSeekDatabase(..., CEDB_SEEK_CURRENT, ..., ...)	Retrieves the current record.
GetIdent()	CeOidGetInfo(idCEOID, ...)	Calls CeOidGetInfo() to retrieve the database type identifier.
GetLastModified()	CeOidGetInfo(idCEOID, ...)	Calls CeOidGetInfo() to retrieve the date that the database was last modified.
GetName()	CeOidGetInfo(idCEOID, ...)	Calls CeOidGetInfo() to retrieve the database name.
GetNumRecords()	CeOidGetInfo(idCEOID, ...)	Calls CeOidGetInfo() to retrieve the number of records in the database.
GetSize()	CeOidGetInfo(idCEOID, ...)	Calls CeOidGetInfo() to retrieve the database size.
GetSortProps()	CeOidGetInfo(idCEOID, ...)	Calls CeOidGetInfo() to retrieve the database's SORTORDERSPEC array.

TABLE 6.3 CONTINUED: CCEDBDatabase Methods and Corresponding API Functions

CCEDBDatabase Method	API function	Description
Open()	CeOpenDatabase()	Calls CeOpenDatabase() to open the database.
ReadCurrRecord()	CeReadRecordProps()	Calls CeReadRecordProps() to read in the current record.
SeekFirst()	CeSeekDatabase(..., CEDB_SEEK_BEGINNING, 0, ...)	Calls CeSeekDatabase() and passes in CEDB_SEEK_BEGINNING.
SeekFirstEqual()	CeSeekDatabase(..., CEDB_SEEK_VALUEFIRSTEQUAL, cepvPropToMatch, ...)	Calls CeSeekDatabase() and tells it to try to match a specific value/property.
SeekLast()	CeSeekDatabase(..., CEDB_SEEK_END, 0, ...)	Calls CeSeekDatabase() and passes in CEDB_SEEK_END.
SeekNext()	CeSeekDatabase(..., CEDB_SEEK_CURRENT, 1, ...)	Calls CeSeekDatabase() and tells it to advance the record pointer by one record.
SeekNextEqual()	CeSeekDatabase(..., CEDB_SEEK_NEXTEQUAL, ..., ...)	Calls CeSeekDatabase() and passes in CEDB_SEEK_NEXTEQUAL.
SeekPrev()	CeSeekDatabase(..., CEDB_SEEK_CURRENT, -1, 0)	Calls CeSeekDatabase() and tells it to move the record pointer back by one record.
SeekToIndex()	CeSeekDatabase(..., CEDB_SEEK_END, -1val, ...) or CeSeekDatabase(..., CEDB_SEEK_BEGINNING, 1val, ...)	Calls CeSeekDatabase() and tells it to seek from the beginning or end of the database if the number of records to seek is positive or negative, respectively.
SeekToRecord()	CeSeekDatabase(..., CEDB_SEEK_CEID, idCEID, ...)	Calls CeSeekDatabase() and tells it to find the record with a matching CEID.
SeekValueGreater()	CeSeekDatabase(..., CEDB_SEEK_VALUEGREATER, cepvPropToMatch, ...)	Calls CeSeekDatabase() and tells it to find the first value/property greater than that specified.
SeekValueSmaller()	CeSeekDatabase(..., CEDB_SEEK_VALUESMALLER, cepvPropToMatch, ...)	Calls CeSeekDatabase() and tells it to find the first value/property smaller than that specified.

TABLE 6.3 CONTINUED: CCEDBDatabase Methods and Corresponding API Functions

CCEDBDatabase Method	API function	Description
SetIdent()	CeSetDatabaseInfo (idCEOID,...)	Calls CeSetDatabaseInfo () to set the database type identifier.
SetLastModified()	CeSetDatabaseInfo (idCEOID,...)	Calls CeSetDatabaseInfo () to set the last modified date.
SetName()	CeSetDatabaseInfo (idCEOID,...)	Calls CeSetDatabaseInfo () to set the database name.
SetSortProps()	CeSetDatabaseInfo (idCEOID,...)	Calls CeSetDatabaseInfo () to set the database's SORTORDERSPEC array.
WriteCurrRecord()	CeWriteRecordProps(...,idCEOID,...)	Calls CeWriteRecordProps() to save changes to current record. Used for editing an existing record.

Using the CCEDB Classes: Database as Document

In this section, we'll look at how to integrate the CCEDB classes into an MFC application. Specifically, we'll take an existing Windows CE MFC application and convert it so that it stores data in a database instead of in a file.

TIP

Although the techniques outlined here apply to any file-based MFC application, the project we're using is the one we'll be working with in the next chapter.

The first thing you need to know about using the CCEDB classes is that in the MFC Document/View architecture, the database itself becomes the Document. In order to make this happen, you need to add one member data element to the public section of our CShoppingListDoc class. This will be a CCEDBDatabase object called m_Mydb. Or, in code:

```
// Attributes
public:
    CCEDBDatabase m_Mydb;
```

WARNING

Previously, in the file-based version of this application, the `CShoppingListDoc` class had only public member data and no methods. The example here assumes that you have already deleted the existing member data elements.

In addition to that data element, you'll also add four methods to our `CShoppingListDoc` class:

- `CreateDB()`
- `OpenDB()`
- `CloseDB()`
- `AddRecord()`

Before you implement these methods, though, you need to define some global constants. The first of these is the database type identifier, `DB_IDENT`:

```
const DWORD DB_IDENT = 13245;
```

Next, define a constant to hold the name of the database:

```
const WCHAR DB_NAME[] = _T("ShoppingCEDatabase");
```

Finally, declare the property type identifiers, one for each of the record properties:

```
const WORD PROP_DESCRIPTION = 101,  
        PROP_QTY = 102,  
        PROP_STORE1 = 103,  
        PROP_STORE2 = 104,  
        PROP_STORE3 = 105,  
        PROP_STORE4 = 106;
```

CreateDB() The `CreateDB()` will be responsible for creating the actual database that your application will use to store the user's shopping list data. First, declare a variable of type `CEOID` that you'll use in just a moment when you create the database:

```
BOOL CShoppingListDoc::CreateDB()  
{  
    CEOID poid;
```

Next, set up the sort order for your database. To do this, create a one-cell array of type `CCeDBProp`. The property for which you're creating the sort order is a

string property, identified by the value `PROP_DESCRIPTION`. The sort order will be ascending, case insensitive:

```
CCeDBProp SortProps[1] =
{
    CCeDBProp(CCeDBProp::Type_String, PROP_DESCRIPTION,
        CCeDBProp::Sort_Ascending | CCeDBProp::Sort_CaseInsensi-
tive)
};
```

Next, it's time to attempt to create the database. At the same time you do that, store the result (a `CEOID`) as well and verify that the creation was successful:

```
if (!(poid = m_Mydb.Create(DB_NAME, DB_IDENT, 1, SortProps)))
    return FALSE;
Next, call CShoppingListDoc::OpenDB()
return ::OpenDB();
}
```

OpenDB() The next method you'll implement is your `OpenDB()` function. This function tests whether or not the database has already been created by calling the `Exists()` method of the `CCeDBDatabase` class:

```
BOOL CShoppingListDoc::OpenDB()
{
    if (CCeDBDatabase::Exists(DB_NAME))
    {
```

NOTE

`Exists()` is a static method of the `CCeDBDatabase` class, which means you should not call it as a member function of a specific object; instead, you should call it as you would any other globally available function.

If the database does not exist, you must attempt to create it. This accounts for first-time uses of the program where the database isn't necessarily created ahead of time:

```
if (!CreateDB())
    return FALSE;
}
```

Otherwise, if the database does exist, simply open it, calling the `Open()` method of your `CCeDBDatabase` object, `m_Mydb`:

```
    else
    {
        m_Mydb.Open(DB_NAME);
    }
    return TRUE;
}
```

CloseDB() The `CloseDB()` method is the simplest of the ones you're creating here. All it does is call the `Close()` method of our `CCeDBDatabase` object, `m_Mydb`:

```
BOOL CShoppingListDoc::CloseDB()
{
    m_Mydb.Close();
    return TRUE;
}
```

AddRecord() The next method you'll create is the `AddRecord()` method. This is the function that will receive the data from the Add Item dialog and write it to the database. Your first job is to convert the float values representing the stores' prices into strings, because the CE database engine does not support floating point values. The quickest and easiest way to perform this conversion is via `swprintf()`, as shown below:

```
BOOL CShoppingListDoc::AddRecord(CString sDescription, int nQTY, float
fStore1, float fStore2, float fStore3, float fStore4)
{
    WCHAR wcDescription[255], wcStore1[10], wcStore2[10], wcStore3[10],
wcStore4[10];

    swprintf(wcDescription, _T("%s"), sDescription);
    swprintf(wcStore1, _T("%.2f"), fStore1);
    swprintf(wcStore2, _T("%.2f"), fStore2);
    swprintf(wcStore3, _T("%.2f"), fStore3);
    swprintf(wcStore4, _T("%.2f"), fStore4);
}
```

NOTE

CE 2.11 has added support for floats as a standard datatype but, as of this writing, that support has not been added to MFC.

Next, declare an object of type `CCeDBRecord`:

```
CCeDBRecord rec;
```

And declare a six-cell array of type `CCeDBProp`, which will hold all of the information about our record's properties:

```
CCeDBProp props[6];
```

Then, create and initialize all of the `CCeDBProp` objects in that array:

```
props[0] = CCeDBProp(wcDescription, PROP_DESCRIPTION);  
props[1] = CCeDBProp((USHORT)nQTY, PROP_QTY);  
props[2] = CCeDBProp(wcStore1, PROP_STORE1);  
props[3] = CCeDBProp(wcStore2, PROP_STORE2);  
props[4] = CCeDBProp(wcStore3, PROP_STORE3);  
props[5] = CCeDBProp(wcStore4, PROP_STORE4);
```

You then add these properties to your `CCeDBRecord` object, `rec`:

```
rec.AddProps(props, 6);
```

Now, write the record out to the database:

```
if (!m_Mydb.AddRecord(&rec))  
    return FALSE;
```

Finally, clean up your used memory by calling the `DeleteAllProps()` method of `rec`:

```
rec.DeleteAllProps();  
return TRUE;  
}
```

Now that you've managed to implement the Document part of your database work, let's take a look at the second half of the equation and see what it takes to get the database to integrate into the View portion of your application.

Using the *CCeDB* Classes: Database as View

Of course, just making the database your document isn't quite enough. You also have to integrate this new type of document into your application from the View side of the application. This means writing new code to populate the `ListView` control and handle deletion of records. It also means writing a slightly

different handler for the user inserting a new record. Those three methods of your `CShoppingListView` class are

- `UpdateListBox()`
- `OnDelete()`
- `OnInsert()`

Let's take a look at each of these methods now and see exactly how the database Document interacts with your View.

`UpdateListBox()` In the file-based version of the Shopping List application, `UpdateListBox()` was the function responsible for reading the data out of the file and then populating the `ListView`. In the database-aware version, however, `UpdateListBox()` will be reading data in from the database object and populating the `ListView` with the records.

The first thing you need to do here is clear out the `ListView` so that you can start fresh:

```
void CShoppingListView::UpdateListBox()
{
    m_ListBox.DeleteAllItems();
}
```

Next, declare some local variables to be used as temporary placeholders for the data as you read it from the database and store it momentarily before putting into the `ListView` control:

```
CString  sProductName, sTemp;
int      nQty;
CString  sStore1Price;
CString  sStore2Price;
CString  sStore3Price;
CString  sStore4Price;
```

Then declare a `CCeDBRecord`, which you'll be using to access the data:

```
CCeDBRecord rec;
```

Next, declare pointers to `CCeDBProp` objects, so that you have a means of accessing the individual properties once they're read into your `CCeDBRecord` object, which you just declared above:

```
CCeDBProp *pPropDescription,
          *pPropQty,
```



```
*pPropStore1,  
*pPropStore2,  
*pPropStore3,  
*pPropStore4;
```

The first action is to close and then reopen the database using the `CloseDB()` and `OpenDB()` methods of our `Document`:

```
GetDocument()->CloseDB();  
GetDocument()->OpenDB();
```

Next, seek to the first record, using the `SeekFirst()` method of your `Document`'s `m_Mydb` member object:

```
GetDocument()->m_Mydb.SeekFirst();
```

Next, set the `m_bAutoSeekNext` flag of `m_Mydb` to indicate that you want the record pointer to be automatically advanced to the next record each time you read a record:

```
GetDocument()->m_Mydb.m_bAutoSeekNext = TRUE;
```

Now, retrieve the number of records in the database by calling the `GetNumRecords()` method of `m_Mydb`:

```
int nNumOfRecs = 0;  
nNumOfRecs = (int)GetDocument()->m_Mydb.GetNumRecords();
```

Then reserve that number of rows in your `ListView` control:

```
for (int t = 0; t <= nNumOfRecs; t++)  
{  
    m_ListBox.InsertItem(t, NULL);  
}
```

Now you can begin to populate the `ListView` with the data in the database. Do this via a `for` loop based on the record count we retrieved earlier:

```
for (int i = 0; i < nNumOfRecs; i++)  
{
```

The first thing to do is to read the current record from the database:

```
    GetDocument()->m_Mydb.ReadCurrRecord(&rec);
```

TIP

This is where the `m_bAutoSeekNext` flag comes in: each time you call `ReadCurrRecord()`, the record pointer is automatically positioned on the next record. You don't have to worry about it at all.

Now, extract `CCeDBProp` objects from the `CCeDBRec` object (`rec`) using the constants you declared at the beginning of the program:

```
pPropDescription = rec.GetPropFromIdent(PROP_DESCRIPTION);
pPropQty         = rec.GetPropFromIdent(PROP_QTY);
pPropStore1     = rec.GetPropFromIdent(PROP_STORE1);
pPropStore2     = rec.GetPropFromIdent(PROP_STORE2);
pPropStore3     = rec.GetPropFromIdent(PROP_STORE3);
pPropStore4     = rec.GetPropFromIdent(PROP_STORE4);
```

Then extract the actual property values into your local variables, which you declared earlier in this function:

```
sProductName = pPropDescription->GetString();
nQty         = pPropQty->GetUShort();
sStore1Price = pPropStore1->GetString();
sStore2Price = pPropStore2->GetString();
sStore3Price = pPropStore3->GetString();
sStore4Price = pPropStore4->GetString();
```

Next, add the items to the list box, one column at a time:

```
m_ListBox.SetItemText(i, 0, sProductName);
sTemp.Format(_T("%i"), nQty);
m_ListBox.SetItemText(i, 1, sTemp);
m_ListBox.SetItemText(i, 2, sStore1Price);
m_ListBox.SetItemText(i, 3, sStore2Price);
m_ListBox.SetItemText(i, 4, sStore3Price);
m_ListBox.SetItemText(i, 5, sStore4Price);
}
```

Finally, just as you did with the file-based version, create a row indicating the total cost for each store:

```
CreateTotals();
}
```

OnDelete() The `CShoppingListView`'s `OnDelete()` method has changed considerably from that of the file-based version of your application. Now, rather than

just removing the item from an array, you're going to immediately delete it from the database.

The first thing to do is to ensure that the currently selected item in the ListView represents a valid record. You can do this by testing that the index of that item—the number of records between the selected record and the first item in the ListView—is less than the number of records in the database and greater than or equal to 0:

```
void CShoppingListView::OnDelete()
{
    if (m_nListBoxIndex <= GetDocument()->m_Mydb.GetNumRecords() &&
        m_nListBoxIndex >= 0)
    {
```

If it is, tell your Document's `m_Mydb` to seek to that record in the database:

```
        CCeDBRecord rec;
        GetDocument()->m_Mydb.SeekToIndex(m_nListBoxIndex);
```

Then ask the user to confirm that they want to delete the record:

```
        if (AfxMessageBox(_T("Are you sure you want to delete "
            + m_ListBox.GetItemText(m_nListBoxIndex, 0) + "
            ?"), MB_OKCANCEL) == IDOK)
        {
```

If they do, go ahead and delete the record from the database by calling the `DeleteCurrentRecord()` method of your Document's `m_Mydb`:

```
        GetDocument()->m_Mydb.DeleteCurrRecord();
        UpdateListBox();
    }
}
}
```

OnInsert() The database-aware `OnInsert()` has changed very little from the file-based version of the function. You still use it to launch the Add Item dialog:

```
void CShoppingListView::OnInsert()
{
    CInsertDlg dlg;
    if (dlg.DoModal() == IDOK)
    {
```

You also still retrieve the values from the dialog and store them into local variables:

```
int      nQty = dlg.m_nQty;
CString  sProductName = dlg.m_sPdtName;
float    fStore1Price = dlg.m_fStore1;
float    fStore2Price = dlg.m_fStore2;
float    fStore3Price = dlg.m_fStore3;
float    fStore4Price = dlg.m_fStore4;
```

But instead of adding these values to a set of arrays, you then call the `AddRecord()` method of your Document class, passing in the local variables we just created:

```
        GetDocument()->AddRecord(sProductName, nQty, fStore1Price,
        fStore2Price, fStore3Price, fStore4Price);
    }
    UpdateListBox();
}
```

And that's how to integrate databases into the View side of the Document/View architecture. Of course, the various pieces of making the database work within the Document/View architecture will vary with each application's needs, but this does provide you with a good model to follow.

Now that you know all about the new classes of MFC for CE, let's move on to the classes that have been altered in some way to conform to the CE operating system.

Modified Classes

It would seem that all of MFC's classes, including `CString`, have been modified some way when they were ported to the CE operating system. Thankfully, most of these changes are minor and should not affect us. Although space does not permit a full listing of each of the changed classes and their changed elements, we will review some of the more common classes and how their changes might affect us.

The modified classes fall into two categories:

1. Classes that lost some functionality
2. Classes that gained some functionality

Classes That Lost Functionality

Most of the modified classes lost some degree of functionality. After all, CE as an operating system doesn't have as much functionality as Windows 98/NT, so it only makes sense that some classes would lose a feature or two. The more important classes that have lost some functionality are

- `CFrameWnd`
- `CWnd`
- `CDC`

CFrameWnd

Most of the functionality that `CFrameWnd` has lost relates to the fact that CE itself does not offer support for dockable toolbars. For instance, of the 13 methods no longer supported by `CFrameWnd`, 7 are directly related to the lack of dockable toolbars. The remaining unsupported methods can also be traced directly to some feature of Windows 98/NT that is not supported by CE.

CWnd

Similarly, the methods that are not supported by `CWnd` are also closely tied into missing features of the CE operating system. There are 97 missing functions in all! While that seems daunting at first, many of these functions probably won't affect you directly. In fact, 20 of these missing functions are directly related to the fact that most CE devices don't have a mouse.

CDC

Almost as bad as `CWnd` is `CDC`, which has lost 86 of its functions. Here again, most of these functions do not work under CE simply because the operating system does not offer them.

Classes That Gained Functionality

While many classes lost some features, a few classes actually gained functionality. It seems strange, but there are actually some features offered by CE that

Windows 98/NT doesn't offer. The more important classes that have gained functionality are

- `CFrameWnd`
- `CWnd`

CFrameWnd

`CFrameWnd` actually gained functions for the same reason it lost them! Although CE doesn't offer dockable toolbars, it does offer `CommandBars`. So, `CFrameWnd` picked up all of the functions for dealing with `CommandBars`:

- `AddAdornments()`
- `AddBitmap()`
- `AddComboBoxString()`
- `GetComboCount()`
- `GetComboCurSel()`
- `InsertButtons()`
- `InsertComboBox()`
- `InsertMenu()`
- `SetComboCurSel()`

TIP

For more information on how to use these new functions, see Chapter 7.

CWnd

As a class, `CWnd` didn't pick up any new members. But it did get something very important under Windows CE: the ability to process the `WM_HIBERNATE` message. As we saw in Chapter 2, the `WM_HIBERNATE` message is sent by the operating system under low-memory conditions as a way to request that running applications free some memory. Of course, this new feature is relatively transparent to you, except for the fact that you can now create your own handler for `WM_HIBERNATE` within the context of a `CWnd` object.

Missing Classes

Some classes are gone completely. In most of these cases, the class is unsupported because the underlying control does not exist in Windows CE. For instance, there is no Font Selection Dialog, so `CFontDialog` is not supported. Likewise, there is no Printer Setup Dialog in Windows CE, so `CPrintSetupDialog` is not supported. And, of course, all of the Windows 98/NT-specific common controls that don't exist under CE (`AnimateCtrl`, `CheckListBox`, etc.) are also not supported by MFC.

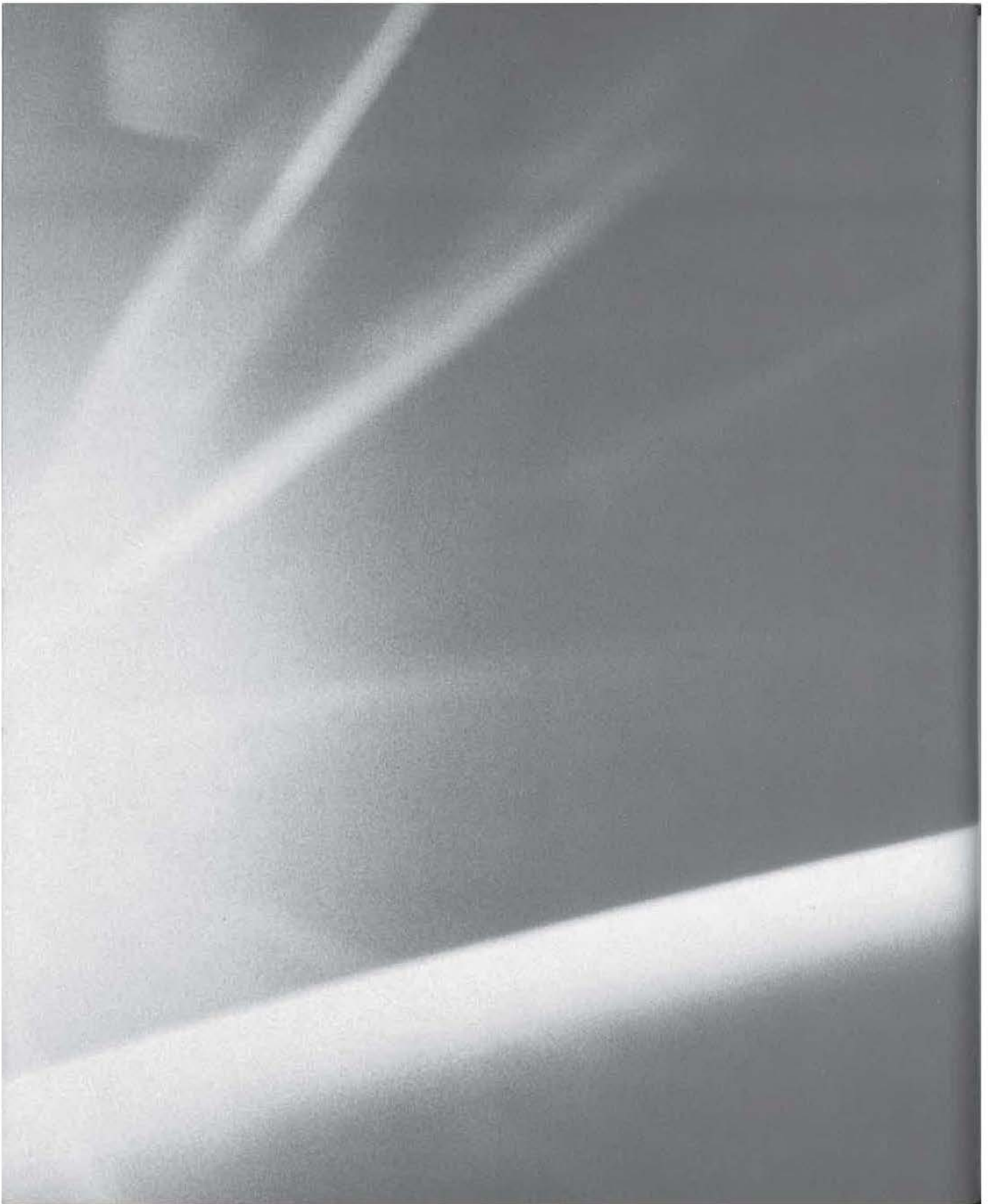
NOTE

`CPrintDialog` is still supported by MFC 2.10; it's only `CPrintSetupDialog` that doesn't exist under Windows CE.

Summary

In this chapter, we learned about the new classes that MFC offers for CE. Specifically, we saw that `CCeSocket` is almost exactly the same as `CAsyncSocket`. In addition, we learned how to work with databases and the Windows CE database engine. Then we took a brief look at some of the classes that have changed a bit from their Desktop-based counterparts. Finally, we learned what to expect in the way of classes that aren't supported at all under CE.





CHAPTER

SEVEN

7

Real MFC Applications Ported to CE

- The Shopping List Application
- Mechanical Issues of Porting
- Toolbars and Status Bars
- Printing Support
- Optimizing for CE

In the last chapter, you examined what sort of things were and were not supported by MFC for CE. In this chapter, you'll be converting a real MFC application to Windows CE. When you're done, you'll have an understanding of what's involved in converting even the simplest of MFC Desktop applications to the CE platform.

First you'll take a look at the project as it exists now. Then, starting with the simpler tasks, you'll work your way through the conversion to CE to get a no-frills version of the application running on a CE device. Next, you'll look at porting some of the optional features and at optimizing the user interface and the code for the Windows CE platform. Finally, you'll see how to optimize your program for the CE platform and why it may be necessary to drop or add certain features.

The Shopping List Application

The Shopping List application that you'll be porting is a program designed to help you keep track of your purchases so you can determine which store, if any, really has better prices on the items you want to buy. The program itself is designed primarily for supermarkets, where a user might have time to collect prices from various stores by shopping at each of them a few times. They can then enter prices of various items at a variety of stores and sit back and let the program show them how much they actually spent. The main window of this application is shown in Figure 7.1.

FIGURE 7.1:
The Desktop version of the Shopping List application

	Qty	Product Name	Got	Lucky	Vons	Ralphs
1	1	Soda	<input checked="" type="checkbox"/>	1.99	2.15	2.35
2	1	Cereal	<input checked="" type="checkbox"/>	1.35	2.00	1.15
3	5	Bottled Water	<input type="checkbox"/>	0.59	0.62	0.68
4			<input type="checkbox"/>			
5			<input type="checkbox"/>			
6			<input type="checkbox"/>			
7			<input type="checkbox"/>			
8			<input type="checkbox"/>			
9			<input type="checkbox"/>			
10			<input type="checkbox"/>			

You'll note that most of the user interface of this program is taken up with a large grid. The user can navigate the grid with the keyboard or the mouse, entering quantity, item description, and the prices at each of the stores. This data can then be serialized out to a text file whenever the user clicks the Save button. The user also has the option of changing the names of the supermarkets/stores at which they're doing their price comparisons.

NOTE

The grid you see in Figure 7.1 is not part of MFC itself; it is a third-party MFC library. This grid handles its own serialization and contains built-in functionality for simple operations such as summing a column.

At first glance, the application looks pretty much like it could run on a CE device without any modifications at all. However, there are a number of changes you'll have to make to this application just to get a minimal version working on CE. The areas of this application that you'll have to either eliminate or alter to get your application running on a CE device are

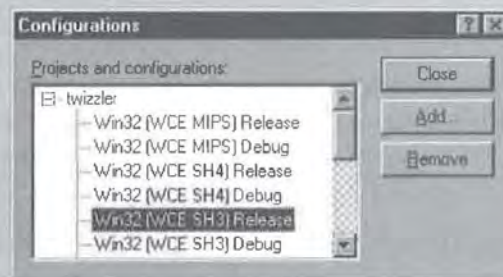
- Toolbars and status bars
- Printing support
- The grid

When you first look at this list, you might think that you'd need to change everything that makes this application what it is. After all, the items on this list basically represent the entire graphical portion application, and, especially without a gridlike control to display the pricing information to the user, the Shopping List application seems at a definite disadvantage.

But it's not as bad as it first appears. Some of these items are easy to fix and involve nothing more than choosing a different option in the WCE MFC App-Wizard. Now that you have an idea of what portions of your Shopping List application might need some reworking, let's take each item and begin to port the application, performing the tasks in order of complexity.

How, Exactly, Do You Get a Desktop MFC Application to Compile for CE?

In Toolkit for VC++ 5, it was very easy to compile an existing Desktop application for Windows CE. All you had to do was choose Build>Configurations from the menu, click the Add button on the dialog shown below, and add a CE-based configuration. Assuming your code was written in such a way that all of your classes were available under CE, a port was merely one build away.



Even if major pieces of code had to change, this was still the simplest way to start a port of a desktop program to CE. (Although if you knew you were going to be changing major pieces of code, it was probably advisable to make a copy of your project and then begin the work of porting it.)

If you were going to do a full port, you'd simply add all of the CE configurations that you needed and then remove the Desktop configurations. That's all it took to get a 100 percent CE-based project from your existing Desktop project.

As you may know, however, VC++ 6 does not allow projects to contain both Desktop and CE configurations. The unfortunate truth is that, unless Microsoft announces some kind of patch or special tool to make this possible, you'll have to re-create your application from scratch, cutting and pasting code and files as appropriate.

For this reason, it may actually be advisable to keep a copy of VC++ 5 and the earlier toolkit on one of your machines. There is simply no other way to create an application that will build for both Desktop and CE platforms. Further, there is no easier way to get started with a port of an existing program to CE.

In the case of the application in this chapter, you will employ the cut-and-paste method of porting so that some of the changes are more manageable. This will apply to many applications that require major changes; the cut-and-paste method will probably be easier in the long run, as opposed to attempting to compile your project for both Desktop- and CE-based platforms at once.

Mechanical Issues of Porting

Some of the items that won't port are easier to fix than others. We'll work from the easiest to fix to the most difficult.

Toolbars and Status Bars

The easiest items to fix, at least in terms of actual coding required, are the status bar and the toolbar. In fact, fixing these items requires that you do nothing more than correctly check off some of the options on the WCE MFC AppWizard. However, deciding *which* options to check off is sometimes not that simple. As you learned in Chapter 5, there are a number of subtle differences between MFC for CE version 2.0 and MFC for CE version 2.1.

Probably, the most notable of these differences is the multiple types of command bars. You'll recall that of the four possible command-bar types offered by the WCE MFC AppWizard, only one of them (the fourth) can compile and run on both the PPC, HPC, and HPC/Pro devices. The first three will only run on PPC and HPC/Pro devices. In the current context of having to port a status bar and a toolbar, this 2.0/2.1 version difference affects us in several ways.

First and foremost, you must choose which devices you really want to target with your application. If you'd like to run it on all of the three major CE platforms, then you must choose the fourth type of command bar. However, by doing this, you are explicitly setting your application to be 100 percent compatible with MFC 2.0 and thus eliminating the option of having a status bar for your application, since status bars are only supported in MFC 2.1. How frustrating! This means that some elements of the design and visual appearance of your program may be limited simply because you want to reach the widest possible market for your program.

MFC 2.0 vs. MFC 2.1

So, why are there different versions of MFC on an operating system and platform that are basically brand new?

With the release of CE 2.0—the version that’s running on all of the Handheld and Palm-size PCs currently on the market—there was also a 2.0 release of MFC. Officially, the 2.0 version of MFC would work only on the Handheld devices, not the Palm-size.

Of course, that didn’t stop a large number of developers from copying the MFC DLLs to their PPCs and trying to run their applications anyway. What they found was that although their MFC programs could and did run, they were rather slow at times.

When Microsoft released Windows CE version 2.11—that’s the version that runs on HPC/Pro devices, such as the HP Jornada—they also released a new version of MFC, dubbed 2.1.

One of the major advantages of this new MFC is that you can now officially use it to build programs for Palm-size devices. In addition, MFC 2.1 added features such as new command bar styles, status bars, and printing support. The following table summarizes the features in each version.

MFC Version	Devices supported	Toolbars	Status Bar	Printing
2	HPC	1 style	No	No
2.1	PPC, HPC/Pro	4 styles	Yes	Yes

As you can see, the configuration that gives you the most flexibility in terms of devices to target also takes away your ability to print and/or provide a status bar. Which configuration you decide to use comes down to whether support for printing is more important than being able to sell your product to owners of a standard HPC device.

In the case of the project that we’re porting now, let’s opt to support the PPC, HPC, and HPC/Pro devices, even though this means that you’ll lose your status bar. To support all three devices, make sure that you select the fourth command bar type when you’re going through the WCE MFC AppWizard.

As it turns out, the status bar isn’t a great loss. You weren’t really using it for much, so letting it go does not really affect your application. In practical terms, all

it means is that you must remove the following lines of code from your CMainFrame's OnCreate() event:

```
if (!m_wndStatusBar.Create(this) ||
    !m_wndStatusBar.SetIndicators(indicators,
    sizeof(indicators)/sizeof(UINT)))
{
    TRACE0("Failed to create status bar\n");
    return -1;    // fail to create
}
```

It's worth noting that, with one simple choice in a wizard, we've managed to deal with two of the items on the list of possible problem areas that appeared earlier in the chapter. As far as the command bar is concerned, the original code to create the toolbar in the Desktop application is called from within the CMainFrame's OnCreate() event handler. There, you create the toolbar object and populate it with the buttons contained in the IDR_MAINFRAME resource, as follows:

```
if (!m_wndToolBar.Create(this) ||
    !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
{
    TRACE0("Failed to create toolbar\n");
    return -1;    // fail to create
}
```

Then, set the style of the toolbar, so that it will give the ToolTip hints and be resizable:

```
m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() |
    CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);
```

Finally, make it possible for the user to dock and undock the toolbar:

```
m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
EnableDocking(CBRS_ALIGN_ANY);
DockControlBar(&m_wndToolBar);
```

In the Windows CE version of your CMainFrame::OnCreate(), the code to create the Command Bar looks somewhat different. All you have to do here is call the CFrameWnd::InsertButtons() member function, as follows:

```
if (!InsertButtons( g_tbSTDButton,
    sizeof(g_tbSTDButton)/sizeof(TBBUTTON),
    IDR_MAINFRAME,
    6 ))
```



```

    {
        TRACE0("Failed to add command bar buttons\n");
        return -1;
    }

```

In the above code, `g_tbSTDButton` is an array of properties that describe how the buttons are to be displayed and `IDR_MAINFRAME` is the resource ID for the Command Bar images.

WARNING

The `InsertButtons()` call can be the cause of some very odd behavior if it's not called with the correct number of buttons specified as the final parameter. If the value is less than the number of actual buttons, the X (or Close) button of the application will have the same image as the last button on the toolbar. If the value is greater than the number of actual buttons, the X button will have absolutely no image at all!

More on *CMainFrame*: *InsertButtons()*

`InsertButtons()` takes an array of type `TBBUTTON`. `TBBUTTON` is a structure that defines properties for each of the buttons to be displayed and is defined as follows:

```

typedef struct _TBBUTTON {
    int iBitmap;
    int idCommand;
    BYTE fsState;
    BYTE fsStyle;
    DWORD dwData;
    int iString;
} TBBUTTON, NEAR* PTBBUTTON, FAR* LPTBBUTTON;
typedef const TBBUTTON FAR* LPCTBBUTTON;

```

Of the items in the structure, the most important ones to you will be

- `iBitmap`, which specifies the order of the button/image in the Command Bar

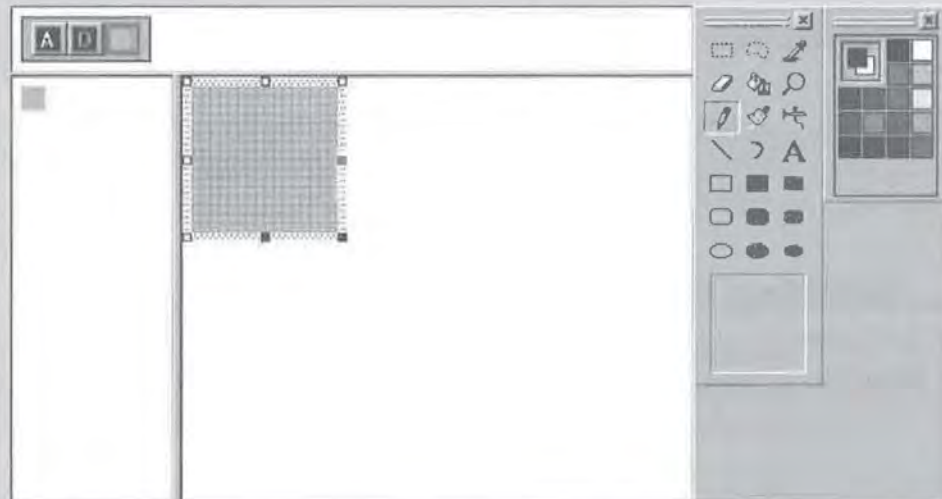
Continued on next page

- `idCommand`, which makes your Command Bar button act like a true button in that the value specified in `idCommand` will be passed to the application's `WM_COMMAND` handler when the button is clicked
- `fsState`, which specifies that the state (hidden, enabled, etc.) will typically be 0 (disabled), `TBSTATE_ENABLED`, or `TBSTATE_HIDDEN`
- `fsStyle`, which specifies whether you're creating a checkbox, a button, or a separator

The rest of the values in the structure are simply set to -1 or 0, as shown below. A standard `TBBUTTON` array would look something like this:

```
static TBBUTTON g_tbSTDButton[] = {
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0},
    {0, ID_FILE_NEW, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {1, ID_FILE_OPEN, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {2, ID_FILE_SAVE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, -1},
    {3, ID_EDIT_CUT, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {4, ID_EDIT_COPY, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {5, ID_EDIT_PASTE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, -1},
    { 0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0}
};
```

If you had custom buttons to add, you'd simply draw them in the Resource Editor, as shown below, and then add their properties to this array.



Continued on next page

The modified TBBUTTON array, then, would look like this:

```
static TBBUTTON g_tbSTDButton[] = {
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0},
    {0, ID_FILE_NEW, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {1, ID_FILE_OPEN, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {2, ID_FILE_SAVE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, -1},
    {3, ID_EDIT_CUT, 0, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {4, ID_EDIT_COPY, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {5, ID_EDIT_PASTE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, -1},
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0},
    //A new button!
    {6, ID_NEWBUTTON, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
};
```

Then, you'd simply update the value you passed to `InsertButtons()` to reflect the new button count:

```
InsertButtons( g_tbSTDButton,
              sizeof(g_tbSTDButton)/sizeof(TBBUTTON),
              IDR_MAINFRAME,
              7 ); //now there are 7 buttons.
```

This simple change is, in fact, one you'll be making a little later in the process of porting your Shopping List application.

Although you did have to drop a status bar, the port here was relatively simple. However, you'll see that, as you progress into the application, your next porting issues may not be so easy.

Printing Support

The second major consequence of choosing to target all three of the major CE platforms is that we've now lost the ability to offer printing support in your application. This is yet another of the subtle differences between MFC 2.0 and 2.1. If you want to support all three CE-based device types, you won't be able to use MFC's printing support.

However, there are a few possible solutions to this dilemma. The first is that you could drop printing support altogether. After all, the majority of CE devices do not have access to a printer, and there is only one printer that's even specifically targeted to the CE market.

TIP

The printer specifically targeted to the CE market is the PocketJet II from Pentax Technologies. For more information, visit <http://www.pentaxtech.com>.

The second option is for you to create your own printing object that would encapsulate all of the API's own printing routines. The problem with this, of course, is that it can be quite time-consuming, and requires you to learn more about the API's methods of printing than you probably want to know.

These two options might seem a little bleak, but there is a third that is ideal for certain types of applications. If the application in question is already being tailored to a wireless environment, the solution to consider is one provided by Odyssey Software (<http://www.odysseysoftware.com>). As part of an extensive package called CEFusion that exposes many of the NT/BackOffice products to CE developers, Odyssey offers a CE-based client product that makes it possible for developers to write code to print to any Windows NT-based network printer. The kind of printing that CEFusion allows you to do is fairly basic, but if the application really needs to be able to print and you're already considering a wireless environment, this could be the ideal solution.

The Grid

The most complex change you'll probably have to make is to the grid that now occupies most of the main form of your application. Because you're using a third-party MFC class that hasn't yet been ported to CE, you're in a tight spot with this issue.

Of course, if the third-party grid existed on CE, you'd be all set, and the porting process would pretty much be done by now. Instead, what you'll have to do is to select a replacement grid or control that you can use in the same way. As far as selecting a new grid control is concerned, you have two options:

- The Microsoft ActiveX Grid Control
- A replacement control

Unfortunately, there's no ideal solution here. Whatever you choose will likely require serious reworking of your code. While that's never really a pleasant task, it may still be the best—if not the only—way to successfully port your application.

Microsoft's ActiveX Grid Control

Microsoft provides a simple, no-frills grid control in the form of an ActiveX control. This control is available either from the ActiveX control pack, which is freely downloadable from the Microsoft Web site, or from the Windows CE toolkit for Visual Basic 6. As grid controls go, the ActiveX grid control is more than adequate for your needs. There are, however, a few disadvantages to using it.

The first disadvantage is that it's really designed for Visual Basic. That's not to say that you couldn't use it, but VC++ doesn't come with the Control Manager tool for registering ActiveX controls on the device or in the emulator. That means that you have to manage the registration of this component yourself instead of having VB take care of it for you. Plus, assuming that you don't have the latest toolkit for Visual Basic, you'd have to use the version of the control that comes with the ActiveX control pack. Because the ActiveX control pack was created before the development of CE 2.11, the ARM and SH4 processors aren't supported, which means that you would not be able to target some of the HPC/Pro devices.

NOTE

The Control Manager tool from the Visual Basic toolkit is covered in more detail in Chapter 8.

The other problem with using the Microsoft Grid ActiveX control, or any ActiveX control, for that matter, is the additional disk space it requires. The Grid control itself is about 100K in size, and on devices with limited resources, 100K can represent a good deal of space. Instead of using the ActiveX grid, then, let's look at a solution that works well with VC++ and doesn't use too much additional space.

A Replacement Control

When it comes to finding a replacement for your grid that won't occupy any additional space and will still allow you the same or similar functionality, one control stands out: the CListView control. ListView controls can perform almost

all of the functions that a grid can perform, and using a `ListView` doesn't add 100K to the size of the files you'll have to distribute.

However, `ListView` controls will require you to write code to handle the editing and display of the user's data. Previously, of course, the grid handled all of this work itself. The basic tasks that you've got to handle on your own are

- Managing the user's data while the program is running
- Displaying the user's data
- Allowing the user to enter and delete data

Let's take a look at how you can implement each one and finish getting your application ported to CE.

Managing the User's Data While the Program Is Running The task at hand is to store all of the user's data as they enter it, so that you can display it to the `CListView` control, serialize it out to a file, etc. To accomplish this, you'll need to add several arrays to the `CDocument` class. In all, there will be seven arrays:

- An array to maintain the names of the stores on your shopping list:
`CStringArray m_sStoreNames;`
- An array to store the quantities of the items you'll be price comparing:
`CArray<int, int> m_Qty;`
- An array to store the names of the actual products you'll be price comparing:
`CStringArray m_sProductName;`
- Four arrays to store the prices for each item at each of the four stores on your shopping list:
`CArray<float, float> m_fStore1Price;`
`CArray<float, float> m_fStore2Price;`
`CArray<float, float> m_fStore3Price;`
`CArray<float, float> m_fStore4Price;`

With these arrays, you can store all of the user's data while the program is running. Now, you must handle the task of displaying this data to the user.

Displaying the User's Data Given your list of arrays, your first task whenever the user requests a new document is to set up default values for the store names and clear out the other arrays. In the `CShoppingListDoc::OnNewDocument()` event, first clear out the `m_sStoreNames` array:

```
m_sStoreNames.RemoveAll();
```

Then, initialize the array with default values:

```
m_sStoreNames.Add(_T("Store 1"));
m_sStoreNames.Add(_T("Store 2"));
m_sStoreNames.Add(_T("Store 3"));
m_sStoreNames.Add(_T("Store 4"));
```

Next, clear out the other arrays, so that everything is reset for the next shopping list:

```
m_fStore1Price.RemoveAll();
m_fStore2Price.RemoveAll();
m_fStore3Price.RemoveAll();
m_fStore4Price.RemoveAll();
m_Qty.RemoveAll();
m_sProductName.RemoveAll();
```

The next step is to set up the column headers on the `CListView` control, so that the data has some meaning to the user and they know what it is they're looking at. Do this in the `CShoppingListView::CreateListBoxHeader()` method. First, retrieve the store names from the `m_sStoreNames` `CStringArray` and store those names in local variables:

```
CString sStore1Name = GetDocument()->m_sStoreNames[0];
CString sStore2Name = GetDocument()->m_sStoreNames[1];
CString sStore3Name = GetDocument()->m_sStoreNames[2];
CString sStore4Name = GetDocument()->m_sStoreNames[3];
```

Then, delete and then re-create each column as you change its heading:

```
m_ListBox.DeleteColumn(0);
m_ListBox.InsertColumn( 0, _T("Product Name") , LVCFMT_LEFT, 175 );
m_ListBox.DeleteColumn(1);
m_ListBox.InsertColumn( 1, _T("Qty") , LVCFMT_LEFT, 50 );
m_ListBox.DeleteColumn(2);
m_ListBox.InsertColumn( 2, sStore1Name , LVCFMT_RIGHT, 95 );
m_ListBox.DeleteColumn(3);
m_ListBox.InsertColumn( 3, sStore2Name , LVCFMT_RIGHT, 95 );
```

```
m_ListBox.DeleteColumn(4);
m_ListBox.InsertColumn( 4, sStore3Name , LVCFMT_RIGHT, 95 );
m_ListBox.DeleteColumn(5);
m_ListBox.InsertColumn( 5, sStore4Name , LVCFMT_RIGHT, 95 );
```

You do this to ensure that the heading text is properly refreshed each time.

NOTE

This is also where you could change the width of a column if you wanted to; just change the value of the fourth parameter in the calls to `InsertColumn()`.

The next step is to add code to get the data from these arrays that you created into your `CListView` control. This is done in the `CShoppingListView::UpdateListBox()` method. First, clear out the `CListView`:

```
m_ListBox.DeleteAllItems();
```

Next, pre-reserve all of the rows you'll need in the `CListView`:

```
//how many product names do we have?
int nIndex = GetDocument()->m_sProductName.GetSize();
for (int t = 0; t <= nIndex; t++){
    m_ListBox.InsertItem(t, NULL);
}
```

Then loop through the arrays, adding their information to your replacement grid, `CListView`:

```
for (int i = 0; i < nIndex; i++)
{
```

Here, you retrieve the array values and store them in local variables temporarily:

```
nQty          = GetDocument()->m_Qty[i];
fStore1Price  = GetDocument()->m_fStore1Price[i];
fStore2Price  = GetDocument()->m_fStore2Price[i];
fStore3Price  = GetDocument()->m_fStore3Price[i];
fStore4Price  = GetDocument()->m_fStore4Price[i];
sProductName  = GetDocument()->m_sProductName[i];
```

Then, add the product name to the 0th column:

```
m_ListBox.SetItemText(i, 0, sProductName);
```


Now, add the quantity to the first column:

```
sTemp.Format(_T("%i"), nQty);
m_ListBox.SetItemText(i, 1, sTemp);
```

Add the first store's price to the second column:

```
sTemp.Format(_T("%.2f"), fStore1Price);
m_ListBox.SetItemText(i, 2, sTemp);
```

You repeat this process for each of the remaining pieces of data:

```
sTemp.Format(_T("%.2f"), fStore2Price);
m_ListBox.SetItemText(i, 3, sTemp);

sTemp.Format(_T("%.2f"), fStore3Price);
m_ListBox.SetItemText(i, 4, sTemp);

sTemp.Format(_T("%.2f"), fStore4Price);
m_ListBox.SetItemText(i, 5, sTemp);

}
```

So far, so good! Now let's move on to allowing the user to enter and delete data.

Allowing the User to Enter and Delete Data Your first task is to allow the user to enter data into the CListView control. However, instead of letting them simply click on the CListView and add whatever they'd like, add a button to the toolbar that, when clicked, will pop up a dialog box. This dialog will ask the user to enter information about the product that they're adding to their shopping list. This dialog-based approach allows for greater validation on your part because it's easier to limit the number of ways that data can be entered into the system. The first step is to go to the Resource Editor and create a new button that will be for adding records. For the sake of simplicity, just type a large letter *A* on the button face, as shown in Figure 7.2.

FIGURE 7.2 :

The Add button



The next step to getting your new button loaded onto the Command Bar is to add it to the TBBUTTON array discussed earlier. So, add an entry to that array as follows:

```
[6, ID_INSERT, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1]
```

Then you've got to change one value in your call to `CMainFrame::InsertButtons()`. Previously, you had:

```
if (!InsertButtons( g_tbSTDButton,
    sizeof(g_tbSTDButton)/sizeof(TBBUTTON),
    IDR_MAINFRAME,
    6 ))
//...
```

Now, because you've added an additional button, you must increase the number of buttons (six, in the above code) to seven. Therefore, your call to `InsertButtons()` should now read:

```
if (!InsertButtons( g_tbSTDButton,
    sizeof(g_tbSTDButton)/sizeof(TBBUTTON),
    IDR_MAINFRAME,
    7 ))
//...
```

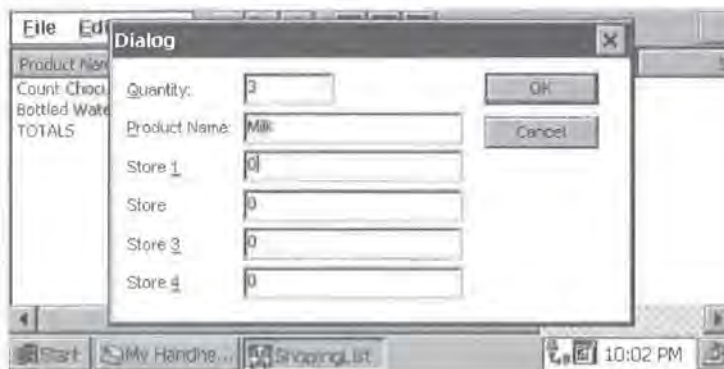
Now that you've added an Add button onto the Command Bar, you must create the Add dialog. To do this, use the Resource Editor to insert a new dialog. You can now add controls to the dialog according to the data you'll need to get from the user. In this case, the dialog should have six edit boxes:

- One edit box to record the quantity of the item being added
- One edit box to record the name of the product
- Four edit boxes to record the price of the item at each store

Figure 7.3 shows a possible design for this dialog.

FIGURE 7.3:

The Add/Insert dialog



Now create a class to act as a wrapper for the dialog. Simply double-clicking anywhere on the dialog while in the Resource Editor will do the trick. Call your class `CInsertDialog`. Then, using the Class Wizard, create member variables for `CInsertDialog` to store the data entered by the user and add code to launch the dialog in the `CShoppingListView::OnInsert()` event:

```
CInsertDlg dlg;
if (dlg.DoModal() == IDOK)
{
```

If the user clicked the OK button, go ahead and get the values that they entered and store them into local variables:

```
int    nQty = dlg.m_nQty;
CString sProductName = dlg.m_sPdtName;
float fStore1Price = dlg.m_fStore1;
float fStore2Price = dlg.m_fStore2;
float fStore3Price = dlg.m_fStore3;
float fStore4Price = dlg.m_fStore4;
```

Then add the values you just retrieved from the dialog into your arrays:

```
GetDocument()->m_Qty.Add(nQty);
GetDocument()->m_sProductName.Add(sProductName);
GetDocument()->m_fStore1Price.Add(fStore1Price);
GetDocument()->m_fStore2Price.Add(fStore2Price);
GetDocument()->m_fStore3Price.Add(fStore3Price);
GetDocument()->m_fStore4Price.Add(fStore4Price);
}
```

Finally, make sure that you indicate that the data has changed, and update the CListView control:

```
GetDocument()->SetModifiedFlag(TRUE);
UpdateListBox();
```

The next step in allowing the user to enter and delete data is to allow them to change the names of the four supermarkets/stores at which you're performing your price comparisons. Just as you did with the Add dialog, start by creating a button (using, let's say, a capital S for stores) and add it to the TBBUTTON array:

```
{7, ID_STORE_NAMES, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
```

Again, change the call to CMainFrame::InsertButtons() to reflect the additional button:

```
if (!InsertButtons( g_tbSTDButton,
    sizeof(g_tbSTDButton)/sizeof(TBBUTTON),
    IDR_MAINFRAME,
    8 )) //changed to '8' from '7'
```

To launch the dialog, create an OnStoreNames() event in your CShoppingListView class. However, unlike in the Add dialog, put some initial values into the dialog box:

```
CChgStrNameDlg dlg;
dlg.m_sStore1Name = GetDocument()->m_sStoreNames[0];
dlg.m_sStore2Name = GetDocument()->m_sStoreNames[1];
dlg.m_sStore3Name = GetDocument()->m_sStoreNames[2];
dlg.m_sStore4Name = GetDocument()->m_sStoreNames[3];
```

Then, if the user clicks the OK button, retrieve their values and store them in the m_sStoreNames array:

```
if (dlg.DoModal() == IDOK)
{
    GetDocument()->m_sStoreNames[0] = dlg.m_sStore1Name;
    GetDocument()->m_sStoreNames[1] = dlg.m_sStore2Name;
    GetDocument()->m_sStoreNames[2] = dlg.m_sStore3Name;
    GetDocument()->m_sStoreNames[3] = dlg.m_sStore4Name;
}
```

When you're all done, refresh the column headings:

```
CreateListBoxHeader();
```


At this point the only thing left to do is to add functionality to allow the user to delete items from their Shopping List. Here again, start by creating a button (with a capital *D* for delete) and adding it to the TBBUTTON array:

```
(8, ID_DELETE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1),
```

Again, change the call to `CMainFrame::InsertButtons()` to reflect the additional button:

```
if (!InsertButtons( g_tbSTDButton,
    sizeof(g_tbSTDButton)/sizeof(TBBUTTON),
    IDR_MAINFRAME,
    9 )) //changed to '9' from '8'
```

To actually delete the selected record, you'll need to create a `CShoppingListView::OnDelete()` method which will do the actual work of deleting the item out of the array.

First, make sure that the index of the selected item is valid by ensuring that it's less than the number of items available and greater than 0:

```
if (nListBoxIndex <= GetDocument()->m_sProductName.GetSize() &&
    nListBoxIndex >= 0)
{
```

Next, confirm with the user that they do want to delete the record in question:

```
if (AfxMessageBox(T("Are you sure you want to delete "
    + GetDocument()->m_sProductName[nListBoxIndex] +
    "?"),MB_OKCANCEL) == IDOK)
{
```

If they do want to delete the item, give them the various arrays to delete the values at the appropriate index:

```
GetDocument()->m_Qty.RemoveAt(nListBoxIndex,1);
GetDocument()->m_sProductName.RemoveAt(nListBoxIndex,1);
GetDocument()->m_fStore1Price.RemoveAt(nListBoxIndex,1);
GetDocument()->m_fStore2Price.RemoveAt(nListBoxIndex,1);
GetDocument()->m_fStore3Price.RemoveAt(nListBoxIndex,1);
GetDocument()->m_fStore4Price.RemoveAt(nListBoxIndex,1);
```

Finally, set the modified flag, and update the contents of the `CListView`:

```
GetDocument()->SetModifiedFlag(TRUE);
UpdateListBox();
}
```

When you're all done, you should have an application that looks something like the one shown in Figure 7.4.

FIGURE 7.4:

The final, ported application

The screenshot shows a handheld application window with a menu bar (File, Edit, Help) and a toolbar. The main content is a table with the following data:

Product Name	Qty	Score 1	Score 2
Count Chocula	1	1.99	2.37
Bottled Water (1 quart)	3	0.99	0.87
TOTALS	4	2.98	3.24

The window also shows a taskbar at the bottom with icons for 'Start', 'My Handline', and 'ShoppingList', along with a clock showing 10:03 PM.

Although it looks very different from the application you started with, you can see that the two applications are functionally very similar. Of course, this is merely a rough port. There's a lot you can do to enhance and optimize this application to be more in tune with the features and limitations of a CE device.

Optimizing for CE

You now have a port of your application to CE, and, technically, you're finished. However, there are a few steps you can take to make your application run just a little bit more smoothly on a CE device. The first one, which takes almost no effort on your part, is to limit the application to only opening one true Shopping List file. Currently, the user can create any number of shopping list files. However, given the limited storage space of a CE device and the fact that you don't really want the user to litter their CE device with dozens of shopping list files, it would be much better to simply enforce a single-file policy. Fortunately, it only takes one line of code in order to ensure that `ShoppingList.dat` is always the file opened by the application:

```
OpenDocumentFile(_T("ShoppingList.dat"));
```

You'll also want to fix the Command Bar so that it does not have any buttons relating to New and Open. To do that, first edit the resource containing the toolbar buttons, so that the first three buttons are deleted, then edit the TBBUTTON array so that it now reflects the true number of button images available.

TIP

To delete a button, either click on it and then press the delete key, or click and drag it off the screen with your mouse.

```
static TBBUTTON g_tbSTDButton[] = {
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0},
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, -1},
    {0, ID_EDIT_CUT, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {1, ID_EDIT_COPY, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {2, ID_EDIT_PASTE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, -1},
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0},
    {3, ID_INSERT, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {4, ID_STORE_NAMES, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {5, ID_DELETE, TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0, 0, -1},
    {0, 0, TBSTATE_ENABLED, TBSTYLE_SEP, 0, 0, 0, 0}
};
```

You'll notice that the first three buttons have now been removed, and the index values for the remaining buttons have all been appropriately adjusted.

Naturally, you must also alter your call to `CMainFrame::InsertButtons()`:

```
if (!InsertButtons( g_tbSTDButton,
    sizeof(g_tbSTDButton)/sizeof(TBBUTTON),
    IDR_MAINFRAME,
    6 )) //reflects loss of 3 buttons
```

This modification helps to cut down on possible excessive storage space use and does so with only a minimal loss in functionality.

There's one additional optimization that you can make here that would improve the efficiency and memory usage of your application. When you originally performed the port on this application, there were a number of occasions where, for debugging and illustration purposes, you stored values from an array into a local variable and then displayed them to the user. The step of storing the values into local variables can be entirely eliminated. In fact, to be truly low-memory aware, you should eliminate as many nonessential variables as possible,

as they are essentially memory allocations. Let's take a look at how one of these blocks of code might change.

You'll recall that in the `CShoppingListView::CreateListBoxHeader()` method you retrieve the store names from the `m_sStoreNames` `CStringArray` and put the values into local variables:

```
CString sStore1Name = GetDocument()->m_sStoreNames[0];
CString sStore2Name = GetDocument()->m_sStoreNames[1];
CString sStore3Name = GetDocument()->m_sStoreNames[2];
CString sStore4Name = GetDocument()->m_sStoreNames[3];
```

Then, you refresh the actual heading text, passing the local variables to the `InsertColumn()` method, where appropriate:

```
m_ListBox.DeleteColumn(0);
m_ListBox.InsertColumn( 0, _T("Product Name"), LVCFMT_LEFT, 175 );
m_ListBox.DeleteColumn(1);
m_ListBox.InsertColumn( 1, _T("Qty"), LVCFMT_LEFT, 50 );
m_ListBox.DeleteColumn(2);
m_ListBox.InsertColumn( 2, sStore1Name, LVCFMT_RIGHT, 95 );
m_ListBox.DeleteColumn(3);
m_ListBox.InsertColumn( 3, sStore2Name, LVCFMT_RIGHT, 95 );
m_ListBox.DeleteColumn(4);
m_ListBox.InsertColumn( 4, sStore3Name, LVCFMT_RIGHT, 95 );
m_ListBox.DeleteColumn(5);
m_ListBox.InsertColumn( 5, sStore4Name, LVCFMT_RIGHT, 95 );
```

Although this does make it easier to debug this procedure, there's no reason at all to leave the code organized in this way for the final shipping product. Instead, you'd do much better to pass the array values directly to the `InsertColumn()` method, thus eliminating the need for local variables entirely:

```
m_ListBox.DeleteColumn(0);
m_ListBox.InsertColumn( 0, _T("Product Name"), LVCFMT_LEFT, 175 );
m_ListBox.DeleteColumn(1);
m_ListBox.InsertColumn( 1, _T("Qty"), LVCFMT_LEFT, 50 );
m_ListBox.DeleteColumn(2);
m_ListBox.InsertColumn( 2, GetDocument()->m_sStoreNames[0],
LVCFMT_RIGHT, 95 );
m_ListBox.DeleteColumn(3);
m_ListBox.InsertColumn( 3, GetDocument()->m_sStoreNames[1],
LVCFMT_RIGHT, 95 );
m_ListBox.DeleteColumn(4);
```



```
m_ListBox.InsertColumn( 4, GetDocument()->m_sStoreNames[2],  
LVCFMT_RIGHT, 95 );  
m_ListBox.DeleteColumn(5);  
m_ListBox.InsertColumn( 5, GetDocument()->m_sStoreNames[3],  
LVCFMT_RIGHT, 95 );
```

If you make this simple change everywhere else it's needed in the source, you will have saved quite a bit of needless memory allocation.

One Final Surprise

Although it wasn't demonstrated in the porting of the Shopping List application, there is one additional surprise that you need to know about when you're porting to CE.

Under Windows 98 or NT, there are a number of cases where you rely on another program to do a task for you, rather than attempting to duplicate that functionality on your own. For instance, many programs that don't require a full-fledged reporting engine of their own may opt to use Microsoft Word as a reporting engine. Typically, the developer will create template documents for Word. Then, via a service known as OLE Automation, it's possible to launch Word in a hidden state, pass it values to insert into the template document, and even ask it to print the document for you. All this is done without the user's knowledge that Word has been invoked at all. For instance, in your `CWinApp's InitInstance()` method, you might actually launch a hidden copy of Excel with code that looks something like this:

```
m_pExcelWS = new IWorksheetFunction;  
ASSERT (m_pExcelWS!= NULL);  
if (!m_pExcelWS->CreateDispatch("Excel.Application"))  
    return AfxMessageBox("Failed to Connect, Please make sure you have  
Excel 97 Installed");
```

The problem is that under CE, the programs that are most commonly used as OLE Automation servers—namely, the Microsoft Office Suite of applications—do not offer any OLE Automation features. So, if you had an application like the one pictured in Figure 7.5, which uses Excel as an OLE Automation server to perform an interest calculation, you would be facing an additional porting task.

FIGURE 7.5:

The Desktop version
of CarCalc

Price		Down	
Selling Price of car:	30000	Down Payment:	7000
Sales Tax Rate:	8.25	Rebate:	2000
Tax:	2475	Trade In:	1000
Registration Fees:	425	Interest	
Other Fees:	137.98	Num of Months:	36
Totals:		Interest Rate (APR):	3.9
Amount Borrowed:	23037.98	Total Cost of Car:	24449.4
Total Interest:	1411.42	Monthly Payment:	679.15

This application imports all of the functions Excel makes available. Then, using a pointer to the Excel object, it lets Excel perform the calculation:

```
m_nMonthlyPayment = pApp->m_pExcelWS->Pmt(dInt, m_nNumOfMonths,
m_nAmountBorrowed, d, e);
```

In this way, you have a fully functional application without having to do any of the interest payment algorithms yourself. However, when you port this CarCalc application to CE, you soon find that you're not so lucky because, in order to get this application to run on a CE device, you have to do all of the math yourself. What was previously one simple line of code is now an entire algorithm for calculating monthly payments:

```
if (m_nNumOfMonths > 0)
{
    double pow = 1;
    double value;
    double dTempAPR = m_nAPR / 12;

    for (int i = 0; i < m_nNumOfMonths; i++){
        pow = pow * (1 + dTempAPR);
        value = (m_nAmountBorrowed * pow * dTempAPR) / (pow - 1);
    }

    m_nMonthlyPayment = value;
}
```

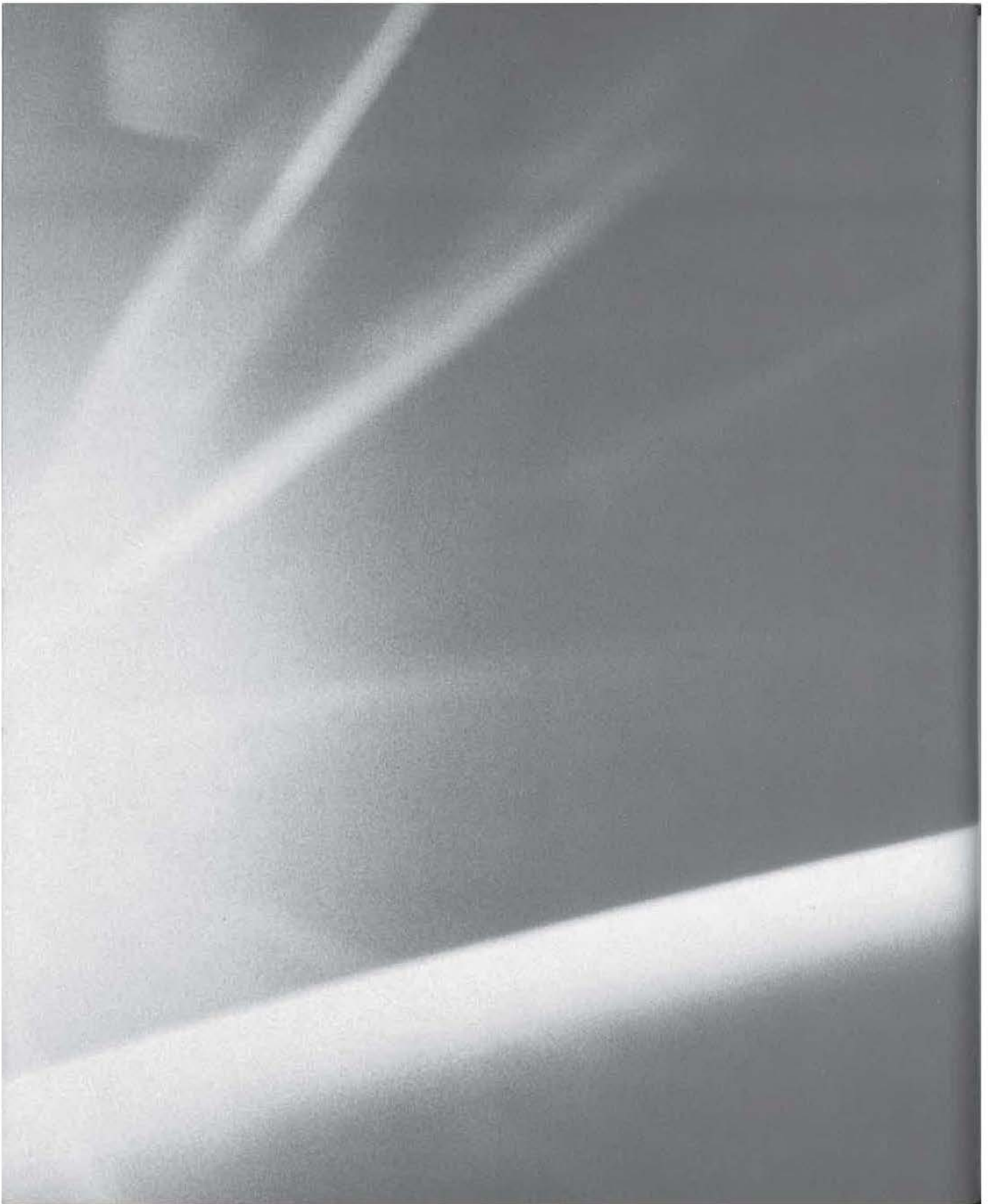
```
sTemp.Format(_T("%.2f"),m_nMonthlyPayment);  
m_nMonthlyPayment = (double)wcstod(sTemp, '\0');  
}
```

All of your code that relies on OLE Automation will almost certainly have to change in a similar fashion.

Summary

In this chapter, you examined the porting of a Desktop-based MFC application to a CE-based MFC application. You learned what sort of perils can arise and looked at some common solutions for these problems. In addition, you took a look at some of the services provided by a third-party application (Office) and what you could do to compensate for the fact that the Pocket versions of those applications did not provide the same features.

In the coming chapters, you'll be looking at the Visual Basic side of programming for CE. You'll then explore some features that are unique to the Windows CE operating system.



CHAPTER

EIGHT

8

Visual Basic Toolkit for CE

- The Application Templates
- The Debugger
- The Runtime Files
- The Control Manager
- The ActiveX Control Pack
- The Standard VBCE Controls
- The Setup Wizard

The purpose of this chapter is to show off the Windows CE toolkit for Visual Basic. We'll take a look at some of the things you can do with VBCE and what exactly the toolkit offers. We'll also take a look at the special debugger for CE-based applications. In addition, we'll pay special attention to VBCE's own Application Setup Wizard, a special tool available only for Visual Basic. Then we'll move on to changes in the actual Visual Basic language for CE. Finally, we'll present a sample Visual Basic application.

The Windows CE Toolkit for VB

When you use the Windows CE toolkit for Visual Basic to build programs, you're really using a collection of tools and products all at once. Unlike VC++, where the various SDKs and toolkit features seem to be one product, VB's toolkit is distinct and different from any tools provided with the SDK. The toolkit for VB consists of the following core components:

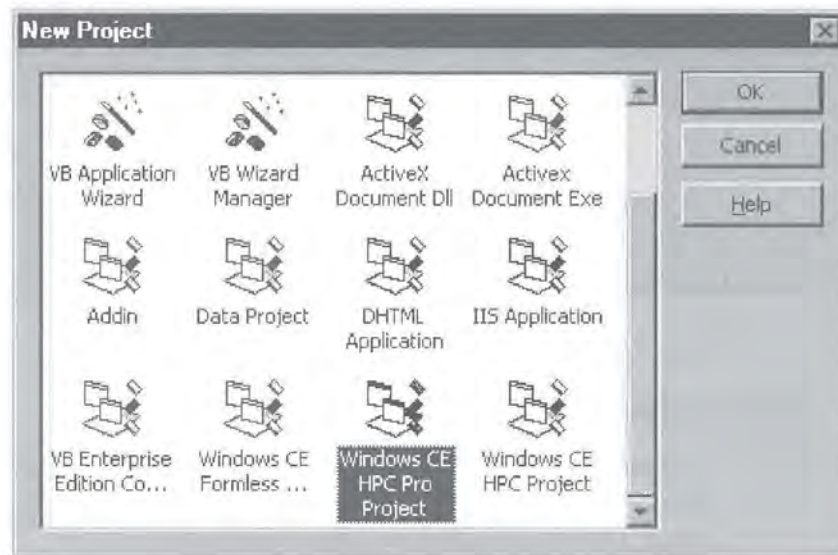
- The application templates
- The debugger
- The runtime files
- The Control Manager
- The ActiveX Control Pack
- The Standard VBCE controls
- The Setup Wizard

The Application Templates

The application templates, as shown in Figure 8.1, are the three types of Windows CE applications that you can create with VBCE. They are

- Windows CE HPC Project
- Windows CE HPC/Pro Project
- Windows CE Formless Project

FIGURE 8.1:
The Application Templates



The first of these templates has existed since the toolkit for VB 5 and is the standard project that you can create for any HPC device. The other two templates are new to the VB 6 toolkit. As you might guess, the HPC/Pro template allows you to create applications for HPC/Pro devices, such as the HP Jornada.

TIP

An HPC/Pro VBCE application will also run on HPCs that have been upgraded to Windows CE version 2.11.

The last of these templates, the Formless Project, may not make sense at first. Starting with the toolkit for VB 6, we can now use Visual Basic to create applications that are formless. The main purpose of such an application would be to develop applications to run on embedded versions of Windows CE.

Embedded Applications with VB

As you may know, Microsoft offers a version of Windows CE for developers of embedded systems. The way it works is that a developer who wants to use CE for an embedded device must purchase the Embedded Toolkit for VC++. The developer then builds a special, completely customized version of Windows CE, according to the specific needs of their application or device.

Continued on next page

However, when they're finished building their custom version of CE, they don't get the attractive Windows 98-like shell. Instead, their version of CE has either a minimal shell or is a text-mode operating system. This lack of a standard shell is the reason for the formless VB application type.

With the advent of the formless VB application, you can build your custom version of CE itself using the Embedded Toolkit and VC++, then code all of your embedded applications using Visual Basic.

The Debugger

Unlike the standard Visual Basic debugger, the debugger for CE-based applications is actually a completely separate program from the IDE. This makes it a little more difficult to use in a number of ways.

1. *Setting breakpoints in your code becomes a challenge in itself.* The problem is that, because the debugger is a separate executable, you can't set breakpoints from within the VB IDE itself. Instead, you must manually set your breakpoints while your application is running in the debugger. When your program is passed to the debugger, there are three distinct events that take place:
 - a. First, if appropriate, the debugger launches the Desktop emulator.
 - b. Next, the debugger launches your program.
 - c. Finally, your program, as shown in Figure 8.2, is displayed in the debugger.

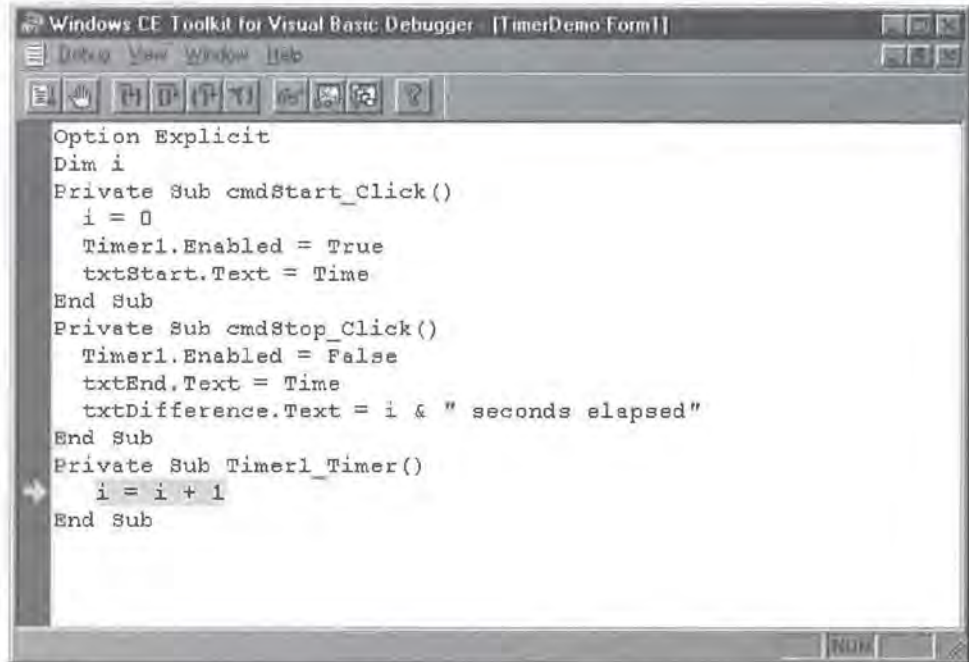
What this means to you is that it's extremely difficult to set any breakpoints on anything that executes as part of the initialization code for your project's startup object. You literally have to race to get to the code in time to put a breakpoint on any of it.

NOTE

In the newest version of the toolkit (for VB 6) there is a Step Into option in the IDE that goes a long way toward fixing the problem. However, it's still not very efficient if your application does a lot of processing at startup and you only want to break toward the end of the operation.

FIGURE 8.2:

An application as viewed in the debugger



```
Windows CE Toolkit for Visual Basic Debugger [TimerDemo Form1]
Debug View Window Help
Option Explicit
Dim i
Private Sub cmdStart_Click()
    i = 0
    Timer1.Enabled = True
    txtStart.Text = Time
End Sub
Private Sub cmdStop_Click()
    Timer1.Enabled = False
    txtEnd.Text = Time
    txtDifference.Text = i & " seconds elapsed"
End Sub
Private Sub Timer1_Timer()
    i = i + 1
End Sub
```

2. *Your breakpoints are only temporary.* As soon as you exit the debugger, they're lost. So, if you want to debug the same code more than once, you must set all the breakpoints each time.
3. *When your program is finished running, the debugger asks you if you want to close the debugger as well.* This can be quite annoying.

NOTE

It is expected that the debugger will no longer ask you if you want to close the debugger in the final release version of the toolkit; however, as of this writing, only the beta version of the toolkit was available.

In conclusion, the VBCE debugger still needs some work before it will be as user-friendly or as easy to use as VB's standard debugger.

The Runtime Files

VBCE's runtime files are not the same as the VBRUNx00.d11 files that exist in the Desktop VB world. Instead, the VBCE runtime engine is a CE port of Microsoft's

Visual Basic Script engine! This means that when you're writing a VBCE program, you're really working with an enhanced version of the VBScript language.

Although the toolkit for VB 5 only supported two processors, VBCE 6 supports nine different processors and devices. They are

- Strong ARM
- Intel 486
- MIPS 39xx
- MIPS 41xx
- MIPS 43xx
- Power PC 821
- SA 1100
- SH3
- SH4

The actual runtime files, should you ever need to manually copy them, are

- pvbload.exe
- pvbform2.d11
- pvbhost2.d11
- vbscript.d11
- vbsen.d11

The Control Manager

The Control Manager is an application designed to help developers know which ActiveX controls are registered on which devices. When it starts, Control Manager queries the Desktop operating system for a list of all registered Windows CE controls. It then queries the Emulator and the currently connected Windows CE device (if any) for the same information.

When finished, Control Manager displays the list of all the CE ActiveX controls available and whether or not they exist on the device and the Emulator. You can

then register (or delete) controls as needed. Of course, when you distribute your application it will be the responsibility of your setup program to register controls on the user's device. That's why Control Manager is so useful: it saves you from having to create that setup program every time you add or remove a control from your project.

The ActiveX Control Pack

The original version of the toolkit didn't come with anywhere near the variety of ActiveX controls that VB developers have come to expect. To remedy this situation, Microsoft released the ActiveX Control Pack, which contains six additional ActiveX controls for Windows CE. Although these controls are now shipped as part of the toolkit, they're worth mentioning separately in the event that you're using the VBCE toolkit for VB 5.

The ActiveX Control Pack includes the following controls:

- Grid
- TabStrip
- TreeView
- ListView
- ImageList
- Common Dialog

Grid

The Grid control of the ActiveX Control Pack is a simple, low-powered version of the MSFlexGrid control that comes with VB 6. While the grid is fine for displaying and organizing small amounts of data, it is read-only and doesn't support features such as FixedRows. For most applications, you're probably better off purchasing a third-party grid or modifying your project to use a ListView control. Figure 8.3 shows the Grid control at runtime.

FIGURE 8.3:

The Grid control at runtime



Solving the Read-Only Problem

For many of us, a read-only grid is not worth very much at all. Typically, users like to be able to edit the data they see in a grid, and CE users are certainly no exception. Unfortunately, the standard Grid control of the ActiveX Control Pack is read-only, just like its Desktop cousin, the MSFlexGrid.

If you've worked with the MSFlexGrid control for any length of time, you've probably seen the MSDN sample code that demonstrates how to simulate in-place editing with the MSFlexGrid. However, the problem with that code is that it doesn't work on CE without some modification. Although there are lots of issues related to porting Desktop VB code to VBCE that we'll deal with in the next chapter, this MSDN code fails for one simple reason: CE-based controls do not respect the Z-order set at design time. (As you probably know, Z-order refers to the order in which controls sit "on top of" one another—the BringToFront and SendToBack menu options available at design time.)

The problem comes from the fact that the Grid control always seems to place itself at the top of the Z-order, obscuring all the controls previously above it. The solution to this problem is to hide the Grid control, then reshown it *after* you show the Edit control.

The original MSDN code to position and show the Edit control looked like this:

```
Edt.Move MSHFlexGrid.Left + MSHFlexGrid.CellLeft, _
        MSHFlexGrid.Top + MSHFlexGrid.CellTop, _
        MSHFlexGrid.CellWidth - 8, _
        MSHFlexGrid.CellHeight - 8
Edt.Visible = True
Edt.SetFocus
```

Continued on next page

To make the Grid control effective, that code must now be changed to hide and then show the Grid control:

```
Edt.Move MSHFlexGrid.Left + MSHFlexGrid.CellLeft, _  
        MSHFlexGrid.Top + MSHFlexGrid.CellTop, MSHFlexGrid.CellWidth -  
8, _  
        MSHFlexGrid.CellHeight - 8  
MSHFlexGrid.Visible = False 'Hide the Grid  
Edt.Visible = True          'Show the Edit control  
MSHFlexGrid.Visible = True  'Show the Grid again  
Edt.SetFocus
```

This fix correctly causes the Grid control to be shown "under" the Edit control, completing the illusion of in-place editing.

TabStrip

The TabStrip control is another rather simplistic control, but it's one that can be quite useful when you have too many controls for one form. Using the TabStrip control, you can logically group related UI elements so that when the user clicks on a tab, a different set of controls is shown. (This behavior can also be seen on any Windows 98/NT property sheet dialog.) Figure 8.4 shows the TabStrip control at runtime.

FIGURE 8.4:

The TabStrip control at runtime



WARNING

The TabStrip control does not actually manage the showing and hiding of related controls. You must write the code to do this yourself.

TreeView

The TreeView control is simply an implementation of the standard Windows CE TreeView control.

ListView

Similarly, the ListView control is an implementation of the standard Windows CE ListView control. It supports the four standard views of data supported by the Desktop ListView control: Large Icon, Small Icon, List, and Report.

ImageList

The ImageList control is an implementation of the standard Windows CE ImageList control. As you probably know, the ImageList control is used to manage bitmaps that are then accessed by other controls, such as TreeViews and ListViews.

Common Dialog

The Common Dialog control is a control that serves as a wrapper for the four common dialogs:

- File Open
- File SaveAs
- Color Selection
- Font Selection

In addition, the Common Dialog control can also be used to display a help file using the CE Help system. The methods of the Common Dialog control are outlined in Table 8.1.

TABLE 8.1: Methods of the Common Dialog Control

Method	What It Does
ShowOpen	Launches the FileOpen dialog. The selected FileName is returned in the FileName property.
ShowSave	Launches the FileSaveAs dialog. The FileName is returned in the FileName property.
ShowColor	Launches the Color Selection dialog. The value of the selected color is returned in the Color property.
ShowFont	Launches the Font Selection dialog. The selected FontName, FontSize, and other attributes are returned in the similarly named properties of the control.
ShowHelp	Launches <code>PegHelp.exe</code> , the Windows CE Help Viewer. Invokes the Windows Help Engine. The HelpFile property specifies the complete path of the application's help file.

The Standard VBCE Controls

Of course, the ActiveX Control Pack is only intended to supplement the core controls offered by VBCE. Some of these controls are implemented as ActiveX controls, while others are included as part of the VBCE language itself. Wherever possible, you'll want to stick to the native VBCE controls instead of the ActiveX controls. That's because each ActiveX control you use adds to the total storage space requirement of your program, and, on a CE device, storage space is always at a premium.

Table 8.2 below lists the controls available to VBCE programmers and describes what type of control each is as well as the purpose of each control.

TABLE 8.2: The Standard VBCE Controls

Control	Implemented As	Purpose
CheckBox	Native VBCE control	Standard CheckBox control
ComboBox	Native VBCE control	Standard ComboBox control
Comm	ActiveX control	Similar to Desktop Comm control; provides access to serial communications

TABLE 8.2 CONTINUED: The Standard VBCE Controls

Control	Implemented As	Purpose
CommandBar	ActiveX control	Windows CE CommandBar; helps to improve CE-like appearance of VBCE applications
File	ActiveX control	Provides read-and-write access functions for working with files
FileSystem	ActiveX control	Exposes file-system functions such as FileCopy
Finance	ActiveX control	Provides financial functions such as PMT
Frame	Native VBCE control	Standard Frame control
Image	ActiveX control	Similar to Desktop Image control; used for displaying bitmaps, but does not support any drawing operations
Label	Native VBCE control	Standard Label control
Line	Native VBCE control	Standard Line control
ListBox	Native VBCE control	Standard ListBox control
Menu	Native VBCE control	Standard Menu control
OptionButton	Native VBCE control	Standard OptionButton control
PictureBox	ActiveX control	Similar to Desktop PictureBox control; used for displaying bitmaps; offers several drawing functions
Scrollbar	Native VBCE control	Standard Scrollbar control
Shape	Native VBCE control	Standard Shape control
TextBox	Native VBCE control	Standard TextBox control
Timer	Native VBCE control	Standard Timer control
Winsock	ActiveX control	Provides Winsock functions for communication over the IR port of the CE device

The Setup Wizard

One of the most useful tools provided with VBCE is the Setup Wizard. As you'll see in Chapter 14, creating a correct, working Windows CE setup application is a task no one welcomes. That's where the Setup Wizard comes in.

The Setup Wizard is a tool that takes your application, any related files, and the VBCE runtime engine, and creates a setup program for you. All you have to do in order to create this setup program is to give the wizard the following information:

1. Which VB (application) file you want to install
2. Where you want your files to end up when copied to the device
3. Which ActiveX controls or additional files you want installed
4. Whether or not to copy the VBCE runtime files

That's all there is to creating a setup. As you'll see in Chapter 14, creating setup programs for Windows CE is usually considerably more complex unless you opt for a third-party tool like InstallShield for Windows CE. However, with VBCE's Setup Wizard, it's a breeze.

Changes in the Visual Basic Language

As you probably know, VBCE is really a souped-up version of the VBScript language. VBScript is a language that was designed to compete with JavaScript; in other words, it's a simple language for scripting Web pages. However, VBScript is loosely based on Visual Basic, and these languages really aren't that different from one another.

But not all features of the Visual Basic language exist on CE and, as you'd expect, the features that do exist are not the same in the version 5 toolkit as they are in the version 6 toolkit.

The version 5 toolkit is fairly restrictive in what it offers in terms of the Visual Basic language. As a general rule, it seems to be much closer to VBScript than to Visual Basic. For instance, the `Dim variable As Type` syntax is not supported. You can still declare your variables using `Dim`, but you cannot specify an explicit type. That's because VBScript is basically a typeless language; every variable is a variant.

The toolkit for Visual Basic version 6 is, as the Microsoft documentation says, "essentially identical to the earlier version." However, there are a number of familiar statements and other language features that have been added. Table 8.3 shows some of the more common Visual Basic for CE Language features and which versions of the toolkit, if either, offer those features.

TABLE 8.3: Visual Basic for CE Language Features

Language Feature	Supported by Version 5 of Toolkit?	Supported by Version 6 of Toolkit?
#If...#Else	No	No
DoEvents	No	No
GoTo	No	No
With...End With	No	No
End	No	No
Debug.Print	No	No
Declare	No	Yes
Dim Variable As Type	No	Yes
LoadResString	No	Yes

In addition to the above list, all file access and financial functions are now encapsulated in ActiveX controls. However, the following standard Visual Basic controls are unsupported:

- Directory List Box
- Drive List Box
- File List Box
- Data
- OLE

A Sample Application

In order to get familiar with the Visual Basic for CE environment and language, let's create a sample application based on one of the old tutorial examples from early versions of VB. Specifically, you'll make a simple stopwatch-like application. It will record the starting and ending times, as well as the number of seconds elapsed.

Start by creating a new HPC or HPC/Pro Project by choosing File > New Project off of the main menu. The Application Properties window will appear. When you finish filling out the Properties window, you'll be presented with a standard-looking blank form.

Let's begin the visual design of the application by adding three edit controls to the form. Stack them so they appear one on top of the other. Name these edit controls `txtStart`, `txtEnd`, and `txtDifference`. When you're done with that, add three label controls to identify the edit controls' purposes as Starting Time, Ending Time, and Difference.

Now, add two buttons to the form. Name the first button `cmdStart` and change its caption to Start. Name the second button `cmdStop` and change its caption to Stop.

Finally, add a timer control to your form. The default name of `Timer1` will be fine, but change the interval to 1000. This will cause the `Timer()` event to be triggered every 1000 milliseconds, or once a second. When you're all finished, you should have a form that looks something like that shown in Figure 8.5.

FIGURE 8.5:

The Stopwatch form at design time



Now you'll add some code to the form to make it actually do something! Start by declaring the variable `i`. Do this just under the `Option Explicit` line at the top of your code. When you're done, the first two lines of your code should read

```
Option Explicit  
Dim i As Integer
```


Next, add some code for the cmdStart button's Click() event. Specifically, you'll be initializing *i*, starting the timer control, and recording the starting time in the txtStart edit control:

```
Private Sub cmdStart_Click()  
    i = 0 'initialize i  
    Timer1.Enabled = True 'start the timer  
    txtStart.Text = Time 'record starting time  
End Sub
```

Then, add some code for the timer control's Timer() event. Specifically, increment *i* by 1 whenever the Timer() event is triggered:

```
Private Sub Timer1_Timer()  
    i = i + 1  
End Sub
```

Finally, add some code to the cmdStop button's Click() event. Here, you'll stop the timer, record the ending time in the txtEnd edit box, and display the number of seconds elapsed by writing the value of *i* to the txtDifference edit box:

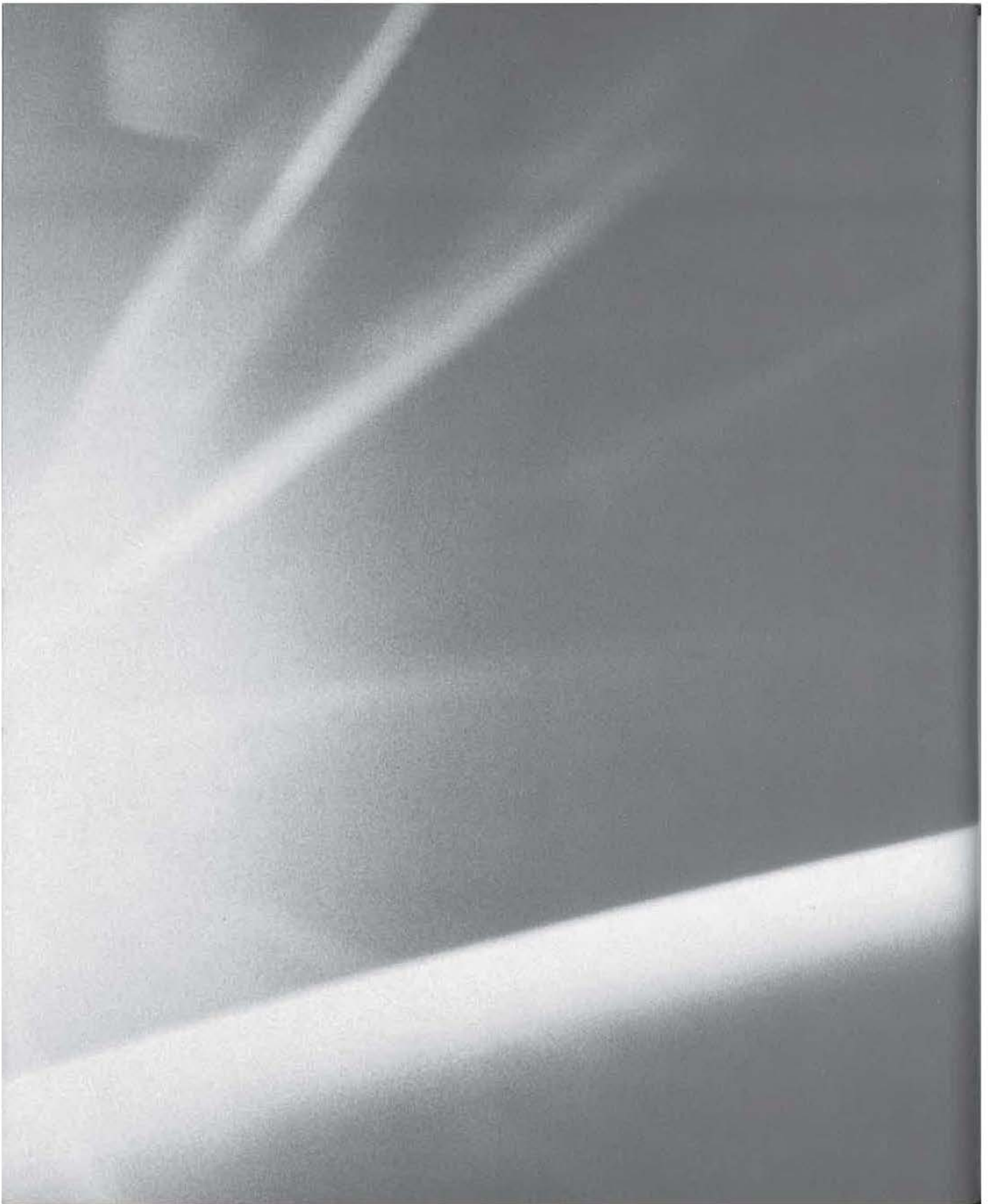
```
Private Sub cmdStop_Click()  
    Timer1.Enabled = False 'stop the timer  
    txtEnd.Text = Time 'record the ending time  
    txtDifference.Text = i & " seconds elapsed" 'display elapsed time  
End Sub
```

You're done! You now have a CE stopwatch application, created in only a few minutes, with a minimum of coding and controls.

Summary

In this chapter, you examined Visual Basic for Windows CE. First, you saw what the toolkit had to offer. You looked at some of the special tools and utilities we'll need to build and distribute VBCE applications. You also examined some of the language differences between the VB you're used to and VBCE. Finally, to familiarize yourself with VBCE, you created a simple application.

In the next chapter, you'll look at what it takes to get a Desktop Visual Basic application ported to Windows CE and how to make that process go as smoothly as possible.



CHAPTER

NINE

9

A Real VB Application Converted to CE

- A Sample Application: International ATM
- Mechanical Issues of Porting
- Optional Features
- Optimizing for CE

In this chapter, we'll be converting a real Visual Basic application to Windows CE. When we're done, you'll have an understanding of what's involved in converting even the simplest of Visual Basic desktop applications to the CE platform. In addition, you'll see why you have to drop certain features and that you may be inspired to add new ones.

At this point, you already have an understanding of which Visual Basic features are and are not supported under Windows CE. It probably comes as no surprise that Microsoft's recommendation for porting existing Visual Basic applications to CE is: *don't!* Microsoft believes that you're better off starting from scratch. However, that's simply not realistic, since many of you already have significant investments in your existing Visual Basic code, and you're probably not willing to rewrite everything. The steps to porting a VB application to CE are covered in the following order:

1. We'll take a look at the project as it exists now.
2. We'll work our way through the conversion, starting with the simpler tasks. The goal is to get a no-frills version of the application running on a CE device.
3. During the conversion, we'll consider porting some optional features.
4. We'll look at optimizing the user interface and the code for the Windows CE platform.

The Application: International ATM

The application we'll be porting is one you're probably already familiar with—it's the ATM sample that ships with VB5. In case you've never run into it before, though, let's take a quick look at it.

NOTE

This chapter uses the ATM sample application that comes with Visual Basic 5. You can find this application in the `VB\Samples\Guide\ATM` directory on the VB 5 CD-ROM.

The ATM program is a simple demonstration of a multilingual cash machine. When you first start it, the application presents the flags of five countries with a button next to each one, as shown in Figure 9.1.

FIGURE 9.1:

The opening form of the ATM application



As you move your mouse over each of the buttons, your cursor changes to indicate that you can click on the buttons. Clicking on one of the buttons selects your preferred language for the rest of your transaction. Upon selecting your preferred language, you are presented with another form, asking for two pieces of information (see Figure 9.2):

- Your PIN number
- The amount you want to withdraw in U.S. dollars

FIGURE 9.2:

The controls on the form display text in whatever language the user chose.



Another form then pops up and asks you to confirm the amount of your withdrawal in U.S. dollars, as shown in Figure 9.3. (Naturally, the program performs the currency conversion for you.)

FIGURE 9.3:

The ATM application performs the currency conversion.



Finally, once the transaction is finished, the program plays a “thank you” message as a WAV file in your chosen language.

Porting the ATM Application: Not As Simple As It Looks

At first glance, it doesn't look like there's anything here that would be difficult to port. After all, there are only about 8 buttons, 4 edit boxes, and just over 150 lines of code in the whole project—it really looks like this is going to be a piece of cake.

As soon as you look a little more closely, however, you can see that there are quite a few things in the ATM program that absolutely will not port to CE under any circumstances. For instance, the “thank you” WAV file in the example requires the following Declare statement to play:

```
Declare Function sndPlaySound Lib "WINMM.DLL" Alias "sndPlaySoundA"  
    (ByVal lpszSoundName As Any, ByVal uFlags As Long)
```

VBCE, however, has no Declare statement and no support for calling external DLLs, so the code for the existing WAV files simply will not port to CE. If you still want to play WAV files, you must find another way.

NOTE

This has been remedied with the VB 6 Toolkit for Windows CE, in which the core VBCE language was modified to include the Declare statement. At the time of this writing, both the version 5 and 6 toolkits are still in popular use. Therefore, in the porting process of this chapter, we have followed the more restrictive standards of the VB 5 toolkit for illustration purposes. Additionally, the majority of the issues discussed here are still relevant to the VB 6 Toolkit for Windows CE.

How, Exactly, Do You Port from VB (Desktop) to VBCE?

There are two ways to get the forms from one project to the other when porting from an existing VB (desktop) application to a VBCE application:

- Start a new CE project, and then attempt to add the existing desktop forms to the new CE project (Project > Add Form on the main menu).
- Start two copies of Visual Basic, open the existing project in one copy of VB, and open the new project in the other copy of VB. Then create a form in the new project. Go back to the old project and proceed to cut and paste controls (one at a time or all at once) and code from one to the other until you are finished.

Officially, Microsoft does not recommend the first method. However, both options produce essentially the same result: as you add the existing form to the new project, or as you paste one or more controls, VB will notify you about each control that is not supported under CE and will then remove it from the form. The only distinct advantage of the second method is that your forms will be automatically sized to the HPC screens by default. With the first method, you have to make sure your forms fit the smaller screens. Aside from this difference, however, both methods appear to work equally well, and which one you choose is a question of personal preference.

Planning the Porting Process

Before you start to dissect the application for porting, perhaps the best thing to do is to make a list of everything that will not port to CE without modifications of some kind. This way, you can get a sense of the work involved and can begin researching some clever workarounds or fixes.

When you take a look at everything that needs to be fixed or changed for the ATM project to compile and run under CE, the list looks like this:

- Several `Global Const` declarations. Although it doesn't appear to be documented anywhere, VBCE does not support `Global` declarations.
- All the project's variable declarations that specify a data type must be changed. For example:

```
Dim ConversionValue As Currency
```

must be changed to

```
Dim ConversionValue
```

- An array of currency values initialized with the following line of code:

```
ConversionTable = Array(1@, 4.8635@, 1.3978@, 1614@, 119.07@,  
89.075@)
```

This will not compile under CE because it uses explicit typing; specifically, the values in the array are being cast to the currency type by the @ sign placed at the end of each value. Other explicit casting operations used in the ATM project also will not port, such as `Val()` or `CCur()`.

- The Flag pictures on `frmOpen` must be changed from standard Image Controls to controls supported under CE, such as the CE Image Control. (The same change must be made to the two other forms in the application, as they also use an Image Control.)
- The WAV-playing functions, as mentioned previously, will not port to CE.
- The changing of the cursor as you pass over the buttons on `frmOpen` will not port to CE, as CE has no mouse cursors.
- The manner in which the secondary forms (i.e., not the main form) are shown modally will not port to CE, as VBCE does not support modal forms.

NOTE

A modal form is one that locks out other parts of the program until the user dismisses or closes it, typically by clicking an OK or Cancel button.

- The very core of the ATM application itself, namely, the ability to load language-specific strings and images from a resource file, will not port to Windows CE because VBCE doesn't support the necessary `LoadResString` and `LoadResPicture` functions.

When you look at this list, porting the ATM application to VBCE can appear to be somewhat daunting. Yes, there are some items that are rather easy to fix and may involve nothing more than a bit of editing and a few `REM` statements. On the other hand, the ATM application relies upon being able to show a form modally and being able to load strings from a resource file. If it can't change the language of the buttons and labels dynamically, it doesn't have a purpose anymore.

Now that we have a better sense of parts of the ATM project that won't immediately port, let's come up with an action plan so that we can manage and rank the tasks in order of importance and complexity.

Mechanical Issues of Porting

When you look at the list of things that won't port, it's clear that some of them will be easier to fix than others. Let's start with the easier parts first.

Change Global to Public Probably the easiest change to make is that of replacing the word `Global` with the word `Public`. This can be done with a simple `Find..Replace` operation that searches all files in the current project. This takes very little effort and fixes the problem completely. After all, the word `Global` only appears about three or four times in the entire ATM application!

Explicit type declarations The next easiest changes to make are the `Explicit Type` declarations. To fix these, search for `As` on all files in the project. Each time you find an occurrence, stop and edit the code to remove the `As` and the type name after the `As`. Again, there are only a few such occurrences in the ATM project, so fixing them is pretty simple.

Explicit Types While you're working on data-type-specific issues, you should also fix the currency conversion array. This occurs on line 26 of the `modATM.bas` module. Before the fix, it reads:

```
ConversionTable = Array(1@, 4.8635@, 1.3978@, 1614@, 119.07@,  
89.075@)
```

To fix it, change it to read:

```
ConversionTable = Array(1, 4.8635, 1.3978, 1614, 119.07, 89.075)
```

Typecasts Let's move on to removing the `Val()` and `CCur()` typecasts that are used on lines 20 and 21 of the code for `frmAmountWithdrawn`. They now read:

```
txtUSDollarsAmt = Val(frmInput.txtUSDollarsAmt.Text)  
txtConvertedAmt = ConversionValue * CCur(txtUSDollarsAmt.Text)
```

To fix it, change it to read:

```
txtUSDollarsAmt = frmInput.txtUSDollarsAmt.Text  
txtConvertedAmt = ConversionValue * txtUSDollarsAmt.Text
```

Bitmaps, icons, and the Image Control Then there's the question of the Image Controls and the little flag pictures. As we discovered earlier, the standard Image Control doesn't exist on CE. While it's true that there is a substitute CE Image Control that you can use for this purpose, that may not be the best action to take, as we'll examine shortly. For the time being, simply leave the flags off the form.

Now that we've finished the simpler tasks, let's move on to some of the more complex conversions and fixes.

Converting WAV Files

First, let's deal with the WAV-playing functionality. Because VBCE offers no way to call an external DLL, and since it does not provide any multimedia functions of its own, you will have to `Rem` out and/or delete most of the WAV-related code for the time being. You should delete:

- The `Declare` statement(s)
- The actual DLL calls
- Any constants associated with the sound functions

Then, stub the existing Sub procedures, so as not to break any other code that may be calling those procedures. *Stubbing* is the process of removing the code from a procedure while keeping the Sub and End Sub blocks. Stubbing these procedures reduces the chance that something dependent on these routines will break when the routine is no longer there; it also ensures that other parts of the program still have a subroutine—even an empty one—that they can reference or call without breaking.

Currently, toward the top of your modATM.bas file, you should have some code that looks like this:

```
' ...
' High level sound support API
#If Win32 Then
Declare Function sndPlaySound Lib "WINMM.DLL" Alias
    "sndPlaySoundA" _ (ByVal lpszSoundName As Any, ByVal
    uFlags As Long)As Long
#Else
    Declare Function sndPlaySound Lib "MMSYSTEM.DLL" _
        (ByVal lpszSoundName As Any, ByVal wFlags As Integer) As
        Integer
#End If
Public Const SND_ASYNC = &H1      ' Play asynchronously
Public Const SND_NODEFAULT = &H2 ' Don't use default sound
Public Const SND_MEMORY = &H4     ' lpszSoundName points to mem

Public SoundBuffer
' ...
```

Mark and delete all of the code. When you're finished with the conversion, no portion of it should remain.

TIP

Even if VBCE did have a Declare statement, you'd still have to do some work on the section of code that references the WAV-playing functions. That's because VBCE doesn't support the conditional compile directives (#If...#Else...#End If) that are used heavily in this block of code.

A little further down in that same module, some additional code must be modified.

```
Sub BeginPlaySound(ByVal ResourceId As Integer)
    Dim Ret As Variant
    #If Win32 Then
        SoundBuffer = StrConv(LoadResData(ResourceId,
            "ATM_SOUND"), vbUnicode)
    #Else
        SoundBuffer = LoadResData(ResourceId, "ATM_SOUND")
    #End If
    Ret = sndPlaySound(SoundBuffer, SND_ASYNC Or SND_NODEFAULT
        Or SND_MEMORY)
    ' Important: This function is necessary for playing sound
    ' asynchronously
    DoEvents
End Sub

Sub EndPlaySound()
    Dim Ret As Variant
    Ret = sndPlaySound(0&, 0&)
End Sub
```

To convert it, change the block of code to now read:

```
Sub BeginPlaySound(ByVal ResourceId As Integer)
End Sub

Sub EndPlaySound()
End Sub
```

What you did here was remove all of the code from the body of the `BeginPlaySound` and `EndPlaySound` procedures. Here again, by stubbing these routines, you ensure that any procedures attempting to call the `BeginPlaySound` or `EndPlaySound` procedures will be able to proceed as if the sound was still started or stopped.

Modifying Mouse/Cursor Code for CE

As we've covered in previous chapters, most CE devices do not have a mouse. And, clearly, if there's no mouse, there's also no mouse cursor. Therefore, to ensure that the ATM application will be compatible with as many CE devices as possible, all of the cursor-related code will have to go. But, as before, let's stub the procedures

related to the cursor operations—just in case. For example, the following block of code is from `modATM.bas`:

```
Sub Cursor_Initialize()
    Set curSelect = LoadResPicture(1, vbResCursor)
End Sub

Sub SetCursor(Button As CommandButton)
    Button.MousePointer = 99
    Button.MouseIcon = curSelect
End Sub
```

To stub the cursor operator procedures, modify this segment of code to read:

```
Sub Cursor_Initialize()
    'CE: no cursors
End Sub

Sub SetCursor(Button)
    'CE: no cursors
End Sub
```

Again, you're completely gutting these procedures by deleting all of the code between the `Sub` and `End Sub` lines. The reason to leave the functions here, although they're empty, is to accommodate other parts of the program that may still contain a call to one of these procedures. Of course, you should still go back and try to remove all references to the empty procedures, but you might not catch all of them.

Converting Modal Forms

Now that you've finished the smaller stuff, you come to one of the bigger difficulties of porting the ATM application to CE—the fact that VBCE doesn't support modal forms. It's simple to compile the application:

1. Use a Find operation for the string `Show 1`. Every time you find a match, delete the `1`.
2. Make sure there are no `Unload Me` commands below a `Show 1` command. In the ATM project, an example of this can be found in the `Sub cmdOK_click()` event of `frmInput`. Before conversion, the code reads:

```
frmAmountWithdrawn.Show 1
Unload Me
```

Unload Me lines should be removed or deleted because Step 1 (changing the Show 1 command to read Show) will cause an error when frmInput attempts to unload itself while its child form, frmAmountWithdrawn, is still visible. Removing the Unload Me command suppresses this error.

Snag! It Works, but the Forms Aren't Modal Anymore! Once you've made these changes, you're on your way...or, are you? As you may have realized, this fix leaves the ATM application with a serious problem. Specifically, none of the forms wait for the active (formerly modal) form to be properly dismissed! This is a disaster for any program that depends on modal forms—not just the ATM application. Although there are several popular workarounds, none truly simulate a modal form perfectly. However, we have an application to port, so let's apply some workarounds and see if we like the final result once it's all done.

The most popular workaround for this problem seems to be to show the fake modal form and then immediately hide the parent form. Then, when the fake modal form is dismissed (as a true modal form would be), it will be up to the fake modal form to show the parent form again. For example, in the parent form, you might have code that looks like this:

```
frmFakeModal.Show      ' first show the fake modal form
Hide                   ' then hide the parent form...
```

Then, in the frmFakeModal.Unload event, you might have code that looks like this:

```
frmParent.Show        ' show the parent form again
```

For our purposes, this is a fairly good workaround for the lack of true modal forms. However, for larger projects, involving a greater number of forms, this could get a bit complicated.

To fix this problem with this project, you've got to find every place that a form was previously shown or unloaded and replace that code with your workaround code. In the case of the ATM project, you must add code similar to the code listed above wherever there is a Command Button click event on each form.

As an example, let's take a look at frmOpen and one of the language selection Command Buttons. The code for the Click event of the button marked Español (Command5) currently looks like this:

```
Sub Command5_Click()
    i = 144                ' User chose Spanish.
    frmInput.Show
End Sub
```

To add the modal form workaround, modify this code to read:

```
Sub Command5_Click()  
    i = 144           ' User chose Spanish.  
    frmInput.Show  
    Hide  
End Sub
```

The next step is to repeat this operation for the remaining five buttons on `frmInput`. Once you've done that, you'll need to add this code to the `Unload` event of `frmInput`, as follows:

```
frmOpen.Show
```

You're not finished yet! You still must add the same logic to the `Command Buttons` of `frmInput` and `frmAmountWithdrawn`. For `frmInput`, the `cmdOK Click` event should be modified to look something like this:

```
Sub cmdOK_click()  
    MsgBox (LoadResString(7 + i)) ' Remmed for the moment  
    Hide ' Hide yourself before showing the next form..  
    frmAmountWithdrawn.Show  
End Sub
```

For `frmAmountWithdrawn`, add the following two lines of code to the end of the `cmdOKEnd Click` event:

```
frmOpen.Show  
Hide
```

That will solve the modal form problem!

To recap, here's what we just did:

- Added the workaround code to the `Command Buttons` of `frmOpen`
- Modified the `Unload` event of `frmInput`
- Modified the `cmdOK Click` event (on `frmInput`)
- Modified the `cmdOKEnd Click` event (on `frmAmountWithdrawn`)

Modifying Resource Loads

Now, we move to the last—and perhaps most important—porting issue of the ATM application: specifically, the lack of support for loading resources from a resource file. The main feature of the ATM application is that it can dynamically

change its language. It does this by loading all of its string resources (not to mention, flag pictures) from a resource file. In some sense, then, if we cannot port this feature or find a workaround that allows the ATM application to dynamically change its captions and other text, the entire application will be worthless.

At this point, you have a couple of options:

- You can scrap the idea of dynamically changing the language of the program and instead create six separate executables, one for each language.
- You can store the string resources in some kind of separate file to be distributed along with the executable.
- You can use some cleverly designed arrays to store the string resources.

Of these ideas, the last one is probably the most elegant and appropriate solution, because the final CE version of the ATM application will be very similar to the original one. There won't be, as in the case of the second choice, any additional files to distribute along with the executable, not to mention the task of writing the appropriate file reading code. Also, the third option doesn't waste disk space, while the first option clearly does. For these reasons, we'll try to implement the third option.

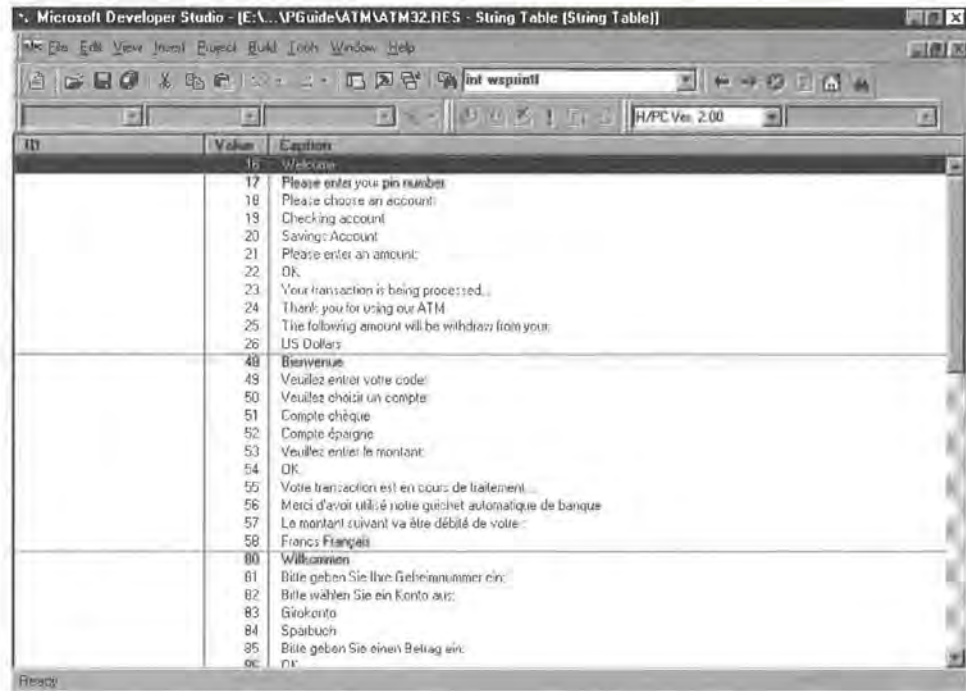
TIP

There are cases when having the different language strings in a separate file is beneficial. For instance, it's easier to change the text of a particular message, as you wouldn't have to recompile the application.

When you examine the actual resource file (using Developer Studio (see Figure 9.4) or your favorite resource editor), you find that for each language, there are actually only 11 messages.

Currently, the ATM project already has one array that it uses to perform the math for the currency conversion figures displayed by `frmAmountWithdrawn`. That array contains one value (representing how many units of the chosen currency it takes to equal one U.S. dollar) for each language, or six entries in all.

FIGURE 9.4:
The resource file as viewed
in Developer Studio



Using this existing concept, we're going to make 11 additional arrays (one for each of the messages) Each array will have six entries (one for each of the languages).

1. Your first step is to declare an array for each of the 11 messages. Each of these arrays will then contain a single phrase in each of the six languages. Declare these arrays in the `modATM.bas` module because all forms of the project will use them. This table shows the declarations for the arrays, as well as the English translation of the message each one will store.

Array Declaration	Messages
<code>Public LanguageWelcome</code>	"Welcome"
<code>Public LanguagePinNumber</code>	"Please enter your PIN number:"
<code>Public LanguageChooseAccount</code>	"Please choose an account:"
<code>Public LanguageChecking</code>	"Checking account"

Array Declaration	Messages
<code>Public LanguageSavings</code>	"Savings account"
<code>Public LanguageEnterAmount</code>	"Please enter an amount:"
<code>Public LanguageOK</code>	"OK"
<code>Public LanguageTransaction</code>	"Your transaction is being processed..."
<code>Public LanguageThankYou</code>	"Thank you for using our ATM."
<code>Public LanguageAmountWithdrawn</code>	"The following amount will be withdrawn from your:"
<code>Public LanguageCurrency</code>	"U.S. Dollars"

- Now that you've declared the arrays, write the code to initialize and populate them with the translated messages. Put this code in the Load event of `frmOpen`, as that's where the original ATM project did its initialization.

Your code to initialize the array for the Welcome messages will look like this:

```
LanguageWelcome = Array("Welcome", "Bienvenue", "Willkommen", "Benvenuti", "Bienvenido", "TRANSLATE: Welcome")
```

NOTE

Microsoft did not provide translations for the Japanese messages in the original ATM program, so they are not translated here.

Then add the array containing the different "Insert Pin Number" prompts, as follows:

```
LanguagePinNumber = Array("Please enter your pin number:", "Veuillez entrer votre code:", "Bitte geben Sie Ihre Geheimnummer ein:", "Digitare il proprio codice segreto:", "Por favor, ingrese su número de identificación secreto:", "TRANSLATE: Please enter your pin number:")
```

Next, the array containing the different "choose an account" prompts looks like this:

```
LanguageChooseAccount = Array("Please choose an account:", "Veuillez choisir un compte:", "Bitte wählen Sie ein Konto aus:", "Scegliere il tipo di conto:", "Por favor, elija una cuenta:", "TRANSLATE: Please choose an account")
```

The array containing the different checking account labels follows:

```
LanguageChecking = Array("Checking account", "Compte chèque",  
"Girokonto", "Conto corrente", "Conto corrente", "TRANSLATE: Checking  
account")
```

Then add the array containing the different savings account labels:

```
LanguageSavings = Array("Savings Account", "Compte épargne", "Spar-  
buch", "Libretto di risparmio", "Cuenta de ahorros", "TRANSLATE: Sav-  
ings Account")
```

Then add the array containing the different "enter an account" prompts:

```
LanguageEnterAmount = Array("Please enter an amount:", "Veuillez  
entrer le montant:", "Bitte geben Sie einen Betrag ein:", "Digitare la  
cifra richiesta:", "Por favor, ingrese un importe:", "TRANSLATE: Please  
enter an amount:")
```

Next add the array containing the translations of "OK":

```
LanguageOK = Array("OK", "OK", "OK", "Attendere", "Aceptar",  
"TRANSLATE: OK")
```

Then add the array containing the "transaction is being processed" status messages:

```
LanguageTransaction = Array("Your transaction is being processed...",  
"Votre transaction est en cours de traitement...", "Ihr Auftrag wird  
bearbeitet...", "La vostra operazione è in corso di esecuzione...", "Su  
transacción se está procesando...", "TRANSLATE: Your transaction is  
being processed...")
```

Next, add the array containing the different "thank you" messages:

```
LanguageThankYou = Array("Thank you for using our ATM", "Merci  
d'avoir utilisé notre guichet automatique de banque", "Vielen Dank daß  
Sie unseren Geldautomaten verwendet haben.", "Grazie e arrivederci",  
"Le agradecemos que haya usado nuestro cajero automático", "TRANSLATE:  
Thank you for using our ATM")
```

Next, add the array with the different "amount withdrawn" messages:

```
LanguageAmountWithdrawn = Array("The following amount will be withdrawn  
from your:", "Le montant suivant va être débité de votre :", "Die fol-  
gende Summe wird abgebucht von Ihrem:", "La seguente cifra verrà prele-  
vata dal vostro:", "La siguiente cantidad será retirada de su:",  
"TRANSLATE: The following amount will be withdraw from your :")
```


Then add the array holding the different currency names:

```
LanguageCurrency = Array("US Dollars", "Francs Français", "DM", "Lire  
Italiane", "Pesetas", "TRANSLATE: US Dollars")
```

With your arrays prepared in this manner, you will be able to minimize the number of changes you must make to the code. All you need to do is steal the code that currently does the lookup into the currency conversion array and modify it to work with your new string arrays. The code that performs the conversion lookup is as follows:

```
Dim ConversionValue  
ConversionValue = ConversionTable((i - 16) \ 32)
```

Based on that, you would change a line of code that looked like this:

```
MsgBox LoadResString(8 + i)
```

This references the new Language Arrays as follows:

```
MsgBox LanguageThankYou((i - 16) \ 32)
```

3. Your next task is to go through the source, looking for calls to `LoadResString`. (As with the changes you made earlier, this can probably be accomplished most efficiently by doing a Find operation on all files in the project for the text `LoadResString`.)

When you find a call to `LoadResString`, you must determine which message the ATM program is really looking for. This can usually be determined from the context of the surrounding code, or from the object being referenced. (However, if you get stuck, you can always run the original desktop ATM application.)

WARNING

After you've made all of your conversions from `LoadResString` to the Language Arrays, you may notice that the controls on the secondary forms (`frmInput` and `frmAmountWithdrawn`) do not change their text once they've been loaded for the first time. This is one of the consequences of the lack of true modal forms and using the workaround we were forced to use. What's happening is that those forms' Load events are only being called once—when the form is first shown. The way to get around this problem is to move all of the code that loads the language-specific text into the `Form_Activate` event instead.

Preparing the ATM's frmAmountWithdrawn Form

Once you've finished replacing all the LoadResString code, you should have a functional version of the ATM application. Upon running it, however, you may notice some errors coming from the Load event of frmAmountWithdrawn. To get the ATM program to work correctly, you'll have to make a few more changes to the code in that event.

1. The first surprise is related to the If statement, which, if you've finished your language conversion code, should look like this:

```
tblAmountWith = LanguageAmountWithdrawn((i - 16) \ 32) & " " & _  
IIf(frmInput.optChecking, LanguageChecking((i - 16) \ 32),  
LanguageSavings((i - 16) \ 32))
```

For reasons that are not clear, VBCE did not care for this statement. Instead, it works well to convert the IIf statement to an If..Else..End If block, like this:

```
If frmInput.optChecking.Value Then  
tblAmountWith = LanguageAmountWithdrawn((i - 16) \ 32) & " " &  
LanguageChecking((i - 16) \ 32)  
Else  
tblAmountWith = LanguageAmountWithdrawn((i - 16) \ 32) & " " &  
LanguageSavings((i - 16) \ 32)  
End If
```

2. In that same procedure, the following line may also create errors under certain conditions:

```
txtUSDollarsAmt = frmInput.txtUSDollarsAmt.Text
```

The error occurs when the user has not typed a value in the frmInput.txtUSDollarsAmt text box. VBCE is unable to convert the empty string value to a numeric value, and the operation fails. To fix this, insert this block of code just before the above line of code:

```
If frmInput.txtUSDollarsAmt.Text = "" Then  
frmInput.txtUSDollarsAmt.Text = "0"  
End If
```

That will prevent that situation from causing an error.

As it now stands, you have an application that will run on CE and perform all of the basic tasks that the desktop version of the same application performed.

The Optional Features

In case you were wondering, yes, we do have an application that is a CE-based port of an existing desktop application. And, for all practical purposes, it works.

But, it still doesn't play any sounds. It also doesn't show any flag pictures. The ATM application does not depend on these features to run properly, so they are an optional part of the porting process. In fact, when we get into optimizing the application for CE, we'll probably decide not to include them in the final product.

Now, however, let's take a look at each of these features and determine:

- If it's possible to port them

and

- Whether or not they should be ported

Re-Adding the WAV Files

First, there's the issue of the WAV files that the original ATM application played at the end of the transaction. These WAV files are recordings of someone saying something to the effect of "thank you for using the ATM" in the chosen language. As we've already learned, VBCE has no way to call the WAV-playing functions of Windows CE directly. In addition, VBCE does not have any kind of CE Multimedia control. That leaves only one option: to look for a third-party control that will play WAV files.

Fortunately, just such a control exists. And, as if this isn't enough, it's also free and can be downloaded from the CD included with this book or from the Internet by visiting <http://www.vbce.com>. The name of the control is VBCE Miscellaneous Utility Control and, as the name implies, it's a collection of miscellaneous routines and functions that weren't included in VBCE itself.

One of the procedures this control offers is `PlayWaveFile`, which, naturally, plays a WAV file.

1. The first step is to extract the WAV files from the resource file. This is simple and can be easily done with Visual Studio (or any other resource editor). You must perform the extraction for two reasons:
 - VBCE doesn't support loading anything from resource data.
 - The VBCE Miscellaneous Utility Control requires the actual name of a physical WAV file rather than a resource stored in the executable.

To extract the WAV files from the resource file using Visual Studio:

- a. Open ATM32.res in Visual Studio.
 - b. Find the ATM_SOUND resource type, and expand it. It will reveal six WAV files.
 - c. To extract them to individual files, right-click on each one, and choose Export.
 - d. When prompted with a Save File As... dialog box, change the file extension to .wav, but do not change the file name. For example, the first WAV file in the resource file is named 112. When you've extracted it, you should have a file called 112.wav.
2. Once you've extracted the WAV files, copy them to the emulator and the device in a directory of your choice. (The examples on the CD-ROM use the \Windows directory.) Then, go back to Visual Basic and find the line of code that previously played the WAV file in the Click event for cmdOKEnd handler. The line of code in question previously read:

```
BeginPlaySound i
```

3. If you haven't already commented that line out, do that now.
4. Next, if you haven't already installed the VBCE Miscellaneous Utility Control and added it to your toolbox, do that now. (Right-click on the toolbox and select Components.)

Once you've done that, add a VBCE Miscellaneous Utility Control to frmAmountInput.

5. Next, head back to the code for the cmdOKEnd Click event and add the following line where the previous WAV-playing code was located:

```
VBCEUtil1.PlayWaveFile ("\windows\" & i & ".wav")
```

Snag! WAV Files, but Still No Sound! And that's all there is to it... Or is it? You may notice one rather interesting problem when you go to test this program. It still doesn't play the sounds! Not on the emulator, nor on the device! Yet, both the emulator and the device *can* play WAV files, so what's the difference?

A comparison of the properties of WAV files that do play on the device (i.e., Alarm1.wav) with those that don't play reveals that the WAV files we're working with are in a different WAV file format.

There are two WAV file formats involved here:

1. PCM, 11.025kHz, 8-bit, Mono, which plays on the CE device
2. Microsoft ADPCM, 8.000kHz, 4-bit, Mono, which came with the ATM project but does not play on the CE device

NOTE

It's puzzling that CE will play the higher quality 8-bit format, but not the 4-bit format. However, CE insists on the PCM, 11.025kHz, 8-bit, Mono format and will not play anything but that.

So, our next task is to convert the WAV files to the correct format, which we can do with a tool as simple as the standard Windows Sound Recorder.

To convert the WAV files to the correct format:

1. Open them one at a time in Sound Recorder. Choose File > Properties from the main menu. You will see a Properties dialog like that pictured in Figure 9.5.

FIGURE 9.5:

The WAV file properties dialog of Sound Recorder



2. Click the Convert Now button, and you will see the Sound Selection dialog box, which allows you to choose a new format for the WAV file. This is shown in Figure 9.6.

FIGURE 9.6:

The Sound Selection dialog box



3. Choose PCM, 11.025 kHz, 8 Bit, Mono from the Attributes list box and click the OK button. Then close the Properties dialog.
4. From the Sound Recorder's main menu, select File > Save. Now, simply recopy the WAV files to the device, and they should play just fine.

NOTE

The WAV files do not play in the emulator, regardless of the file format. This may be an issue related to the emulator or the VBCE Miscellaneous Utility Control. (They do, however, play correctly on the device after conversion to the correct file format.)

Size Matters One issue that almost evaded us while we were converting WAV files to the correct format is the need to monitor the size of the WAV files. Because the original format of the WAV files was a lesser quality format, the files were smaller than after the conversion.

In fact, their size after the conversion is now more than double what it was before. The total size of all the WAV files before the conversion was 157K; after the conversion their total size was 404K!

This difference cannot be ignored on a platform where space is at such a premium. So, although it's possible to port the WAV file functionality, you may want to think twice about doing it because of the resources required. You might be better off just playing Alarm1.wav instead of the other WAV files.

TIP

The VBCE Miscellaneous Utility Control will not cause an error if the WAV file does not exist. This is very helpful because you can add the code to play the WAV files and, if you decide later that you'd rather not distribute them, you won't have to rebuild your program.

The Country and Flag Bitmaps

Now we come to the issue of the images of the different countries' flags and geographical shapes and whether they should be included in the final application. In the original ATM application, the flag images were icons displayed by an Image Control, and the country images were loaded from resources just like the WAV files. As we've already discovered, resources and the resource-related functions are not supported under CE. However, all is not lost.

Although it's true that the standard Image Control does not support resources, it will load a bitmap directly from a file. Therefore, it will probably still suit our purposes.

1. First, let's extract the country bitmaps from the resource file of the original ATM project, following the same procedures we did when extracting the WAV files earlier.

Extract the country bitmaps following the same steps as before:

- a. Open `ATM32.res` in Visual Studio.
- b. Find the bitmap resource type, and expand it to reveal six items. These are the individual bitmaps.
- c. Right-click on each one and choose Export.
- d. When prompted with the Save File As... dialog, accept the default file name, which should be the same as the resource ID. For example, the first bitmap file in the resource file is marked as 112. When you've extracted it, you should have a file called `112.bmp`.

When you're finished with the extracting process, you should have a `112.bmp`, a `144.bmp`, etc.

2. Copy these files to the emulator and the device in a directory of your choice. (As with the WAV files, the examples on the CD-ROM use the \Windows directory.)

If you haven't already added the CE Image Control to your toolbox, do that now.

3. Right-click on the toolbox and select Components. Add a CE Image Control to *both* frmInput and frmAmountWithdrawn, in about the same place they were in the original ATM project (Left: 120, Top: 120). For the sake of consistency with the original ATM project, name both of these CE Image Controls imgFlag.
4. In the Form_Activate procedure for both of the forms, add the following line of code:

```
imgFlag.Picture = "\windows\" & i & ".bmp"
```

Both forms are restored to the way they looked in the original ATM application, but frmOpen still does not have the flag icons next to each button. Of course, the icons themselves are readily available—they don't even have to be exported from a resource file, as they're just some of the standard icons that come with Visual Basic (they're located in the \Graphics\Icons\Flags directory). The problem occurring this time is that neither of the image-handling controls of VBCE support icons!

So, if you want to put these flags back on the main form, you'll have to convert them to bitmaps first.

1. You can use a variety of graphics packages or utilities to do this. In Corel Photo-Paint, for example, after opening the icon, select Image > Convert To > Paletted (8-bit). Then, choose the Optimized palette, and select 16 colors. (The images on the CD-ROM were also cropped to 16 × 16 instead of 32 × 32.)

NOTE

Just as with WAV files, Windows CE is limited to the types of bitmap formats it recognizes. Although the documentation claims that CE will recognize bitmaps that use up to 32 bits per pixel, 8 bits per pixel seems to be preferred.

For these conversions, follow your existing naming convention of using the numeric identifier (established in the original ATM project) for each language. This time, though, since you already have a 112.bmp, 144.bmp and so on, name

this new batch of flag bitmaps with a prefix of flg so that you can distinguish them from the country bitmaps. The list of file conversions looks something like this:

Original File Name	Becomes...
FlgUSA02.ico	Flg16.bmp
FlgFran.ico	Flg48.bmp
FlgGerm.ico	Flg80.bmp
FlgItaly.ico	Flg112.bmp
FlgSpain.ico	Flg144.bmp
FlgJapan.ico	Flg176.bmp

TIP

The converted resources (WAV and bitmap files) are located in the directory for this chapter on the CD.

2. When you finish converting the icons to bitmaps, copy the bitmaps to the emulator or the device. (Again, the examples on the CD-ROM use the \Windows directory.)
3. When you're ready, go back to Visual Basic and complete the operation by adding six CE Image Controls to frmOpen. As in the original ATM project, add one CE Image Control next to each button.

NOTE

In the original ATM project, the Image Controls were defined as one control array. However, there doesn't seem to be any particular reason for this, so it's up to you whether you'd like to make the CE Image Controls one array, or six individual controls.

4. Set each CE Image Control's `Picture` property to reflect the path of the correct flag bitmap. The first CE Image Control's `Picture` property should be set to `\windows\Flg16.bmp`, the second control's `Picture` property should be set to `\windows\Flg48.bmp`, and so on. (This can be done via the Property Inspector.)

When you've completed these steps, the project should be ready to run.

WARNING

The CE Image Controls do not appear to honor their Visible property if the Visible property is set to False. In other words, they will not hide themselves. In the context of this project, that's only a problem with the Japanese flag. In the original ATM project, the Japanese flag and button are simply hidden to prevent their being shown on a non-Kanji system. However, since the CE Image Control doesn't hide itself, the Japanese flag is always visible—even though the button is not. To resolve this, you may find it easier to simply remove the CE Image Control for the Japanese flag entirely, or show the Japanese button regardless of whether the system supports Kanji.

At this point, we come to the question of whether the bitmaps are a worthwhile feature to port to CE. The WAV files, you'll recall, took up quite a bit of space—nearly half a megabyte! The bitmaps, on the other hand, take up almost no space at all—about 7K. Even on a platform like CE, where all the resources are precious, the bitmaps are not a big concern.

However, as a file, the CE Image Control itself does occupy about 67K, so you should be concerned about it if you start adding lots of other controls or large files to the project.

NOTE

If you haven't already done so, you can freely remove the ATM32.res file from the ATM project. Simply find it in the VB Project window (under Related Documents), right-click it and select Remove.

The only other noticeable disadvantage to using the CE Image Controls is that frmOpen takes longer to load. You will have to decide for yourself whether the delay in loading is justified by the enhanced user interface. If you don't mind the delay, the bitmaps offer an enhancement with a minimum of cost.

At this point, you've successfully ported the ATM application to Windows CE. Everything that the original application did, the CE application does. Congratulations!

Optimizing for CE

We now come to the third and final issue of the design side of porting a desktop application to CE: optimizing and polishing the application for the constraints of a CE device.

We've already ensured that all the features are included; now we must determine which ones to eliminate to make the ATM project a better CE application.

Even though the program does work in its current state, it could use some optimization and cleaning for the low-resource environment of a CE device.

Here are some issues that need to be considered:

- Reducing the number of forms used in the application from three to two
- Eliminating any noncritical controls and DLLs
- Optimizing the code to improve overall performance

Eliminating a Form

First, there is the question of whether we can reduce the number of forms used in the application. As a general rule, reducing the number of forms or dialogs in any application will help improve the application in a number of ways:

- The file size of the executable will be reduced, because less data is compiled into the program.
- The speed of the application will be improved—if there's one less form to display and manage, there's that much more time for other operations in the program.

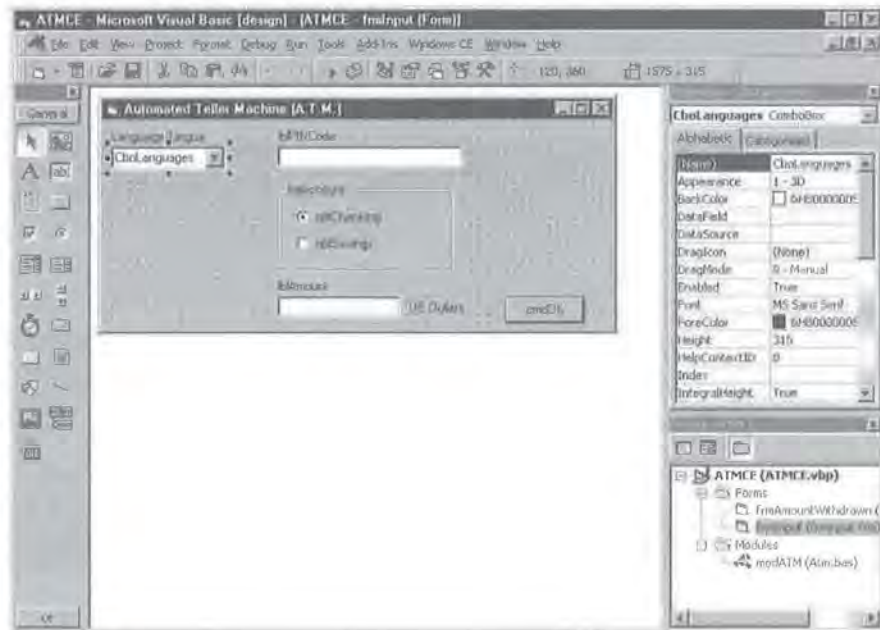
When you take a close look at the ATM project, you realize that `frmOpen` serves no real purpose—at least not so important a purpose that it requires a separate form. In fact, if we can reduce the language selection process to a single combo box, we can eliminate `frmOpen` from the project.

1. Start by adding a `ComboBox` to `frmInput` and calling it `CboLanguages`.

2. Populate its `List` property with the names of the various languages offered by the ATM project. When you're finished, `frmInput` should look something like the one pictured in Figure 9.7.

FIGURE 9.7:

`frmInput` with the Languages ComboBox added



3. That done, populate the `ItemData` property of `CboLanguages` with the existing language identifiers (i.e., 16= English, 48= French, etc.) so that the language names in the `List` property correspond to the language identifiers in the `ItemData` property.
4. Set the language ID variable (`i`) with this line of code, to be placed in the Click event of `CboLanguages`.


```
i = CboLanguages.ItemData(CboLanguages.ListIndex)
```
5. Then take all of the code that's in the `frmInput` Activate event handler, and move it to just below that line of code in the `CboLanguage` Click event handler:

TIP

With just a single line of code, we've theoretically already eliminated six event handlers—one for each of the language-selection buttons located on `frmOpen`.

This way, all of the controls that change their `Caption` or `Text` properties will refresh themselves every time the user chooses a different language.

To facilitate `frmInput` being our start-up form, we'll remove all of the initialization code from the `frmOpen` Load event, and paste it into the `frmInput` Load event.

1. In Visual Basic, find the Load event of `frmOpen`. It should currently look something like this:

```
Private Sub Form_Load()
    ' Initialize the currency conversion table.
    ConversionTable_Initialize
    LanguageWelcome = Array("Welcome", "Bienvenue", "Willkommen",
"Benvenuti", "Bienvenido", "TRANSLATE: Welcome")
    LanguagePinNumber = Array("Please enter your pin number:",
"Veuillez entrer votre code:", "Bitte geben Sie Ihre Geheimnummer
ein:", "Digitare il proprio codice segreto:", "Por favor, ingrese
su número de identificacior of ways:
no space at all-about 7Ker your pin number:")
    LanguageChooseAccount = Array("Please choose an account:",
"Veuillez choisir un compte:", "Bitte wählen Sie ein Konto aus:",
"Scogliere il tipo di conto:", "Por favor, elija una cuenta:",
"TRANSLATE: Please choose an account")
    LanguageChecking = Array("Checking account", "Compte chèque",
"Girokonto", "Conto corrente", "Conto corrente", "TRANSLATE:
Checking account")
    LanguageSavings = Array("Savings Account", "Compte épargne",
"Sparbuch", "Libretto di risparmio", "Cuenta de ahorros", "TRANS-
LATE: Savings Account")
    LanguageEnterAmount = Array("Please enter an amount:",
"Veuillez entrer le montant:", "Bitte geben Sie einen Betrag
ein:", "Digitare la cifra richiesta:", "Por favor, ingrese un
importe:", "TRANSLATE: Please enter an amount:")
    LanguageOK = Array("OK", "OK", "OK", "Attendere", "Aceptar",
"TRANSLATE: OK")
    LanguageTransaction = Array("Your transaction is being
processed...", "Votre transaction est en cours de traitement...",
"Ihr Auftrag wird bearbeitet...", "La vostra operazione è in corso
di esecuzione...", "Su transacción se está procesando...", "TRANS-
LATE: Your transaction is being processed...")
    LanguageThankYou = Array("Thank you for using our ATM", "Merci
d'avoir utilisé notre guichet automatique de banque", "Vielen Dank
```

```
daß Sie unseren Geldautomaten verwendet haben.", "Grazie e
arrivederci", "Le agradecemos que haya usado nuestro cajero
automático", "TRANSLATE: Thank you for using our ATM")
```

```
LanguageAmountWithdrawn = Array("The following amount will be
withdrawn from your:", "Le montant suivant va être débité de votre
:", "Die folgende Summe wird abgebucht von Ihrem:", "La siguiente
cifra verrà prelevata dal vostro:", "La siguiente cantidad será
retirada de su:", "TRANSLATE: The following amount will be with-
draw from your :")
```

```
LanguageCurrency = Array("US Dollars", "Francs Français",
"DM", "Lire Italiane", "Pesetas", "TRANSLATE: US Dollars")
End Sub
```

2. Mark and cut the entire body of the Load event handler.
3. Find the Load event of frmInput. (If it does not already exist, you can create it by going to the code view window of Visual Basic, selecting frmInput in the Object ComboBox, and Load in the Procedure ComboBox.)
4. Paste the code into this event.

Now you must remove all references to frmOpen from the other forms. Fortunately, there are only two such references, and they're both the exact the same line of code:

```
frmOpen.Show
```

The first of these references to frmOpen occurs in the Unload event of frmInput, and we can simply comment it out, as it no longer applies. The next reference to frmOpen occurs in the cmdOKEnd Click() event of frmAmountWithdrawn. To reflect the fact that frmInput is now our main form, this reference must be changed to:

```
frmInput.Show
```

5. Remove frmOpen from the project by right-clicking on it in the Project window (under Forms), and selecting the Remove option.
6. To complete the removal of frmOpen from the ATM project, you need to set frmInput as the start-up form.

To do that, select Project > Properties from the main menu. Then, find the ComboBox marked Startup Object and select frmInput.

You have now successfully removed `frmOpen` from the project, and the project has only two forms. This means there is one less form to manage, and, because the new start-up form does not have the images to paint (as did `frmOpen`), the entire program loads considerably faster.

Eliminating Optional Controls and DLLs

Continuing our exercise in optimization, let's also remove the remaining CE Image Controls from `frmInput` and `frmAmountWithdrawn`. In addition, let's also remove the VBCE Miscellaneous Utility Control. This will let us see how far we can compress the resources required by the application. After all, if we don't use these two controls, there are two fewer DLLs we'll have to copy onto the user's device.

Eliminating these controls from the project saves us 140K of space on the device. Further, because we've eliminated these controls, we no longer have to worry about the 400K of WAV files that we'd have to distribute with the application. Now, let's move one step further, to optimizing the actual code of the ATM project.

Optimizing the Code Itself

As you may have already realized, every time a language-specific string is loaded from an array, a separate calculation is done:

```
(i - 16) \ 32
```

When we started out, we retained the original language identifiers (i.e., 16= English, 48= French, etc.) because it seemed easier to port that way, as so much of the code was related to the resource file and the various bitmaps and WAV files. However, since you've eliminated all ties to the resource file, there's really no need to stick with the default language IDs. Further, by changing these language IDs to something simpler, you can eliminate the need to do 12 or 13 calculations whenever the user selects a different language.

Therefore, one of the simplest and easiest optimizations we can perform is to re-do all of the language identifiers so that they make more sense. And, since you're already using the `ItemData` property of `CboLanguages` to load the correct

language-specific strings, all you have to do is edit the values of `ItemData` as follows:

Language	Old ItemData Value	New Value
English	16	0
French	48	1
German	80	2
Italian	112	3
Spanish	144	4
Japanese	176	5

Clearly, if you can load the language-specific strings based on a constant value rather than an equation, it will be faster and use less processor time. For example, code that previously read:

```
Caption = LanguageWelcome((i - 16) \ 32)
```

should now read:

```
Caption = LanguageWelcome(i)
```

This will execute faster on any platform—not just CE!

Once you've edited the `ItemData` values, there is only one step that remains. Perform a Find...Replace command on all files in the project, searching for `(i-16) \ 32` and replacing it with `i`. It's that easy!

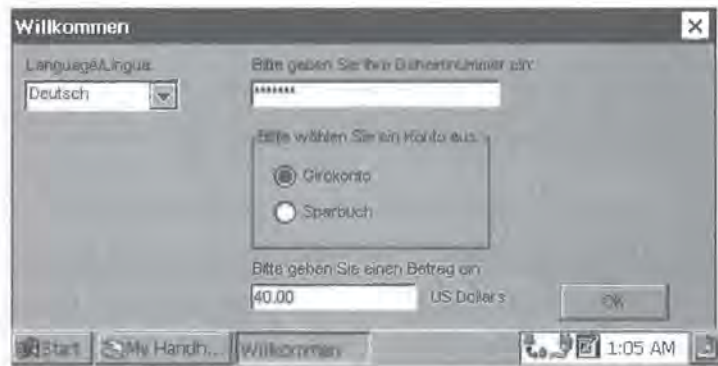
On the surface, it may appear that for all of your optimizations, you haven't really saved or optimized all that much. The size of the PVB file (VBCE executable) only shrunk from 24K to 17K, not a terrific savings. However, our program is now much faster. In fact, if you run the project now, you'll find that the main form loads faster and, every time you change the language, the various controls update themselves faster, as well.

More importantly, by eliminating the pictures, WAV files, and the controls associated with them, *you've reduced the amount of space the ATM application requires by well over 500K!*

Summary

In this chapter, you converted a desktop Visual Basic application to Windows CE. In fact, you now have a full-featured port (as well as an optimized, stripped-down version) of the application we started with.

FIGURE 9.8:
The final, optimized ATM application



You found that, although it did take some work at times, you were able to complete the porting process and then go back through your code and optimize it for the limited resources of a CE device.

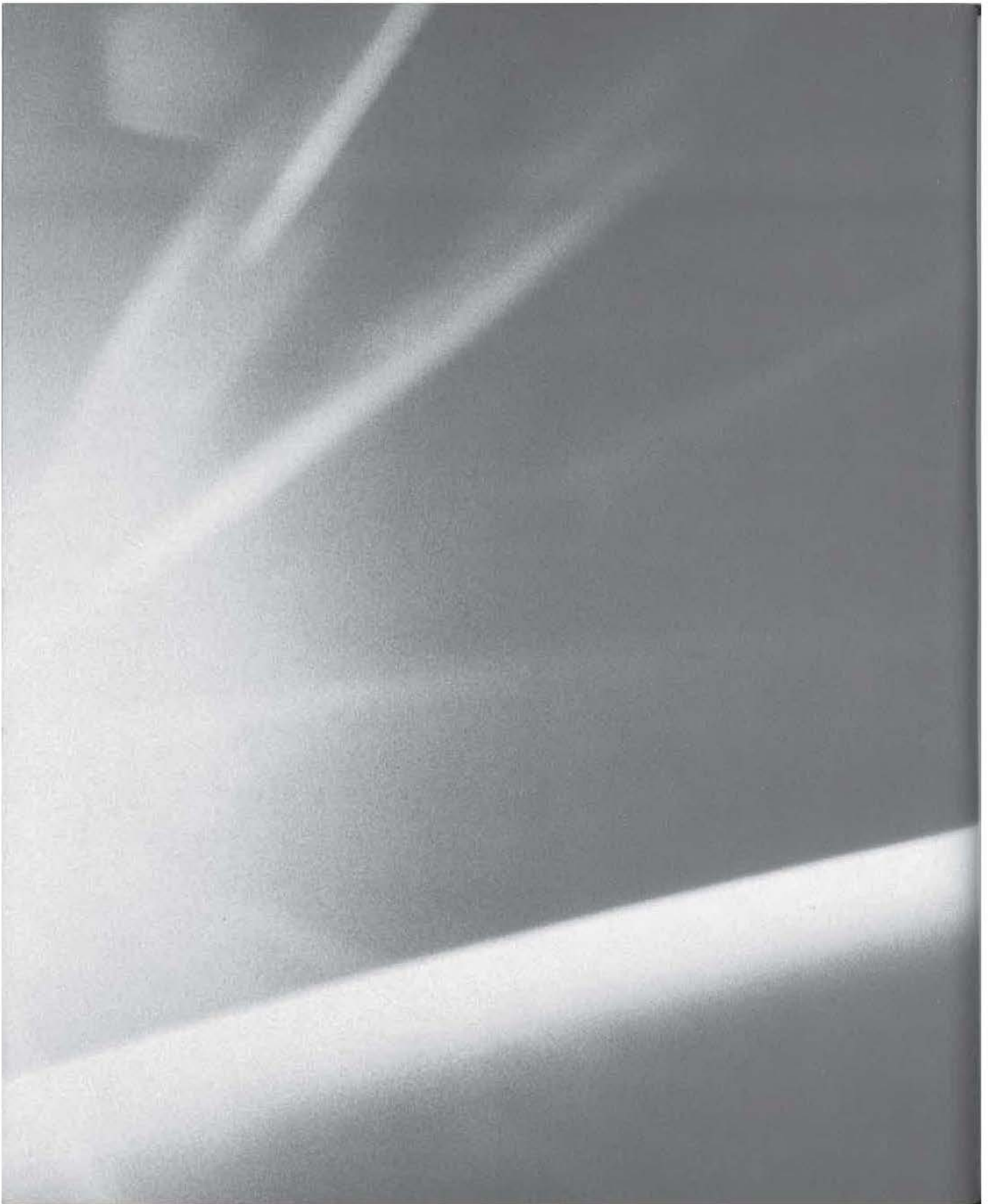
Interestingly enough, with the exception of the custom cursors that the original ATM application had, *we could port every feature to the CE platform!* Considering that Microsoft doesn't recommend the porting approach, that's quite an accomplishment.

Of course, not every application will port as well as this one did, but all of them will likely present similar issues and problems along the way.

PART III

Advanced Topics

- CHAPTER 10: RAPI: How the Outside World Talks to CE
- CHAPTER 11: How CE Talks to the Outside World
- CHAPTER 12: Third-Party Database Engines
- CHAPTER 13: Windows CE Case Studies and Cost Analysis



CHAPTER

TEN

10

RAPI: How the Outside World Talks to CE

- What Is RAPI, Anyway?
- General RAPI Management Functions
- A Sample RAPI Application
- Using Other Languages

Okay, so you've just written a killer Windows CE application that collects data, stores it into a CE database, lets the user query the data, runs reports, and maybe even has the ability to share data with other CE devices.

Now comes the tough part: You need a way to get that data into the desktop version of your software so that it can be stored in your database server along with the rest of the company's corporate data.

How do you do that?

Well, if your data is contact information, appointments or tasks, it will be automatically synched up with Outlook. But your data doesn't fall into any of those categories, and you're not using Outlook. That means there's only one way to export that data from your CE device and import it to your desktop software: RAPI.

RAPI, or Remote API, is a special set of functions that allows developers in situations like yours to access any files, databases, or system information on a CE device, regardless of what type of chip it's running or what version of CE it has. You don't even have to worry about any of the mechanics of making the connection to the device or anything like that. All you have to do is ask RAPI to make sure a device is connected, and then you're set to go.

One of the best things about RAPI is that, because it's a standard Win32 API and the code runs only on Desktop Windows, you can use any development tool that will produce Win32 executables. This is the one area of CE where you don't have to be concerned about chip types or cross-compilers!

In this chapter, we'll be looking at some of the features of RAPI and what it can do for us. We'll start with a general overview of the more important areas of RAPI, then move on to some sample applications, including one that will help solve the problem described above. Finally, we'll demonstrate that we can use any development tool to write RAPI applications by creating a sample program using Borland's Delphi tool.

What Is RAPI, Anyway?

The thing to remember about the majority of the RAPI functions is that if you know how to write CE code, you also know how to write RAPI code. All but one or two of the RAPI functions have a CE-based counterpart that behaves in exactly the same manner, takes the same parameter types, etc.

RAPI is actually just one DLL called—not surprisingly—`Rapi.dll`. In all, there are some 80 functions to RAPI, which can be broken down into the following four basic categories:

- General RAPI management functions
- System information functions
- Registry access functions
- File access functions
- Database access functions
- Miscellaneous Shell and System functions

General RAPI Management Functions

The general RAPI management functions are those functions used to load and unload the RAPI DLL and establish a connection to the currently connected device. The two functions in this group that you'll use most often are

- `CeRapiInit()`
- `CeRapiUninit()`

You must call the `CeRapiInit()` function before you do anything else. `CeRapiInit()` attempts to connect you with a CE device. In order to test whether or not you are able to connect to an actual device, use the Win32 API's `FAILED()` macro, which should look something like this:

```
//...
HRESULT hRapiResult;
hRapiResult = CeRapiInit();
if (FAILED(hRapiResult))
{
    MessageBox(hwnd, TEXT("Unable to connect to CE Device and RAPI
services"), TEXT("Error:"), MB_OK);
    SendMessage(hwnd, WM_DESTROY, 0, 0);
    return TRUE;
}
//...
```


As the code demonstrates, `CeRapiInit()` returns an `HRESULT` value that we can then use to see if there is a device for our application to talk to.

The next function that you'll be using a lot is `CeRapiUninit()`. It must be called to properly disconnect your application when you're all done working with the CE device:

```
//...
CeRapiUninit();
```

With these two management functions, your application can use any other RAPI functions.

It's worth noting that just because you use some of the RAPI functions in your application, the CE device does not have to be connected to the Desktop machine the entire time your program is running—only while you're actually using the RAPI library. It's quite possible to have your application connect to the device through RAPI, perform some action, and then immediately disconnect from the RAPI services. For instance, if you wanted to know what version of the CE operating system was running on the device, you might do something like this:

```
//...
hRapiResult = CeRapiInit();
if (FAILED(hRapiResult))
{
    MessageBox(hwnd, TEXT("Unable to connect to CE Device and RAPI
services"), TEXT("Error:"), MB_OK);
    SendMessage(hwnd, WM_DESTROY, 0, 0);
    return TRUE;
}
OsInfo.wOSVersionInfoSize = sizeof(OSInfo);
CeGetVersionEx(&OSInfo);
wsprintf(szBuf, TEXT("You are running CE Version %d.%d"),
OSInfo.dwMajorVersion, OsInfo.DwMinorVersion);
MessageBox(hwnd, szBuf, TEXT("About your device."), MB_OK);
CeRapiUninit();
//...
```

This way, the application would only be connected to RAPI and the CE device for as long or as short a time as it needs. Remember that, unless your application is going to be constantly interacting with the device, there's no need to stay connected to RAPI for any length of time.

System Information Functions

The next group of RAPI functions is the group of system information functions. These functions provide basic information about the device's hardware. The most common ones in this group are

- CeGetSystemInfo()
- CeGetVersionInfo()
- CeGlobalMemoryStatus()
- CeGetDesktopDeviceCaps()
- CeOIDGetEx()
- CeGetSystemPowerStatusEx()

Of these functions, the most useful is probably CeGetSystemInfo(). It does everything that the Windows 98/NT version of that function (called GetSystemInfo()) does. CeGetSystemInfo() can be used to determine the device's CPU, platform type, and OS version. To call it, just pass the address of a SYSTEM_INFO structure. The SYSTEM_INFO structure is defined as follows:

```
typedef struct _SYSTEM_INFO {
    DWORD dwOemId;
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    DWORD dwReserved;
} SYSTEM_INFO;
```

Likewise, CeGetVersionEx() tells us what version of Windows CE is running on the device. To call it, pass in the address of an OSVERSIONINFO structure. The OSVERSIONINFO structure is defined as follows:

```
typedef struct _OSVERSIONINFO{
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
```

```

DWORD dwBuildNumber;
DWORD dwPlatformId;
TCHAR szCSDVersion[128];
} OSVERSIONINFO;

```

CeGlobalMemoryStatus() tells you everything you need to know about the available memory on the device. When you call CeGlobalMemoryStatus(), you pass in a pointer to a MEMORYSTATUS structure, which is defined as follows:

```

typedef struct _MEMORYSTATUS {
    DWORD dwLength;           // sizeof(MEMORYSTATUS)
    DWORD dwMemoryLoad;      // percent of memory in use
    DWORD dwTotalPhys;       // bytes of physical memory
    DWORD dwAvailPhys;       // free physical memory bytes
    DWORD dwTotalPageFile;   // bytes of paging file
    DWORD dwAvailPageFile;   // free bytes of paging file
    DWORD dwTotalVirtual;    // user bytes of address space
    DWORD dwAvailVirtual;    // free user bytes }
MEMORYSTATUS;

```

You can then use the information in this MEMORYSTATUS structure in any way you like.

CeGetDesktopDeviceCaps() retrieves the value of certain system variables. To use it, pass in an identifier indicating the system variable you'd like, and the function will return the requested value. For most purposes, the values you'll be requesting are shown in Table 10.1.

TABLE 10.1: Common CeGetDesktopDeviceCaps() Identifiers

Parameter	What the device returns
HORZSIZE	Physical screen width (millimeters)
VERTSIZE	Physical screen height (millimeters)
HORZRES	Screen width (pixels)
VERTRES	Screen height (raster lines)

Registry Access Functions

The registry access functions make it possible for our Desktop applications to access the system registry of the device as if it were the registry on the Desktop machine. The functions involved here are

- CeRegOpenKeyEx()
- CeRegEnumKeyEx()
- CeRegCreateKeyEx()
- CeRegCloseKey()
- CeRegDeleteKey()
- CeRegEnumValue()
- CeRegDeleteValue()
- CeRegQueryInfoKey()
- CeRegQueryValueEx()
- CeRegSetValueEx()

One of the great things about these functions is that they operate just like the Desktop registry functions of the same name. For instance, to open a key in a Desktop program, you might have code that looks like this:

```
retCode = RegOpenKeyEx (*hKeyRoot,
                       RegPath,
                       0,
                       KEY_ENUMERATE_SUB_KEYS |
                       KEY_EXECUTE |
                       KEY_QUERY_VALUE,
                       &hKey);

if (retCode != ERROR_SUCCESS)
{
    if (retCode == ERROR_ACCESS_DENIED)
        wsprintf (Buf, TEXT("Error: unable to open key. Probably due
to security reasons.));
    else
        wsprintf (Buf, TEXT("Error: Unable to open key, RegOpenKey =
%d, Line = %d"), retCode, __LINE__);
}
```



```

    MessageBox (hDlg, Buf, TEXT(""), MB_OK);
    PostMessage (hDlg, WM_COMMAND, IDB_BACK, 0);
    return;
}

```

If you were writing for RAPI, the only difference between the above code and the RAPI-based code would be the addition of the “Ce-” prefix to `RegOpenKeyEx()`:

```

retCode = CeRegOpenKeyEx (*hKeyRoot,
                        RegPath,
                        0,
                        KEY_ENUMERATE_SUB_KEYS |
                        KEY_EXECUTE |
                        KEY_QUERY_VALUE,
                        &hKey);

if (retCode != ERROR_SUCCESS)
{
    if (retCode == ERROR_ACCESS_DENIED)
        wsprintf (Buf, TEXT("Error: unable to open key. Probably due
to security reasons.));
    else
        wsprintf (Buf, TEXT("Error: Unable to open key, RegOpenKey =
%d, Line = %d"),
                retCode, __LINE__);

    MessageBox (hDlg, Buf, TEXT(""), MB_OK);
    PostMessage (hDlg, WM_COMMAND, IDB_BACK, 0);
    return;
}

```

File Access Functions

The file access functions are those functions that allow you to create directories, read and write files, and find files matching a certain criteria. In this category, these are the most important functions:

- `CeFindFirstFile()`
- `CeFindNextFile()`
- `CeFindClose()`

- CeCreateFile()
- CeReadFile()
- CeWriteFile()
- CeCloseHandle()
- CeGetSpecialFolderPath()
- CeGetTempPath()

The first three functions—CeFindFirstFile(), CeFindNextFile(), CeFindClose()—all work together to perform file searches, much in the way that the similarly named functions work together on the Desktop. For example, if you want to find all files in the \Windows directory called "win*. *", simply call CeFindFirstFile(), passing in your file mask and an empty CE_FIND_DATA structure:

```
hFind = CeFindFirstFile( TEXT("\\windows\\win*. *"), &wfd);
```

NOTE

The CE_FIND_DATA parallels the FIND_DATA structure of the Win32 (Desktop-based) API.

WARNING

Keep in mind that all of the RAPI functions are Unicode-based, just like the Windows CE API. That means that you must convert from Unicode to ANSI when writing a RAPI program for Windows 98. Because Windows NT is already Unicode-based, no conversion is necessary.

After checking the return result of that function call, enter a do..while loop based on the result of CeFindNextFile(). Continue to loop until there are no more matching files:

```
if (INVALID_HANDLE_VALUE == hFind)
{
    CeRapiUninit();
    return 1;
}
do
{
    _tprintf(TEXT("%s\n"), wfd.cFileName);
}while(CeFindNextFile(hFind, &wfd));
```

When you've found all the matching files, free the memory allocated by the functions by calling `CeFindClose()`:

```
CeFindClose( hFind);
```

Working with the `CeCreateFile()` set of functions is also just like working with the similarly named Desktop-based functions. For instance, to open a file in a Desktop program, you might have code that looked something like this:

```
hSrc = CreateFile(
    wszSrcFile,
    GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL);
if (INVALID_HANDLE_VALUE == hSrc)
{
    MessageBox(hWnd, TEXT("Error opening file."), TEXT("ERROR"),
    MB_OK);
}
```

This is fairly standard code for opening the file for reading. If you perform the same operation under CE, your code will look like this:

```
hSrc = CeCreateFile(
    wszSrcFile,
    GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    NULL);
if (INVALID_HANDLE_VALUE == hSrc)
{
    MessageBox(hWnd, TEXT("Error opening file."), TEXT("ERROR"),
    MB_OK);
}
```

As you can see, the CE code is virtually indistinguishable from standard, Desktop-specific file access code.

CeGetSpecialFolderPath()

`CeGetSpecialFolderPath()` is an often-overlooked function that can be very useful at times. It can retrieve the true path of certain system-defined directories, such as the location of the recycle bin, the Desktop directory, etc.

`CeGetSpecialFolderPath()` is defined as follows:

```
DWORD CeGetSpecialFolderPath( int nFolder, DWORD nBufferLength,
                              LPWSTR lpBuffer);
```

The first parameter, `nFolder`, specifies an identifier for the folder we're interested in. It should be one of the following values:

- `CSIDL_BITBUCKET` The Recycle Bin
- `CSIDL_COMMON_PROGRAMS` The Start > Programs menu folder
- `CSIDL_CONTROLS` The Control Panel folder
- `CSIDL_DESKTOP` The root directory
- `CSIDL_DESKTOPDIRECTORY` The Desktop folder
- `CSIDL_FONTS` The Fonts directory
- `CSIDL_PERSONAL` The My Documents folder
- `CSIDL_PROGRAMS` Location of the user's program groups
- `CSIDL_RECENT` The Start > Documents folder
- `CSIDL_STARTMENU` The Start menu folder
- `CSIDL_STARTUP` The StartUp folder
- `CSIDL_TEMPLATES` The Templates directory

The second parameter, `nBufferLength`, specifies the length of the buffer you've allocated to hold the actual path. The third parameter, `lpBuffer`, specifies the actual buffer to hold the path.

Database Access Functions

The database access functions expose all of the functionality of the CE database engine to your Desktop-based RAPI program. Using these functions, you can view and edit databases stored on the CE device. These functions include

- `CeCreateDatabase()`
- `CeDeleteDatabase()`
- `CeDeleteRecord()`
- `CeFindFirstDatabase()`
- `CeFindNextDatabase()`
- `CeOpenDatabase()`
- `CeReadRecordProps()`
- `CeSeekDatabase()`
- `CeSetDatabaseInfo()`
- `CeWriteRecordProps()`

When it comes to database access, these functions work exactly like the CE-based versions—the only difference is the “Ce-” prefix on the function names. For a complete CE database reference, see Chapter 4.

Miscellaneous Shell and System Functions

The last and final category of RAPI functions is a set of miscellaneous shell and system functions that allow you to retrieve information about the various applications running on the CE device, work with CE shortcuts, and start CE applications remotely. These include

- `CeCreateProcess()`
- `CeSHCreateShortcut()`
- `CeSHGetShortcutTarget()`
- `CeGetWindow()`
- `CeGetWindowLong()`
- `CeGetWindowText()`

- `CeGetClassName()`

Here again, you have a set of functions that work exactly like their Desktop-based counterparts, with the exception being that they have the “Ce-” prefix. For example, if you call `GetWindowLong()` from a Desktop program, your code will look something like this:

```
lWndStyle = GetWindowLong(hWnd, GWL_STYLE);
```

Similarly, if you call the RAPI version of `GetWindowLong()`, `CeGetWindowLong()`, your code will look like this:

```
lWndStyle = CeGetWindowLong(hWnd, GWL_STYLE);
```

A Sample RAPI Application

Now that you have some idea of what RAPI is all about, let’s see what it can do for you. Let’s fulfill the purpose outlined at the beginning of this chapter and create an application that will export all of the records from a CE database into a comma-delimited text file. Of course, you could integrate this application with your existing applications and transfer the data directly into your Desktop application’s database, but this way you can use the data with any program that can read delimited text files. You can also use your CE data with any other applications. The final application will look something like the one pictured in Figure 10.1.

NOTE

Of course, before you do anything else, you must call `CeRapiInit()`.

Your first job will be to enumerate all of the databases on the CE device, adding them to a list box as you find them. To do this, first call `CeFindFirstDatabase()`, passing in 0 as the parameter.

```
if( (hEnumObject = CeFindFirstDatabase(0)) != INVALID_HANDLE_VALUE)
{
```

The 0 value tells `CeFindFirstDatabase()` that you’re going to be retrieving a list of all of the databases, so it will return a handle to the first database on the device.

FIGURE 10.1:

The final RapiDBSave application



Next, begin to iterate through the databases by calling `CeFindNextDatabase()`, this time passing in the handle of the previous database:

```
while( (ObjectID = CeFindNextDatabase(hEnumObject)) != 0)
{
```

Then, retrieve the database's name and test whether or not you were successful in this effort:

```
if ( !CeOidGetInfo(ObjectID, &CeObject) )
{
    wsprintf(szBuf, TEXT("CeOidGetInfo failed with
error (%ld)"), CeGetLastError());
    MessageBox(NULL, szBuf, TEXT("Error"), MB_OK);
    break;
}
```

If you were successful, add the database's name to your list box:

```
else
{
```

```

        // add database name to treeview
        if (!SendMessage(hwndLB, LB_ADDITEM, CeObject.infDatabase.szDbaseName,
            0, 0))
        {
            }
        }
    }
}

```

Finally, if you've exited your while loop, check to make sure that you did not encounter any errors along the way:

```

        if ( (rc=CeGetLastError()) != ERROR_NO_MORE_ITEMS )
        {
            wsprintf(szBuf, TEXT("CeFindFirstDatabase failed with error
            (%ld)", rc);
            MessageBox(NULL, szBuf, TEXT("Error"), MB_OK);
        }
    }
}

```

Your next task is to handle the saving of the database to a text file. To do this, you need to provide a button control on the main form of the application. When the user clicks the button, you'll get the name of the currently selected database in the list box and then export the data. The first step is to make sure that a database name is highlighted in your WM_COMMAND message handler:

```

        if (hwndCtl == hButton) //the export button was clicked!
        {
            if (!szCurrentDBName)
                return 0L;
        }
    }
}

```

If the user did select a database from the list, create a text file (called dbdump.txt) to hold the exported records:

```

        hFile = CreateFile(
            TEXT(".\\dbdump.txt"),
            GENERIC_WRITE,
            FILE_SHARE_READ,
            NULL,
            CREATE_ALWAYS,
            FILE_ATTRIBUTE_NORMAL,
            NULL);

        if (INVALID_HANDLE_VALUE == hFile)
        {
            MessageBox(hwnd, TEXT("Unable to open file for export."),
                TEXT("Error:"), MB_OK);
            return 1;
        }
    }
}

```


If you were able to successfully create the text file, begin to export the database's records by first ensuring that you can open the database:

```

    hOpenDB = CeOpenDatabase(&CeOID, szCurrentDBName, 0,
        CEDB_AUTOINCREMENT, NULL);
    if (hOpenDB == INVALID_HANDLE_VALUE)
        MessageBox(NULL, TEXT("CeOpenDatabase failed after creating
        DB"), TEXT("Error"), MB_OK);

```

NOTE

Note the use of the CEDB_AUTOINCREMENT flag so that as you read each record in the database, the record pointer will automatically move to the next record in the database. For more information, see Chapter 4.

Then, if you were able to open the database, read in all of the properties or fields of the database:

```

    else
    {
        while ( (CeRecObj = CeReadRecordProps(hOpenDB,
            CEDB_ALLOWREALLOC,
                &wNumRecProps, NULL, &lpRecProps, &cbRecProps)) )
        {
            if ( lpRecProps )
            {
                if ( wNumRecProps )
                {
                    pCePropVal = (PCEPROPVAL)lpRecProps;

```

NOTE

Again, we don't go into the purpose of the database-related functions here because they are explained in detail in Chapter 4.

Next, take each field and stream it out to your text file, based on the field's data type:

```

        for ( i= 0 ; i < wNumRecProps; i++)
        {
            switch(
                TypeFromPropID(pCePropVal[i].propid) )
            {

```

If it's one of the four numeric types, convert it to a string:

```

        case CEVT_I2:
        case CEVT_UI2:
        case CEVT_I4:
        case CEVT_UI4:
            wsprintf(szBuf, TEXT("%d"), pCePropVal[i].val.u1Val);
            break;

```

If it's a date/time value, convert the FILETIME value to a SYSTEMTIME value, and then copy the portions of the date that you want into a string:

```

        case CEVT_FILETIME:
            FileTimeToSystemTime(pCePropVal[i].val.filetime, &stSystemTime);
            wsprintf(szBuf, TEXT("%d/%d/%d"),
                stSystemTime.wDay, stSystemTime.wMonth, stSystemTime.wYear);
            break;

```

If it's a string, just copy it into a string, and you're done:

```

        case CEVT_LPWSTR: //and this one
            wsprintf(szBuf,
                pCePropVal[i].val.lpwstr);
            break;
        default:
            break;

```

}

NOTE

The strings here are wide or Unicode-based strings. In this example, you're writing them to the text file as Unicode strings. Just as with all of the RAPI functions, you'd have to convert these strings to ANSI string if you were targeting Windows 98.

WARNING

In this example, we're not accounting for the CEVT_BLOB type. That's because BLOBs (Binary Large Objects) can contain any type of data of any length. Further, the data in a BLOB field is very application-specific and likely requires special processing depending on the nature of the data. This, of course, makes it difficult to handle BLOBs in any kind of generic manner.

Then, simply write the string containing the contents of the field out to the file, with the appropriate error checking:

```

        if (!WriteFile(hFile, szBuf, _tcslen(szBuf)
* sizeof(TCHAR), &dwNumWritten, NULL))
        {
            MessageBox(hwnd, TEXT("Unable to write
record props to text file."), TEXT("Error"), MB_OK);
            break;
        }

```

Now, it's time to write the field separator out to the file. As with all standard comma-delimited text files, only write the comma if you're not at the end of a line. The way to do that here is to make sure you're not writing out the last field or property of the current record:

```

        if (i != wNumRecProps-1)
            WriteFile(hFile, TEXT(","),
_tcslen(TEXT(",")) * sizeof(TCHAR), &dwNumWritten, NULL);
    }
}

```

Then, when you're finished writing the entire record, tack on a carriage return and linefeed:

```

        WriteFile(hFile, TEXT("\r\n"), _tcslen(TEXT("\r\n")) *
sizeof(TCHAR), &dwNumWritten, NULL);
    }
}

```

Finally, close the handle to the database, free the record buffer, and close the text file:

```

    CeCloseHandle(hOpenDB);
    LocalFree(lpRecProps);
    CloseHandle(hFile);
    return 0L;
}

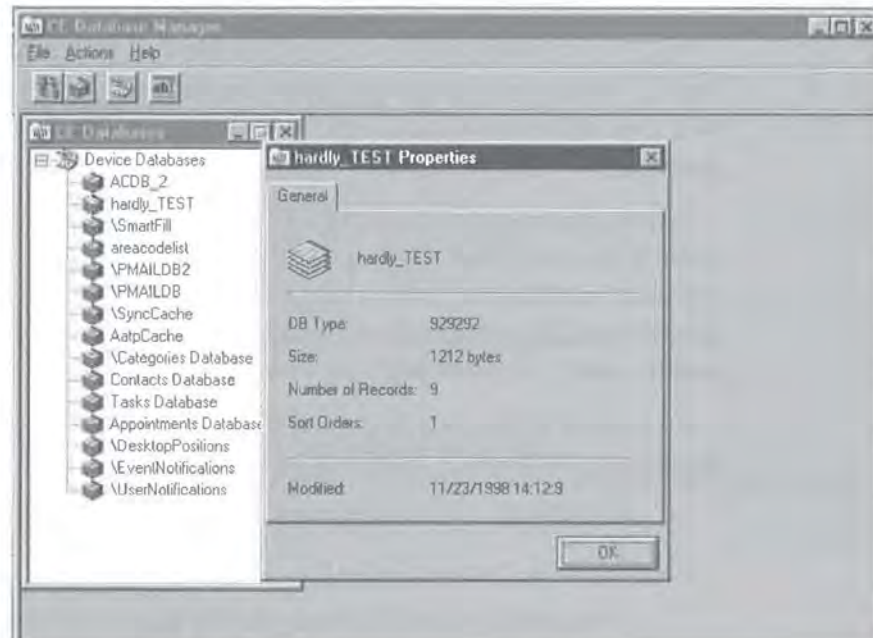
```

Using Other Languages

As you learned at the beginning of this chapter, one of the great things about RAPI is that you can use any development tool to create RAPI applications. At the

time of this writing, there is already one RAPI ActiveX control for Visual Basic, and there's also some free RAPI source code for Delphi. In this section, we'll build a simple database properties viewer in Delphi using the RAPI source code unit. The final Delphi application is shown in Figure 10.2.

FIGURE 10.2:
The final Delphi application



Static vs. Dynamic Linking

Static linking is when you let the compiler hard code the addresses of the functions you'll be calling. Dynamic linking is when you load the DLL(s) at runtime and then attempt to resolve the addresses of the functions you're looking for. Static linking is faster, but it requires that the DLL you call be present on the user's machine when your application starts. If the DLL isn't there, you'll get an error message, the program will likely shut down, and you'll have no control at all.

You have more control with dynamic linking because you can attempt to load the DLL you're after and, if you can't find it, you can handle the situation in any way you like.