

 WILEY



Essential
Windows[®] CE
Application
Programming

Robert Burdick

INCLUDES
CD-ROM



Essential Windows[®] CE Application Program

QA
76.76
.063
B856
1999



WILEY



GEORGE MASON UNIVERSITY
UNIVERSITY LIBRARIES

Essential **Windows® CE Application Programming**

Robert Burdick

Wiley Computer Publishing



John Wiley & Sons, Inc.

NEW YORK • CHICHESTER • WEINHEIM • BRISBANE • SINGAPORE • TORONTO

Publisher: Robert Ipsen

Editor: Marjorie Spencer

Assistant Editor: Margaret Hendrey

Managing Editor: Brian Snapp

Electronic Products, Associate Editor: Mike Sosa

Text Design & Composition: NK Graphics

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc., is aware of a claim, the product names appear in initial capital or ALL CAPITAL LETTERS. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This book is printed on acid-free paper. ∞

Copyright © 1999 by Robert Burdick. All rights reserved.

Published by John Wiley & Sons, Inc.

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4744. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, (212) 850-6011, fax (212) 850-6008, E-Mail: PERMREQ @ WILEY.COM.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in professional services. If professional advice or other expert assistance is required, the services of a competent professional person should be sought.

Library of Congress Cataloging-in-Publication Data:

Burdick, Robert, 1965-

Essential Windows CE application programming / Robert Burdick.

p. cm.

ISBN 0-471-32747-6 (pbk. : alk. paper)

1. Microsoft Windows (computer file) 2. Operating systems

(Computers) I. Title.

QA76.76.063B856 1999

005.4'469—dc21

98-50484

CIP

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

QA
76.76
.063
B856
1999

To my wife Katy, for urging me ever onward.

CONTENTS

Acknowledgments	xi	
Introduction	xiii	
Part I	Windows CE Application Programming Fundamentals	1
Chapter 1	Getting Started with Windows CE	3
	What Is Windows CE?	3
	Windows CE Programming Tools	7
	Before We Move On ...	13
	Now Let's Get to Work	17
Chapter 2	A Windows CE Application Template	19
	What Is a Window, Anyway?	20
	Creating Windows	27
	The Windows CE Application Entry Point	31
	The Message Loop	34
	The Template Application	37
	Concluding Remarks	40
Chapter 3	Controls and Dialog Boxes	41
	Programming Child Controls	41
	Programming Common Controls	45
	Dialog Boxes	48
	The Windows CE Common Dialogs	59
	Common Dialog Programming	61
	Concluding Remarks	68
Chapter 4	Menus and the Windows CE Command Bar	69
	I Repeat Myself When under Stress	70
	The Command Bar Control	71
	Windows CE Menu Basics	74

	Creating a Command Bar	77
	Inserting a Menu into a Command Bar	78
	Adding Controls to a Command Bar	79
	Adding Tool Tips to Command Bar Buttons	86
	Other Command Bar Functions	87
	Using Accelerators in Windows CE Applications	88
	Using the Window Menu	91
	The Complete Windows CE Menu API	93
	The Complete CMDBAR Sample Application	103
	Concluding Remarks	103
Chapter 5	Windows CE Common Controls	105
	The Month Calendar Control	107
	The Date Time Picker Control	123
	Rebar Controls	134
	Command Bands	140
	Concluding Remarks	143
Part II	Windows CE Persistent Storage	145
	Object Identifiers	146
	The CeOidGetInfo Function	146
	Viewing the Windows CE Object Store	149
Chapter 6	Working with the Windows CE File System	153
	The File System Explorer Application	154
	File Handles	159
	File Attributes	160
	Searching for Files	162
	Creating and Opening Files and Directories	165
	Reading and Writing File Data	171
	Copying and Renaming Files and Directories	178
	Deleting Files and Directories	180
	Flash Cards and Persistent Storage	180
	Concluding Remarks	183
Chapter 7	Windows CE Databases	185
	The Phone List Application	186
	Programming Windows CE Databases	191

Internal Representation of Record Properties	195
Creating the Database	197
Sorting and the SORTORDERSPEC	198
Opening and Closing the Database	201
Writing and Reading Database Records	203
Searching for Records	208
Database Enumeration	211
Database Notifications	213
The Contacts Database	213
Concluding Remarks	218
Chapter 8 Using The Windows CE Registry	221
Registry Basics	222
Creating And Opening Registry Keys	229
Reading and Writing Registry Values	231
Enumerating Registry Keys and Values	236
Deleting Registry Keys and Values	239
The Registry Sample Application	240
Concluding Remarks	241
Part III Windows CE User Interface Programming	243
What We Will Learn	245
Chapter 9 Owner Draw Controls and Custom Window Classes	247
Why Focus on Owner Draw Buttons?	247
The Example Application	249
The Anatomy of a Windows CE Control	250
How Owner Draw Buttons Are Different	251
The Kiosk Application	258
Concluding Remarks	273
Chapter 10 The Windows CE Custom Draw Service	275
Custom Draw Notification	277
Responding to Custom Draw Notifications	281
Other NMCUSTOMDRAW Info Structures	284
A Real Example	285
Concluding Remarks	288

Chapter 11	Designing Windows CE Custom Controls	289
	The Example Custom Control	290
	Packaging a Custom Control as a Dynamic Link Library	291
	Initializing the DLL in the Client Application	298
	Implementing the Custom Button Control	300
	The Complete Sample Application	310
	Concluding Remarks	311
Chapter 12	The HTML Viewer Control	313
	Overview of the HTML Viewer Control	314
	The Sample Application	317
	Preparing to Use the HTML Viewer Control	318
	Creating HTML Viewer Controls	318
	Displaying HTML Formatted Text	319
	Handling Hyperlinks	321
	Displaying Inline Images	325
	HTML Viewer Control Messages and Notifications	327
Chapter 13	Palm-size PC Input Techniques	329
	The Rich Ink Control	330
	Programming the Rich Ink Control	332
	Programming the Palm-size PC Navigation Buttons	341
	Adding Voice Input to Palm-size PC Applications	348
	A Real Example	352
	Concluding Remarks	354
Part IV	Desktop Connectivity and Memory Issues	357
Chapter 14	Windows CE Data Synchronization	359
	The Sample Code	360
	ActiveSync Technology Overview	361
	The Synchronization Process from the 50,000 Foot Level	366
	Registering ActiveSync Service Providers	369
	Desktop Service Provider Data Model	374
	Initializing a Desktop Service Provider	376
	Reconstructing Folders and Items	377
	Enumerating Objects	380

Reporting Desktop Data Store Changes	386
Transferring the Data	388
Notifying the Service Manager	392
Programming Device Service Providers	394
Conflict Resolution	399
Concluding Remarks	402
Chapter 15 Other Desktop Connectivity Topics	403
The Remote API	404
A RAPI Sample Application	405
Using Remote API Functions	405
Windows CE File Filters	409
The Sample File Filters	411
Registering File Filters	417
Concluding Remarks	419
Chapter 16 Memory and Power Management	421
The Sample Application	422
Windows CE Memory Basics	422
Allocating Memory	426
Windows CE Memory Mapped Files	433
Handling Low Memory Conditions	442
The GetSystemPowerStatusEx Function	443
Concluding Remarks	445
What's on the CD-ROM?	447
Index	451

ACKNOWLEDGMENTS

I started working on this book back in April of 1998, when I wrote the original proposal. Since that time a number of people have contributed in various ways to its successful completion.

Great thanks go to Marjorie Spencer and Margaret Hendrey, and Brian Snapp at John Wiley and Sons. Their thoughtful and professional assistance in every aspect of preparing the manuscript of this book are deeply appreciated. Pam Masara of John Wiley and Sons also deserves many heartfelt thanks for encouraging me to contact Marjorie about the idea for this book. Thanks also go to Rob Vermeulen and Peter van der Linden, both accomplished writers in their own right, for their advice and encouragement.

On the technical front, special thanks go to Martin Heller for his thorough critique of the manuscript. Thanks also to John Ruley for his review of my original proposal and his suggestions for how to improve the focus of the book. I must also thank everyone at UpperCase Software for their patience and understanding during my writing of this book. I would particularly like to thank Frank Halasz and Kim McCall for the opportunity to work for them part time while spending the majority of my time writing, and Tom Zurkan for helping me sort out various ActiveSync issues. I would also like to thank Tor Amundson of Navitel Communications for his help with various hardware issues. Former Navitel compatriot Dianna Tai also deserves thanks for her input on data synchronization.

Thanks also go to several people at Philips Mobile Computing Group. David Hargis and James Beninghaus provided me with some great opportunities to write Palm-size PC applications for the Philips NINO. Also, Michael Croot, Benjamin Beasley, and Sathish Damodaran have been instrumental in helping me meet my deadlines.

I must of course add special thanks to Mom and Dad for all of their love and moral support over the years. I also want to thank my mother-in-law,

Olga Disney, for making the best polenta. I cannot forget John and Katrina Staten for giving me the key to their house in Carmel that week in April, where the outline for this book was conceived.

Thanks also go to my two cats, Boots and Luigi, for their company on many late nights while working on this book. Jumping up on the keyboard aside, thanks for the support. (Any last minute typos are entirely their fault!)

Finally, and most importantly, I owe a debt of gratitude to my wife Katy for all of her support and encouragement. There is no way I could have done this without you. Thanks for enduring with me all of the stress, occasional depression, and of course the jubilation that went along with getting this done.

The Windows CE operating system has been available to application programmers for over two years. Independent software vendors have been writing applications for platforms such as the Handheld PC ever since Windows CE was born. At the same time, original equipment manufacturers have been designing and implementing all kinds of new devices based on the operating system. But despite the growth of the operating system and the number of software developers writing applications for it, there are still only a handful of books on the subject of Windows CE programming.

My interest in writing this book comes from over two years of Windows CE programming experience, during which I have been involved in a number of Windows CE development efforts. I am writing this book out of a desire to share with readers the insights I have gained from these experiences.

As the market for mobile and handheld computing devices continues to grow, Windows CE will continue to change. The features present in Windows CE today may not be there tomorrow. Windows CE features will be shaped by the demands of the users of the devices powered by the operating system.

But certain core technologies will always be a part of Windows CE. This book is a guide to the essential features of Windows CE programming.

How This Book Is Organized

This book is organized into four parts which focus on the following Windows CE application programming topics:

- Windows CE programming fundamentals
- Windows CE persistent storage

- User interface programming techniques
- Desktop connectivity, memory, and power management

Part I

Part I of the book covers Windows CE programming fundamentals and contains five chapters. Chapter 1 describes the architecture of the Windows CE operating system. The various Windows CE subsystems are described. In addition, Chapter 1 takes a look at how to use some of the development tools available for writing Windows CE applications. The chapter takes you through a sample session in which you learn how to build a Windows CE application for emulation as well as for a real hardware platform.

Chapter 2 covers the main ingredients of a Windows CE application. Through the example of a generic template application, the chapter introduces the concepts of the Windows CE entry point, registering window classes, writing window procedures, and creating windows. It also points out some of the fundamental differences between Windows CE windows and windows created for desktop Win32 platforms.

Next, Chapter 3 discusses the fundamentals of programming Windows CE controls and dialog boxes. The chapter introduces the basic concepts you need to use Windows CE child and common controls. It also covers how to program modal and modeless dialogs and how to write and use dialog procedures. Chapter 3 finishes with a discussion of programming the Windows CE common dialogs.

Chapter 4 covers Windows CE menus. The majority of the chapter is devoted to a discussion of Windows CE command bars. The command bar control is an essential part of using menus in Windows CE applications.

Part I concludes with a more detailed discussion of programming the Windows CE common controls. In particular, Chapter 5 covers the month calendar control, the date time picker control, rebar controls, and command bands.

Part II

Part II of this book is dedicated to Windows CE persistent storage. The three chapters in this part are your resource for learning how to program the various features of the Windows CE object store.

Chapter 6 covers using the Windows CE file system and how to program the file system API. You will learn about using files and directories, as well as how to access storage cards attached to Windows CE devices. The concepts of this chapter are made clear with the Windows CE File System Explorer sample application.

Chapter 7 discusses Windows CE database technology. You will learn how to create custom databases for your applications and how to read and write database records. You will also learn how to search for database records, and how to sort databases. In addition, Chapter 7 introduces the Windows CE contacts database.

The last chapter of Part II covers the Windows CE registry. Chapter 8 shows you how to use the registry for persistent storage of small amounts of information when a complete database or directory structure is not necessary.

Part III

Part III concentrates on various Windows CE user interface programming techniques. An entire book could easily be devoted to this subject. The five chapters in this section cover some of the more important and common user interface programming subjects.

Chapter 9 begins the discussion by introducing the concept of owner draw controls. With specific examples of programming owner draw buttons, the chapter provides an overview of how Windows CE owner draw controls can be used to customize the appearance of your applications. This chapter also covers the use of offscreen bitmaps. Chapter 10 expands on the owner draw concept with its treatment of the Windows CE custom draw service

Chapter 11 shows you how to take complete control of the appearance and behavior of your controls by describing how to create Windows CE custom controls. This chapter also provides a valuable review of how to program and use dynamic link libraries.

Chapter 12 is about using the Windows CE HTML viewer control. It shows you how to use this control to add HTML viewing capabilities to your Windows CE applications.

Finally, Chapter 13 introduces various nontraditional Windows CE input techniques. In the context of programming applications for the Palm-size PC, this chapter shows you how to program the rich ink control and how

to add voice recording capability to applications using the voice recorder control. Chapter 13 also describes how to take advantage of the Palm-size PC navigation buttons.

Part IV

The last part of this book discusses programming some of the desktop connectivity features provided by the Windows CE operating system. This section is invaluable if you are interested in writing Windows CE applications that can share data with desktop PCs.

Chapter 14 covers the ActiveSync technology for data synchronization. You will learn how to program ActiveSync service providers for both a desktop PC and a Windows CE device.

Chapter 15 shows you how to use the remote application programming interface, or RAPI, in order to allow your desktop applications to access Windows CE devices. This chapter also covers file filter programming.

The last chapter of the book introduces Windows CE memory management concepts. Chapter 16 also discusses Windows CE power considerations.

Who Should Read This Book

This book is intended primarily for readers with some Windows programming experience. It assumes that you are familiar with the basic components of a desktop Windows application. It assumes that you have written applications for Windows NT, Windows 95, or Windows 98. It also assumes that you already have some experience with Windows graphics programming topics, such as the Graphics Device Interface (GDI) functions.

However, you do not need to be a Windows expert to use this book for your Windows CE programming needs. In fact, the emphasis in this book is on programming Windows CE at the application programming interface (API) level. This book is perfectly suited, therefore, for a programmer with experience using the Microsoft Foundation Classes (MFC), but whose understanding of how the underlying API works is a bit rusty.

This is why, for example, I cover topics such as window procedures, message loops, and dialog box programming early in the book. Many

Windows programmers successfully write applications with MFC, but do not really understand how that class library works. And since Windows CE is for many reasons not particularly well suited to MFC, this book will provide many intermediate level Windows programmers with a thorough understanding of the internal workings of Windows CE.

Experienced Windows programmers will also find this book valuable because it discusses features specific to Windows CE application programming. Many of the advanced topics in this book, such as data synchronization or programming the remote API, may be unfamiliar to the most experienced Windows NT or Windows 98 programmer.

This book is intended, then, for intermediate level and advanced Windows programmers interested in writing Windows CE applications. Advanced readers may find that they want to skip the chapters that cover subjects they are familiar with from programming desktop Windows applications. For example, chapters covering Windows CE dialog box programming, custom controls, or the Windows CE file system can safely be skipped by advanced readers. However it is worth pointing out that although many Windows CE concepts are similar to their Windows NT or Windows 98 counterparts, there are often subtle differences specific to Windows CE programming. More experienced programmers will therefore find all of the chapters of this book useful.

With few exceptions, all of the examples in this book and all of the code samples on the companion CD-ROM are written in C. C++ is only used for some of the code required for the ActiveSync service providers in Chapter 14, and for the file filter examples of Chapter 15.

Tools You Will Need

To use this book, it is assumed that you have a desktop PC running Windows NT version 4.0 or later with Microsoft Developer Studio Visual C++ version 5.0 or later. It also assumes that you have installed the Windows CE Toolkit for Visual C++ version 2.0 or later.

The companion CD provides a number of sample applications illustrating the programming concepts discussed in this book. Read the appendix, "What's on the CD-ROM?" to find out more about it. If you are interested in running any of these on a Windows CE device such as a Handheld PC or Palm-size PC, it is assumed that you have installed Win-

dows CE Services on your desktop PC. This book also assumes that you are familiar with concepts such as connecting the device to the PC and copying files to the device.

NOTE

DEVELOPMENT MUST BE DONE ON WINDOWS NT

Your Windows CE applications must be developed on Windows NT. The emulation environment only works under Windows NT, and the Windows CE Toolkits and SDKs are only supported for Windows NT.

Before We Begin

This book covers a lot of material. However, no single book can possibly discuss all aspects of a subject as vast as Windows CE programming. It is my hope that this book will become your primary reference for understanding the most essential features of Windows CE programming.

As such, this book concentrates on those subjects that are most fundamental to Windows CE. As with any software product, Windows CE will see features come and go. But the fundamental building blocks on which Windows CE applications are based are sure to be around for a long time to come. It is the goal of this book to introduce you to these core Windows CE programming concepts.

Windows CE Application Programming Fundamentals

A thorough understanding of Windows CE programming requires a firm grasp of the fundamentals. We therefore begin our exploration of Windows CE application programming with a discussion of the core Windows CE topics.

We start with a brief look at the overall architecture of the Windows CE operating system. We continue with the anatomy of a typical Windows CE application. Next, programming application building blocks such as Windows CE controls and dialog boxes are covered.

Part I continues with a look at how menus are included in Windows CE applications. We will see that this is very different from how menus are added to Windows 98 or Windows NT applications. The Windows CE command bar control is presented in this discussion. Part I closes with a description of programming Windows CE common controls.

After completing the chapters in Part I, you will have a solid understanding of the basic principles required to write more complex Windows CE application programming.

Getting Started with Windows CE

In this chapter we take a brief look at the architecture of the Windows CE operating system. We also discuss some of the software development tools available to help you write Windows CE applications.

What Is Windows CE?

Windows CE is a compact, modular 32-bit operating system designed for use on devices with small memory requirements. Windows CE is very similar in design to its larger desktop cousin, Windows NT. Windows CE is a multitasking, multithreaded operating system like Windows NT. It includes most of the user interface features of Windows NT so that software developers can take advantage of most users' familiarity with Windows applications.

Storage on Windows CE devices is a combination of random access memory (RAM) and read-only memory (ROM). Devices can also include expansion flash memory storage cards for additional storage space. PCMCIA cards can be added to many devices, and Windows CE provides full support for such cards.

Since storage is all memory based, the contents of the Windows CE file system is stored in RAM. The operating system and all applications

which ship with Windows CE devices are in ROM. The ROM software components are run in place, instead of being paged into RAM, so that they run faster.

Windows CE application programmers get a huge productivity boost because Windows CE is based on the Win32 API. This means that programmers who are familiar with programming for traditional Windows platforms like Windows NT can begin programming Windows CE applications with very little additional training. Certainly there are features that are unique to Windows CE. But understanding traditional Windows programming is a big advantage when moving to the Windows CE operating system.

Architectural Considerations

Windows CE consists of seven subsystems. Each of these subsystems is further broken down into smaller components. The GWE subsystem, for example, consists of smaller components including the window manager and the dialog manager. The seven Windows CE subsystems are:

- The kernel
- The Graphics, Windowing, and Event Subsystem (GWES)
- The object store (including the file system)
- The OEM Adaptation Layer (OAL)
- The device driver layer
- The communication APIs
- Custom shells and the Internet Explorer

The Kernel

The Windows CE kernel is similar to the kernel in Windows NT. It uses the same thread and process model as Windows NT. It supports the same file formats as Windows NT. Additionally, Windows CE uses a virtual memory model similar to Windows NT. You can write Windows CE applications that share memory across multiple processes using memory mapped files.

The Windows CE kernel also implements the object manager. As is the case with Windows NT, windows, GDI objects such as brushes and

bitmaps, files, and all other such objects are manipulated by applications through object handles. The handles, as well as the underlying objects they correspond to, are managed by the object manager.

The Graphics, Windowing, and Event Subsystem

Windows CE has combined the user and GDI components into one subsystem. This subsystem, the Graphics, Windowing, and Event Subsystem, is sometimes abbreviated as GWES, or even GWE.

Windows CE behavior such as creating a window, painting a window, or loading a string resource is handled somewhere within the code of this subsystem. All of the Windows CE child controls, such as buttons, list boxes, and the like, are implemented in GWES.

GWES also contains the event manager. This is where the Windows CE messaging capabilities are implemented.

The Object Store

Random access memory in a Windows CE device is divided into two sections. The first is program memory. The other part contains the Windows CE object store. The object store contains the Windows CE file system and the registry. The object store also contains Windows CE databases such as the Contacts database and custom databases created by applications.

The OEM Adaptation Layer

The OEM adaptation layer, or OAL, consists of all of the pieces of software that an original equipment manufacturer (OEM) must implement to port Windows CE to new hardware.

If you are interested in creating a new class of Windows CE products, such as a point-of-sale terminal for ordering parts at the local auto repair shop, you need to get Windows CE to run on your custom hardware. The OAL is where you customize the interrupt service routines and hardware interfaces that allow hardware to communicate with Windows CE.

Programming the OEM adaptation layer is one of the many aspects of Windows CE embedded systems programming. As this subject

deserves an entire book of its own, it is not covered in this book, which is devoted to application programming.

The Device Driver Layer

This layer of the Windows CE operating system contains all of the drivers for peripherals that are included with a particular device. These might include flash memory card drivers, video drivers, and keyboard drivers. Detailed coverage of this subject, like the OAL, belongs in an embedded systems programming book.

The Communication APIs

Windows CE includes many of the communication APIs that you might be familiar with from Windows NT. For example, sockets, serial communication, TAPI, and the WinINet APIs are all supported under Windows CE.

One of the most important features of many Windows CE devices is their ability to share data with a desktop PC. Windows CE therefore supports ActiveSync technology. ActiveSync allows application programmers to write service providers for synchronizing application-specific data between Windows CE devices and desktop computers. Additionally, there is file filter support for transferring files between platforms.

Custom Shells and the Internet Explorer

OEMs can use the Windows CE shell component to write their own custom shells for their devices. For example, if you do not want the standard Handheld PC shell, you can write your own.

Windows CE also supports a version of the Internet Explorer.

Windows CE Modularity

One of the nicest features of Windows CE from the OEM point of view is the modularity of the operating system. Each of the various subsystem components can be added or removed as needed. If you are designing a product that does not need any of the Windows CE child controls, for example, you can remove them from the ROM operating system image that runs on your hardware. This allows OEMs to shrink the

memory footprint of the operating system by removing any components that are not needed for a particular product.

The SYSGEN tool that ships with the Windows CE Platform Builder makes this possible. When OEMs license Windows CE, they get all of the operating system component libraries. They must, however, build their own operating system image.

Part of this process involves writing a file, called CESYSGEN.BAT, that specifies which component libraries to include in the image. The SYSGEN tool then links those pre-compiled libraries into the operating system image.

Windows CE Programming Tools

Microsoft designed Windows CE with existing Windows programmers in mind. We have already discussed how Windows CE is based on the Win32 API. Programmers can also use many of the same programming tools that they are already familiar with.

This is because Microsoft Developer Studio can be used for writing and debugging Windows CE applications. Emulators for the various Windows CE platforms allow developers to write and debug applications on a desktop PC. The Windows CE Toolkit includes utilities for allowing the Microsoft Developer Studio debuggers to remotely debug applications running on Windows CE hardware. Developers can therefore begin writing and debugging Windows CE applications without learning a new set of development tools.

A Sample Session

To demonstrate the Windows CE programming tools, let's see how to build a sample Windows CE application. We will build the TEMPLATE.EXE application for the Handheld PC emulation environment. Then we will see how to build the same application for real HandheldPC hardware and download it to a device. The project files for this application can be found on the companion CD under \Samples\template.

Building for Emulation

The first step in building a Windows CE application for any target is to open the workspace file for that application. Choose the Open Work-

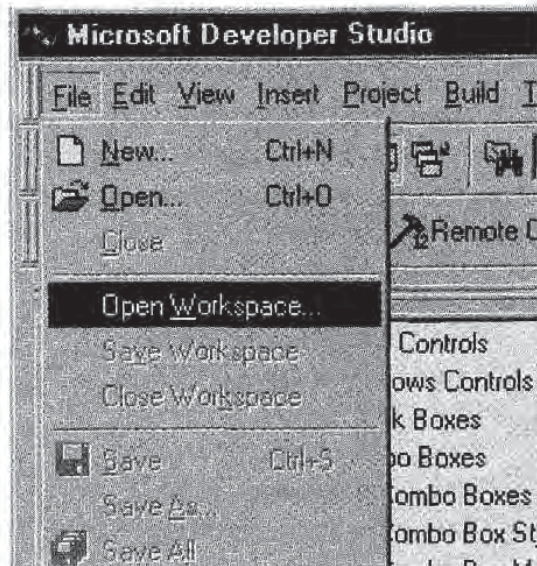


Figure 1.1 The Open Workspace menu option.

space option from the Microsoft Developer Studio File menu (Figure 1.1). From the Open Workspace dialog, find and open the file TEMPLATE.DSW (Figure 1.2).

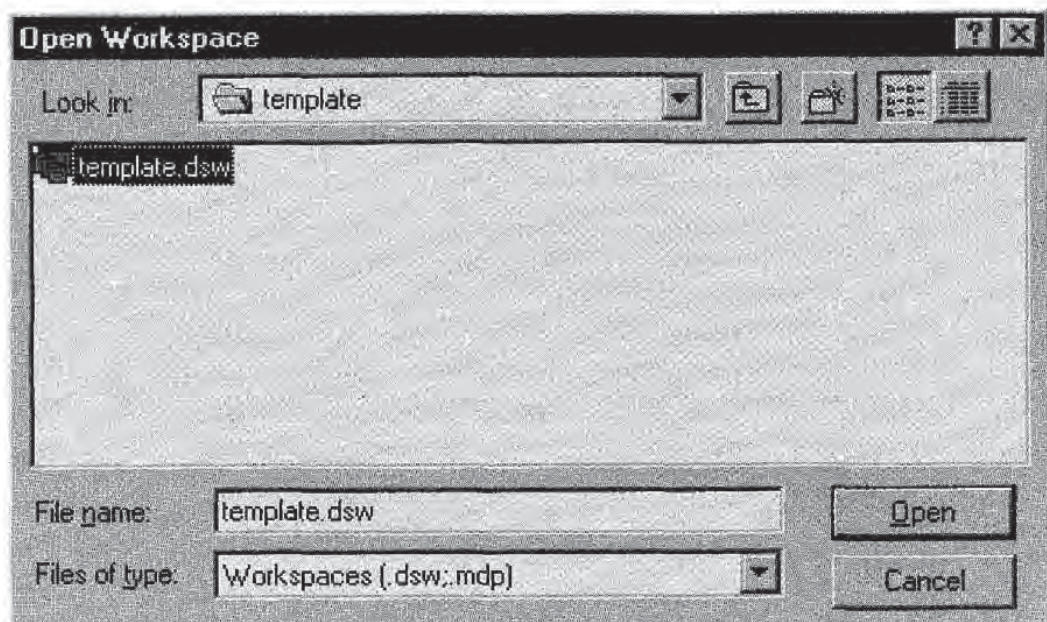


Figure 1.2 Opening the Workspace file.

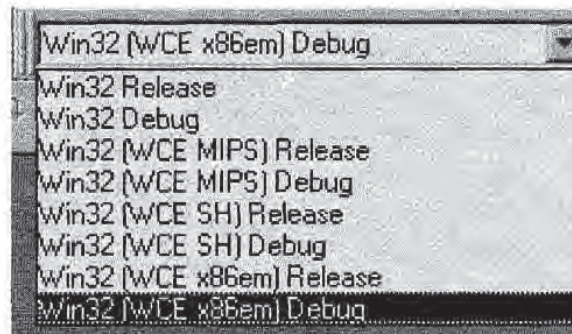


Figure 1.3 Selecting a target build configuration.

You must now specify which *configuration* to build the application for. This is done by making a selection from the combo box shown in Figure 1.3. You can specify whether to build TEMPLATE.EXE for the debug emulation environment. Or you can build the release or debug versions of the application for any of the processors for which you have installed compilers. As we want to build for debug emulation, select the “Win32 (WCE x86em) Debug” option.

If you have installed more than one Windows CE Platform SDK, a second combo box will be included in the Developer Studio toolbar which lets you select the product to build for. For this example, make sure you pick a Handheld PC version.

NOTE

WHAT IF I DON'T SEE THE CONFIGURATION COMBO BOX?

If the configuration combo box does not appear somewhere in the Developer Studio toolbar, you may need to add it manually. Select the Customize... option from the Tools menu. Then click on the Commands tab. From the Category combo box, select Build. The Buttons group will include the configuration combo box. Simply drag it to your toolbar and drop it where you want it.

Now that you have specified the configuration to build, build the application by choosing the Rebuild All option from the Build menu. The application will compile and link. During the link phase, the Handheld PC emulation environment will start up (Figure 1.4). This simulates a real Handheld PC shell.

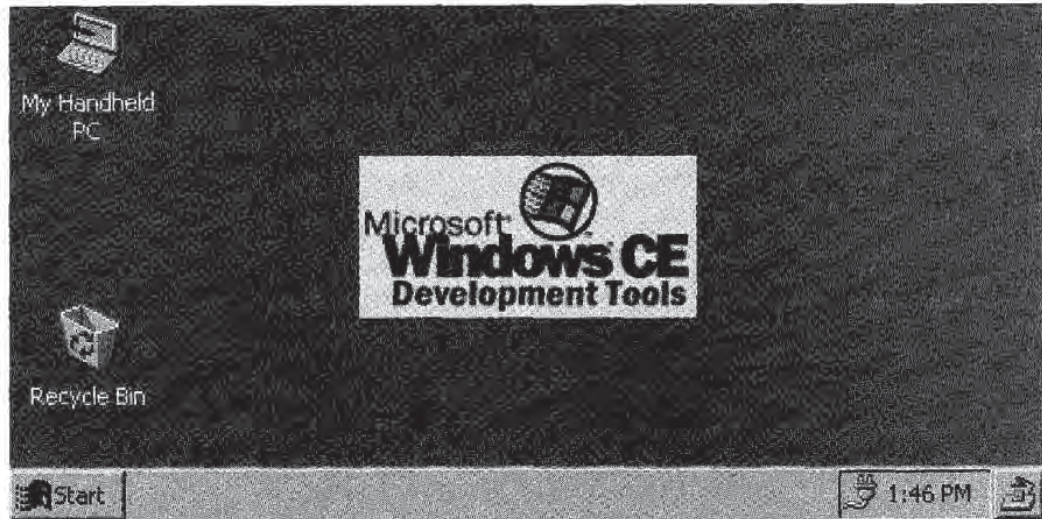


Figure 1.4 The Handheld PC emulation environment.

Now you can run and debug the application just as you would any other Windows application. When you run the `TEMPLATE.EXE` application in the emulator, you should see something like Figure 1.5.

Building for a Real Device

Building an application for a real Windows CE device configuration is similar to building for the emulator. The differences come in when it's

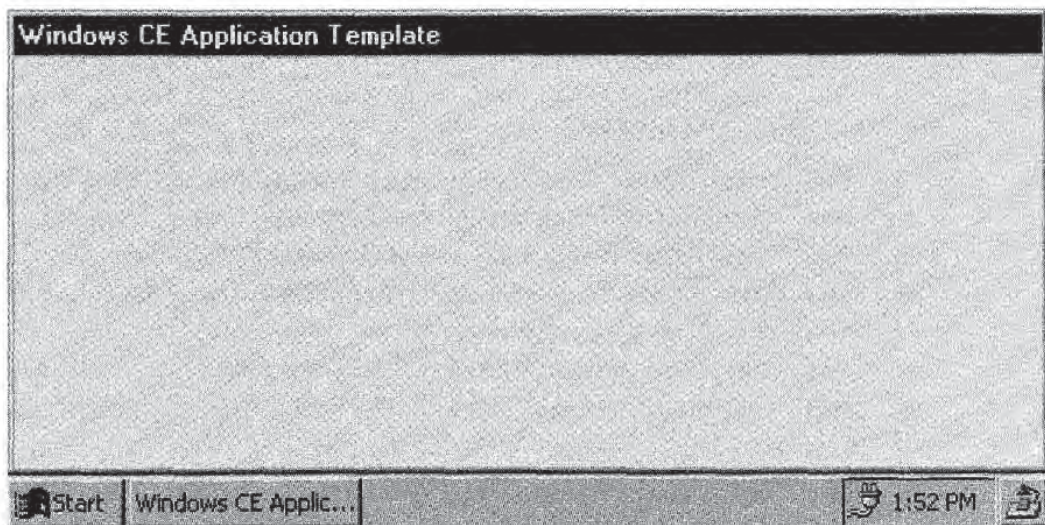


Figure 1.5 Running an application in the emulation environment.

time to transfer the executable image from the desktop PC to the hardware.

Let's say you want to build the release version of TEMPLATE.EXE for a Handheld PC running on an SH3 processor. You simply select "Win32 (WCE SH) Release" from the configuration combo box and rebuild the application (see Figure 1.3).

Now you have to get the application to the Handheld PC. Assuming that you have already connected your Handheld PC to the desktop computer, open the Mobile Devices folder on the desktop PC. You will see a window that looks something like the one in Figure 1.6. The name of the Handheld PC icon will be whatever name you gave your device when you configured it.

Double-click on the icon corresponding to your Windows CE device, and a window similar to that in Figure 1.7 will appear. This window shows the contents of your Handheld PC desktop.

To copy the TEMPLATE.EXE image to your Handheld PC, open Windows NT Explorer and drag the executable you just built to the desired location on the Handheld PC. To place it on the desktop, drop the file in the desktop window shown in Figure 1.7. To copy the file to the Windows directory, double-click the My Handheld PC icon in the Mobile Devices window displaying your Handheld PC desktop. Sev-

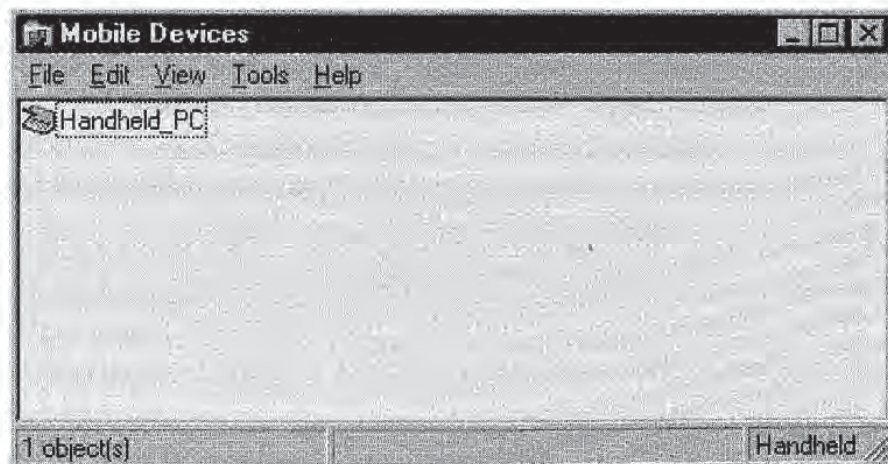


Figure 1.6 The Mobile Devices folder.

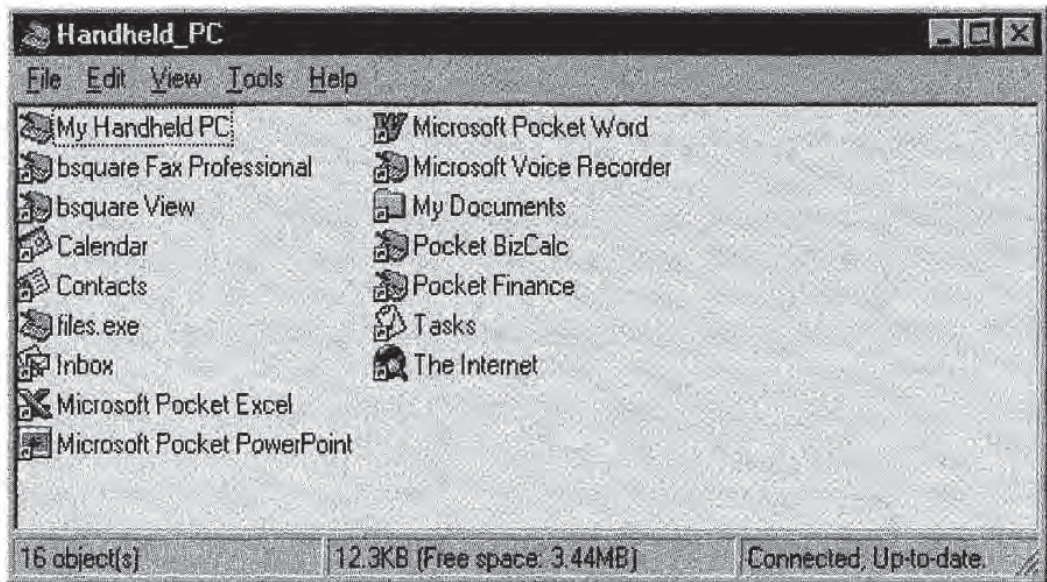


Figure 1.7 A Handheld PC desktop displayed by mobile devices.

eral folders, including the Windows folder, will appear. Drag and drop TEMPLATE.EXE to the Windows folder.

Debugging on the Windows CE Device

Now that you have successfully built a Windows CE application and downloaded it to your Handheld PC, let's take a quick look at how to remotely debug the application using the Visual C++ debugger.

Using the debugger is the same for remote applications as it is for traditional desktop applications, or Windows CE applications running under emulation. But you must first perform a few preliminary steps.

The first step is to build the application in question for the debug, instead of the release, configuration for the target processor. Next, this debug version of the application must be downloaded to the device. To do this, select the Update Remote Output File option from the Developer Studio Build menu.

After that, you can set break points and debug the application just as you would a Windows NT or Windows 98 application.

NOTE**AUTOMATIC DOWNLOADING**

It is possible to have Microsoft Developer Studio automatically download release and debug application images. If you select the Always Download option from the Build menu, compiled executables are downloaded to the appropriate target automatically. This also includes “downloading” applications to the emulation object store.

Other Tools

There are many other development tools that can be used for Windows CE programming. For example, the Windows CE Toolkits include Spy, a process viewer, and a heap walker, to name a few. These tools are not covered in this book. Readers are referred to the on-line documentation for details on using the other Windows CE development tools.

Before We Move On . . .

Companies that hire my Windows CE consulting services quickly learn one very important thing about me: Like it or not, I tell it like it is. Being in the business of helping companies succeed in their development efforts requires nothing less.

The hard part of this is that it often means telling people what they least want to hear. Pointing out serious problems with development plans or software designs can mean a lot of reengineering and taking large steps backward on a project. On the positive side, however, willingness to rethink products realistically is a key ingredient to getting a successful product to market.

So, before we turn our attention to the primary purpose of this book, writing Windows CE applications, I want to take a few pages to describe in general terms some of the biggest mistakes that are made which prevent successful Windows CE-based products from getting to market.

The State of the Art

As this book goes to press, Windows CE finds itself in a somewhat precarious situation. The number of shipping Windows CE-based

product categories is small. The currently available ones include PC companion devices such as Handheld and Palm-size PCs. The Auto PC platform hopes to make it into all of our cars. Windows CE Jupiter class products fall somewhere between the Handheld PC and a laptop computer in features and complexity. And none of these are selling in droves.

About two years ago I had the opportunity to discuss Windows CE at the Windows Hardware Engineering Conference (WinHEC) in Taipei with Frank Fite, director of the Windows CE Product Unit at Microsoft. Frank discussed the interest customers were expressing in Windows CE for products as diverse as slot machines, golf carts, and refrigerators. Today, Microsoft still discusses the queries it is receiving from companies interested in some day putting Windows CE in slot machines, golf carts, and refrigerators.

I do not make these statements to be glib. I make them to point out that to date, no one has come up with a product based on Windows CE that has generated a huge compulsion to buy in consumers.

What Happened to the Customer?

Each new version of the Windows operating system has been released to try and capture a new segment of the computer market. Windows 98 is intended to be the consumer desktop operating system of choice. Windows NT targets the business and software development communities.

Windows CE was intended to take Microsoft into the brave new world of consumer electronics. Furthermore, many of the products based on Windows CE were meant to target consumers with very little experience with (or interest in using) computers.

This presents an enormous challenge for Microsoft. The majority of the products the company sells are software packages. And despite the fact that consumers use products like Microsoft Office for personal business, the majority of users of Microsoft software are paid between eight and twelve hours a day to use computers in their daily jobs.

Moving the Windows model to products aimed at the less computer-literate segment of the population has thus proved a very daunting task. And in my opinion, based on experience with numerous Windows CE software and hardware vendors, the reason for this

difficulty is simple: Windows CE products today are far too complicated to use.

Build Benefits, Not Features

Many years ago, while working for Integrated System Corporation (ironically, a real-time operating system vendor and embedded systems integrator), the team I worked with was treated to an off-site meeting at our manager's beach house. As part of the work portion of the trip, he invited a speaker to discuss various topics with us, including how to sell a product.

This speaker made a simple, yet for many companies, elusive point which has been forever indelibly imprinted in my brain. He said that no customer will care how high-tech a product is, how state-of-the-art the software behind it is, or how sexy the user interface is, if that customer does not get some benefit from using the product. In short, we won't care about the space-age metal used in a new line of ballpoint pen if it leaks ink all over our clothes.

Most Windows CE products suffer from such feature distraction. And nowhere is the problem more pronounced than in the area of Windows CE application user interface design. A trend is evolving where companies place more power and decision-making authority in the hands of user interface design teams than in the hands of the very engineering teams that must make products a reality. Time and again, features are insisted upon which, given the current limitations of Windows CE, draw out development schedules and cause deadlines to slip. And worse, such features are sometimes added to product requirements without a single potential customer expressing a desire for the feature.

The result is late products that are user interface-intensive and far too difficult to use by the inexperienced customers to whom they are supposed to appeal.

Ironically, the most successful PC companion product to date, the PalmPilot, is not based on Windows CE. It has a very boring text-based user interface. But users love it. The reason is that the PalmPilot only tries to do a few things for the user. And the tasks it does do are very easy to perform. My wife used my PalmPilot for the first time to

look up the phone number of our doctor. In ten seconds, she found the phone number she needed. That's a product benefit.

While writing and testing the applications in this book on Handheld and Palm-size PCs, I occasionally handed the devices over to my wife so she could see what was distracting me from spending more time with her. Just figuring out how to launch these applications required the assistance of the author of an entire book about Windows CE.

So Why Use Windows CE?

The foregoing discussion might prompt readers to wonder why I am writing a Windows CE book at all. The reason is that Windows CE is a great platform for building small, easy-to-use devices that do a few things well for their users.

I hope to encourage you to constantly think about the customer for whom you are developing your Windows CE-based products. Desktop software users have (for better or worse) become used to occasionally rebooting a PC when their software crashes. But the user of a consumer electronics product heads straight back to the store for a refund if something goes wrong.

The best advice I can give to companies considering entering the Windows CE market is to keep product designs simple. The success of Windows CE as a consumer product operating system depends on the introduction of products that consumers feel compelled to buy. Have your potential customers define the minimum set of features they would need in order to buy your product. Then incorporate those features into the product with simple user interfaces.

Next, add new features only when enough customers will pay for them. Browsing the Internet on a four-inch screen sounds like a good idea to whom? If the idea originated in the marketing or engineering department, beware. If large numbers of focus group members said it would be nice, start drawing up a new requirements document. In other words, let the customer drive the design.



Now Let's Get To Work

I'll get off my soapbox now and turn my attention to the real objective of this book. In the chapters that follow, we will explore how to program the various features of the Windows CE operating system. It is my belief that after mastering the concepts presented in this book, you will have a good grasp of the essential elements of writing Windows CE applications.

A Windows CE Application Template

It is a bit difficult to know how much to say about the fundamentals of Windows CE programming. It is true that the Windows CE application programming interface (API) is a subset of the traditional Win32 API. It is also true that a majority of application programmers who are interested in developing software for Windows CE-based devices come to this new platform with some level of Windows programming experience.

A detailed introduction to Windows CE programming concepts such as window classes and window procedures would therefore be wasted on programmers with a lot of Win32 experience. On the other hand, not covering these topics at all might alienate those developers whose primary Windows experience is with class libraries such as the Microsoft Foundation Classes (MFC). All of the code samples in this book use the Windows CE API directly in order to promote a solid understanding of Windows CE programming from the most fundamental level. Some coverage of basic concepts is therefore necessary.

In order to strike a compromise, this chapter presents a basic Windows CE template application. This application can be used as the boilerplate for any other application that you may wish to write. It does nothing but display a main application window and implement the

most rudimentary window procedure. I promise that nowhere in this application will you find words even remotely reminiscent of “Hello World”!

In fact, the template application presented in this chapter is the foundation of all the other sample applications presented in this book. Each of the applications was written by taking the template source code and adding functionality specific to the topics and techniques under discussion.

This chapter serves a dual purpose. In addition to describing the basic framework of all the applications to follow in this book, it also introduces the basic ingredients of a complete Windows CE application.

Experienced Win32 API programmers and MFC programmers alike will get something out of the presentation of this template application. For example, the Win32 programmer will benefit from seeing the differences in window styles and window messages between Windows CE and desktop Windows platforms. At the same time, MFC programmers will get a refresher on the underlying mechanics of the Windows programming model.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

Register a window class

Write a window procedure

Create instances of a window class

Write a message loop

What Is a Window, Anyway?

To users of Windows CE-based devices and applications, a window is one of those things on the screen that contains buttons, scroll bars, and all of the other components that are used as the interface to the functionality of the device. To Windows programmers, however, windows have multiple levels of meaning.

One of these levels is that which the user ends up seeing: the user interface aspect of windows. The “look and feel” of an application is

defined primarily by the appearance of the application's windows. As application developers and user interface designers, we must constantly think about windows in these graphical terms.

Then there are the behavioral aspects of windows. Users interact with windows by pressing buttons, selecting menu items, and so forth, and the windows in our applications respond by performing actions.

But as programmers, we also think of windows at the more mechanical level. This level, which the users of our software probably never contemplate, is concerned with the way in which Windows CE represents windows.

The Window Class

To Windows CE, all windows are described in terms of a *window class*. The window class describes all of the attributes of the window, from the background color and the window title text to the way in which the window responds to user input. Every Windows CE window, from the most exalted main application window to the lowliest button or edit box, has a window class lurking somewhere behind it. More than one window can be based on the same window class.

A window based on a particular window class is called an *instance* of that window class. Windows CE applications reference individual windows (i.e., window class instances) via their *window handle*. Window handles are defined by the type `HWND`.

Instances of a window class are created with the Windows CE functions `CreateWindow` or `CreateWindowEx`. We will see `CreateWindow` in action a bit later in the template application.

Windows CE application programmers work with window classes in the form of the `WNDCLASS` structure, defined as follows:

```
typedef struct _WNDCLASS {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HANDLE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
```

```

    LPCTSTR lpszClassName;
} WNDCLASS;

```

The members of the WNDCLASS structure define the attributes of any instance of this class.

lpszClassName points to a null-terminated Unicode string which contains the name of the window class. Applications create instances of a particular window class using this name.

style contains all of the class styles. The styles supported under Windows CE are described in Table 2.1. The *style* parameter is one or more of the values shown in that table, bitwise-ORed together.

lpfnWndProc is a pointer to the *window procedure* for this window class. The subject of window procedures is discussed later.

cbClsExtra specifies the number of extra bytes that Windows CE is to allocate for the WNDCLASS structure. These bytes can be used to define additional class attributes over and above those provided for in the WNDCLASS structure. If used, this value must be a multiple of four. This means that any value stored as an extra class word must be a 32-bit integer.

Extra class words are accessed and set using the Windows CE functions *GetClassLong* and *SetClassLong*. A value assigned to a particular extra class word is the same for all instances of the class.

For example, let's say that every window based on a particular window class needs to have the same caption. This window caption is an

Table 2.1 Windows CE Window Class Styles

STYLE	MEANING
CS_DBLCLKS	Window receives double-click messages (corresponding to double-tapping on the device touch screen).
CS_GLOBALCLASS	Instances of the window class can be created by applications that are not in the same module (.EXE or .DLL) as the window class.
CS_HREDRAW	Redraws the entire window if a movement or size adjustment changes the width of the client area.
CS_NOCLOSE	Close command on the system menu is disabled.
CS_PARENTDC	Sets the clipping region to that of the parent window. This lets instances of this class draw on their parent.
CS_VERDRAW	Redraws the entire window if a movement or size adjustment changes the height of the client area.

attribute that is constant across all instances of the window class. A pointer to the window caption string could therefore be stored as an extra window class word.

Similarly, *cbWndExtra* can be used to specify extra bytes to be allocated for each instance of a window class. In this way, applications can assign unique attributes to each window using the *GetWindowLong* and *SetWindowLong* functions. Each instance of a window class can thus have different values for a particular extra window word. *cbWndExtra*, like *cbClsExtra*, must be a multiple of four.

hInstance identifies the HINSTANCE of the Windows CE module that contains the window procedure of the class.

hIcon and *hCursor* are the icon and mouse cursor to use with instances of the class, respectively. These members can be NULL.

hbrBackground is used to represent the background color of windows based on the particular window class.

The *lpszMenuName* member of WNDCLASS is not supported under Windows CE and therefore must be NULL. This does not exactly mean that Windows CE windows cannot have menus. But menus are added to windows quite differently under Windows CE than under other Windows platforms. Menus in Windows CE are included in *command bar controls*. Command bars are the subject of Chapter 4.

NOTE

ALL WINDOW CLASSES ARE GLOBAL

Under Windows CE, all window classes are global by default. The `CS_GLOBALCLASS` style is included for compatibility with other Windows platforms.

Registering a Window Class

Simply defining a window class with a WNDCLASS structure does not mean that the class can be used to make instances of that class. Before a Windows CE application can create a window of a particular window class, the window class must be registered.

Window class registration is the mechanism by which the class is made available to the Windows CE operating system. *CreateWindow*

requires the window class description in order to successfully create a window instance.

The function used to register a window class is *RegisterClass*:

```
RegisterClass(const WNDCLASS *lpWndClass);
```

RegisterClass takes one argument, a pointer to the WNDCLASS structure representing the window class to be registered. If the function succeeds, it returns an atom representing the registered class. Otherwise it returns zero. An atom is an integer that uniquely identifies a string. In this case, the string identified is the name of the window class, specified in the *lpszClassName* member of the *lpWndClass* parameter.

NOTE

REGISTERCLASSEX IS NOT SUPPORTED

The function *RegisterClassEx* is not supported under Windows CE.

The Window Procedure

Windows CE uses the same message-based architecture that the desktop Windows platforms such as Windows NT use. This means that the Windows CE operating system interacts with the windows in the various applications it is running by sending Windows CE messages.

The window procedure for a given window class is the function that implements the response of each instance of that class to every Windows CE message that it receives. Every window class has a window procedure. It can be implemented by the application programmer, in the case of application-defined window classes. In other cases the window procedure is part of Windows CE. For example, all of the child control window classes such as buttons and list boxes have window procedures that are implemented in the Graphics, Windowing, and Events Subsystem of Windows CE.

Window procedures have the following function signature:

```
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

The first parameter is the HWND of the window to which a specific Windows CE message is sent. *uMsg* is a UINT containing the message identifier of the message. *wParam* and *lParam* are 32-bit parameters whose values depend on the message being sent.

A window procedure is a *callback function* (as indicated by the `CALLBACK` specifier in the function prototype). This means that the function is called by Windows CE, instead of being called directly by an application.

For example, if an application wants to get the font being used by a particular window, it does so by sending the `WM_GETFONT` message to the window in question. Windows CE then calls the window procedure of the window class from which the window is derived:

```
HFONT hFont;  
hFont = (HFONT)SendMessage(hwndSomeWindow, WM_GETFONT,  
0, 0L);
```

Window procedures return an `LRESULT`, which is simply a `LONG` integer. This return value allows a window procedure to return message-specific information for any message posted or sent to it. If the message is sent via *PostMessage* or *SendMessage*, this result is passed back to the sender through the return value of these functions. The *SendMessage* and *PostMessage* functions are discussed in detail later in this chapter.

Window procedure return values can indicate success or failure of a message, or return requested information. In the `WM_GETFONT` example above, the value returned from the window procedure of *hwndSomeWindow* is the current window font.

As an application programmer, you will be implementing window procedures of your own. These will often be for application main windows, but you will also implement them for custom controls and other Windows CE window classes that you design. It would seem that it would be quite a challenge to implement responses for all of the hundreds of Windows CE messages that might be sent to your windows.

Luckily, Windows CE takes care of a lot of the default message handling for you. The function *DefWindowProc* is used to call Windows CE and have the operating system provide default behavior for any specified message:

```
DefWindowProc(hwnd, uMsg, wParam, lParam);
```

DefWindowProc has the same arguments as any window procedure. You pass it the corresponding parameters from your window procedure, and Windows CE performs the default processing for the given message. This greatly simplifies the implementation of window

behavior. It allows you to concentrate on the messages that have a unique or specific meaning to your particular window classes, and not think about the rest.

For example, in the template application, the only message that we handle ourselves is `WM_LBUTTONDOWN`. We want the template application to shut down whenever the user taps in the client area of the main window. The window procedure for the main application window class therefore look like this:

```
LRESULT CALLBACK WndProc(  
    HWND hwnd,  
    UINT message,  
    WPARAM wParam,  
    LPARAM lParam)  
{  
    switch(message)  
    {  
        case WM_LBUTTONDOWN:  
            DestroyWindow(hwnd);  
            PostQuitMessage(0);  
            return (0);  
        default:  
            return (DefWindowProc(hwnd, message, wParam, lParam));  
    }  
}
```

All of the several hundred Windows CE messages that can possibly be sent or posted to this window are handled by these few lines of code. All but one are handled by *DefWindowProc*.

NOTE

EACH MESSAGE RETURNS A VALUE

For every message handled by a window procedure, some appropriate value must be returned. The Windows CE on-line documentation specifies the values to return for each message under various circumstances.

Windows implement their specific behavior by responding to the various Windows CE messages. There are literally hundreds of messages in Windows CE, representing user input, window painting, and updating, and all of the other interactions that go on in Windows CE applications. In some sense, the essence of Windows CE programming is mastering these messages and their meanings, and implementing the responses to them to enable your applications to behave in the

ways that make them unique and interesting. This book is full of sample applications that will help you get started.

Creating Windows

So much for the window theory. How does an application actually create windows?

Windows CE provides two functions, *CreateWindow* and *CreateWindowEx*, for this purpose. These functions are used extensively in Windows CE applications to create everything from main application windows to child and common controls.

CreateWindow has the following form:

```
CreateWindow(lpClassName, lpWindowName, dwStyle, x, y,  
            nWidth, nHeight, hWndParent, hMenu, hInstance, lpParam);
```

CreateWindow creates an instance of a particular window class. *lpClassName* specifies the window class to use. This parameter is the Unicode string name of the class that was used to register the window class (i.e., the *lpszClassName* member of the WNDCLASS structure).

lpWindowName points to a null-terminated Unicode string that contains the window text. For example, in a window with a caption, this string is used as the caption text. For a button, the string is the button text.

dwStyle is a set of one or more window styles. (See Table 2.3 later for a complete list of window styles supported under Windows CE.)

x, *y*, *nWidth*, and *nHeight* specify the position and dimensions of the window: *x* and *y* are the x and y coordinates of the upper left corner of the window in screen coordinates. *nWidth* and *nHeight* are the width and height in screen coordinates.

hWndParent is the parent window of the window being created.

hMenu should be NULL for top-level windows. For child windows, such as Windows CE controls, this parameter specifies the child window identifier. This is the value used, for example, to identify the control sending a WM_COMMAND message.

hInstance is the HINSTANCE of the module in which the window is created.

Finally, *lpParam* is the value sent as the *lpCreateParams* member of the `CREATESTRUCT` sent with the `WM_CREATE` message that is triggered by the *CreateWindow* call. This parameter can be `NULL`. We take a closer look at the `WM_CREATE` message in the next section.

I should point out here that all text in Windows CE is Unicode, not ANSI. Therefore, any string function parameter will be Unicode for Windows CE. All text rendered in any application is Unicode. In short, if it's text, it's Unicode.

NOTE

`CREATEWINDOW` AND `CREATEWINDOWEX` ARE NOT REALLY FUNCTIONS

CreateWindow and *CreateWindowEx* are macros that call the *CreateWindowExW* function, the Unicode-based window creation function. Applications should never call *CreateWindowExW* directly, since porting this code to other versions of Windows may be problematic. For example, as Windows NT applications can be built with Unicode support disabled, *CreateWindowExW* may not be defined. The *CreateWindow* and *CreateWindowEx* macros, however, are guaranteed to resolve to the correct supported API.

CreateWindowEx is the same as *CreateWindow* except that it allows you to specify various *extended window styles* in the first parameter. The rest of the parameters are the same. The extended window styles supported under Windows CE are listed in Table 2.4 (shown later).

The `WM_CREATE` Message

The *CreateWindow* and *CreateWindowEx* functions both send a `WM_CREATE` message to the window procedure of the window being created. This message is sent after the window has been created but before it becomes visible. Furthermore, the `WM_CREATE` message is sent before the *CreateWindow* and *CreateWindowEx* functions return.

As Table 2.2 shows, the `WM_CREATE` message passes a pointer to a `CREATESTRUCT` structure containing information about the window being created. Here is the definition of the `CREATESTRUCT` structure:

```
typedef struct tagCREATESTRUCT
{
    LPVOID lpCreateParams;
    HINSTANCE hInstance;
    HMENU hMenu;
    HWND hwndParent;
```

Table 2.2 The WM_CREATE Message Parameters

PARAMETER	MEANING
wParam	Not used
(LPCREATESTRUCT)lParam	Pointer to the CREATESTRUCT structure containing information about the window being created

```

int cy;
int cx;
int y;
int x;
LONG style;
LPCTSTR lpszName;
LPCTSTR lpszClass;
DWORD dwExStyle;
} CREATESTRUCT, *LPCREATESTRUCT;

```

The *lpCreateParams* member of *CREATESTRUCT* contains the *lpParam* passed to the *CreateWindow* or *CreateWindowEx* call that generated the WM_CREATE message. If you use the *lpParam* parameter of *CreateWindow* or *CreateWindowEx* to pass some application-specific value for use during window creation, your application extracts it from this member.

hInstance identifies the module (application or dynamic link library) that owns the window that was just created.

hMenu identifies the window menu. As we'll see later in Chapter 4, windows do not support menu bars under Windows CE. The *hMenu* member will therefore be NULL unless the window created is a child window. In that case this member will contain the child window identifier.

hwndParent contains the HWND of the parent of the newly created window.

cx, *cy*, *y*, and *x* are the width, height, y position, and x position of the window, respectively. For top-level windows such as overlapped or pop-up windows, these values are given in screen coordinates. For child windows, these coordinates are relative to the upper left corner of the window's parent.

The *style* and *dwExStyle* members are DWORD values containing the style and extended style bits defined for the newly created window. In other words, these members are exactly the same as the *dwStyle* and *dwExStyle* values passed to *CreateWindow* or *CreateWindowEx*.

Adding or Removing Styles after a Window Has Been Created

It is common to want to add or remove window styles or extended styles from windows after they have been created. You can do this using the *GetWindowLong* and *SetWindowLong* functions.

For example, let's say you want to disable scrolling in a window programmatically. An application would do this:

```
//hwndNoScroll is the window of interest
DWORD dwStyle;
//Get the current set of window style bits
dwStyle = GetWindowLong(hwndNoScroll, GWL_STYLE);
//Disable WS_VSCROLL, WS_HSCROLL
SetWindowLong(hwndNoScroll, GWL_STYLE,
(dwStyle & ~(WS_VSCROLL | WS_HSCROLL))
```

Finally, *lpszName* and *lpszClass* contain the window caption and window class name, respectively, of the newly created window.

The value that an application returns in response to the WM_CREATE message controls the value returned by the *CreateWindow* and *CreateWindowEx* functions that triggered the WM_CREATE message. If an application returns 0 (zero) in response to this message, *CreateWindow* and *CreateWindowEx* continue with their normal execution. When the functions finish, they return the HWND of the window that was created.

On the other hand, if the application returns -1 in response to WM_CREATE, the new window is destroyed and *CreateWindow* and *CreateWindowEx* return NULL.

Applications respond to the WM_CREATE message to implement custom window creation behavior or to take more control of the window creation process.

As a very contrived example, let's say that we are writing an application that includes a registered window class with a window procedure named *WideWndProc*. Our application wants to refuse to create any instance of this class that is not at least 300 pixels wide.

To implement this feature, the WM_CREATE message handler of *WideWndProc* would look like this:

```
LRESULT CALLBACK WideWndProc(
    HWND hwnd,
```

```
UINT message,  
WPARAM wParam,  
LPARAM lParam)  
{  
    switch(message)  
    {  
        case WM_CREATE:  
            LPCREATESTRUCT lpcs;  
            lpcs = (LPCREATESTRUCT)lParam;  
            if (lpcs->cx < 300)  
            {  
                return (-1);  
            }  
            return (0);  
            ...  
    }  
}
```

For future reference, Table 2.3 lists all of the window style values that can be specified in calls to `CreateWindow`. Similarly, the extended style values that can be used in `CreateWindowEx` calls are shown in Table 2.4.

NOTE

MDI WINDOWS ARE GONE

Windows CE currently does not support Multiple Document Interface (MDI) windows.

NOTE

MAXIMIZED/MINIMIZED WINDOWS

Windows CE does not support the Windows NT/Windows 98 concepts of maximizing or minimizing windows. This is why you see none of the window styles for including a maximize or minimize box in the style tables in this chapter.

The Windows CE Application Entry Point

As you might remember, back in the days of C programming in non-Windows environments, programs started with a line that looked something like this:

```
void main(int argc, char** argv)
```

This function, *main*, was called the *program entry point*. To make a long story short, this was the function that the operating system called to start the program execution.

Table 2.3 Windows CE Window Styles

STYLE	MEANING
WS_BORDER	Window has a thin border.
WS_CAPTION	Window has a title bar. This style also includes the WS_BORDER style.
WS_CHILD	Window is a child window. Cannot be used with WS_POPUP style.
WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing occurs within window.
WS_CLIPSIBLINGS	Clips child windows relative to each other.
WS_DISABLED	Window is initially disabled when created.
WS_DLGFRAE	Window has a dialog box style border. Windows with this style cannot have title bars (i.e., cannot have WS_CAPTION style).
WS_GROUP	Identifies the window as the first in a group of controls.
WS_HSCROLL	Window has a horizontal scroll bar.
WS_OVERLAPPED	Window has a title bar and a border.
WS_POPUP	Creates a pop-up window.
WS_SYSMENU	Window has a system menu in its title bar. Such windows must also have the WS_CAPTION style.
WS_TABSTOP	Specifies a control that can receive the keyboard focus when the user presses the Tab key. Pressing the Tab key changes the keyboard focus to the next control with the WS_TABSTOP style.
WS_VISIBLE	Window is initially visible when created.
WS_VSCROLL	Window has a vertical scroll bar.

NOTE**THE WINAPI SPECIFIER**

The WINAPI specifier is an alias for the `_stdcall` calling convention.

The various Win32-based operating systems also need an entry point. For the Win32 operating systems, including Windows CE, the entry point is called *WinMain*. The prototype of *WinMain* is:

```
int WINAPI WinMain(hInstance, hPrevInstance,
    lpCmdLine, nCmdShow);
```

Under Windows CE, multiple copies, or instances, of an application may be launched at a time. The *hInstance* and *hPrevInstance* parameters

Table 2.4 Windows CE Extended Window Styles

EXTENDED STYLE	MEANING
WS_EX_NOACTIVATE	A top-level window created with this style cannot be activated. If a child window has this style, tapping it will not cause its top-level parent to be activated. A window that has this style will receive stylus events, but neither it nor its child windows can get the focus.
WS_EX_NOANIMATION	A window created with this style does not show animated exploding and imploding rectangles when created, closed, or deleted, and does not have a button on the taskbar.
WS_EX_CLIENTEDGE	Specifies that a window has a border with a sunken edge.
WS_EX_CONTEXTHELP	Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes to a question mark with a pointer. If the user then clicks a child window, the child receives a WM_HELP message. The child window should pass the message to the parent window procedure, which should call the WinHelp function using the HELP_WM_HELP command. The Help application displays a pop-up window that typically contains help for the child window.
WS_EX_CONTROLPARENT	Allows the user to navigate among the child windows of the window by using the Tab key.
WS_EX_DLGMODALFRAME	Creates a window that has a double border; the window can, optionally, be created with a title bar by specifying the WS_CAPTION style in the dwStyle parameter.
WS_EX_NODRAG	Creates a window that cannot be dragged
WS_EX_STATICEDGE	Creates a window with a three-dimensional border style, intended to be used for items that do not accept user input.
WS_EX_TOPMOST	Specifies that a window created with this style should be placed above all non-topmost windows and should stay above them, even when the window is deactivated. To add or remove this style, use the SetWindowPos function.
WS_EX_WINDOWEDGE	Specifies that a window has a border with a raised edge.

are both HINSTANCE values. *hInstance* is the handle of the current application instance. *hPrevInstance* is always NULL.¹

¹In older versions of Windows, if multiple instances of an application were running, *hPrevInstance* specified the instance of an application that was launched prior to the current one. Under Windows CE you can determine if another instance is running by calling the *CreateMutex* API function and then calling *GetLastError*. If *GetLastError* returns the error code ERROR_ALREADY_EXISTS, there is another instance of the application running.

Windows CE Non-Client Messages

Unlike other Win32-based platforms, Windows CE does not expose any of the non-client area window messages to the application programmer. Non-client area operations, such as painting the non-client area and non-client area stylus tap hit testing, are all performed exclusively by the operating system.

This means that your Windows CE applications cannot include handling for any non-client messages. For example, the following code in the window procedure of a main application window would result in a compilation error:

```
switch(message)
{
case WM_NCPAINT:
//Custom window border drawing code
return (0);
...
}
```

Specifically, the compiler will issue an error 2065 (undeclared identifier) when it tries to compile the line that contains `WM_NCPAINT`. Not only are the non-client messages not forwarded to the window procedure, they are actually excluded from the `WINUSER.H` public header file.

lpCmdLine is a null-terminated Unicode string containing the command line with which the application was launched, if any.

nCmdShow specifies how the main application window is to be shown.

NOTE

WINMAIN SIGNATURE IS A BIT DIFFERENT IN WINDOWS CE

Please note this subtle difference between the Windows CE *WinMain* signature and that for other Win32 platforms: The *lpCmdLine* parameter under Windows CE is a `LPTSTR`, whereas on Windows NT and Windows 98 it is an `LPSTR`.

The Message Loop

How do Windows CE messages end up getting to your window procedure if you never call your window procedure directly?

Messages can get to windows in a couple of different ways. One common way is for an application or Windows CE to send messages directly to a window using the *SendMessage* function:

```
SendMessage(hwnd, uMsg, wParam, lParam);
```

Does this look familiar? The parameters of *SendMessage* are the same as the parameters of any Windows CE window procedure. This is because *SendMessage* immediately turns around and calls the window procedure of the window class of which *hwnd* is an instance.

For example, if I wish to change the text in a button in one of my applications, I can do something like this:

```
SendMessage(hwndButton, WM_SETTEXT, 0,  
(LPARAM)(LPCTSTR)TEXT("New Text"));
```

SendMessage has the same function signature as a window procedure. In fact, *SendMessage* calls the window procedure for the specified window. It does not return until the message specified in the second parameter is processed by the window procedure of the window to which the message is sent.

SendMessage processes messages *synchronously*. That is, the message is processed by the window it is sent to immediately. The other way that Windows CE handles messages is *asynchronously*. Many messages, such as requests to update or repaint a window or notifications that the user has tapped the touch screen, are often sent by the operating system to an application faster than the application can process them.

For this reason, when an application starts running, Windows CE creates a *message queue* for that application.² The message queue is a place where messages can be put by the operating system or an application to be processed asynchronously, that is, when the application gets around to it.

To process asynchronous messages, an application implements a *message loop*. This is a simple piece of code that continuously looks for messages in the application's message queue. When the message loop finds a message, it gets processed. Otherwise the message loop just keeps on looping.

²Actually, Windows CE creates a message queue for every thread created by an application. But for the sake of this discussion, we'll think of each application as having only its main thread. The meaning of the application's message queue is therefore unambiguous.

A typical message loop looks like this:

```
while (GetMessage(&msg, NULL, 0, 0) == TRUE)
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

That's a pretty short while loop for processing a whole lot of messages. The *GetMessage* function gets a message from the message queue. This message is contained in a message structure of type *MSG*. This structure includes information such as which message was sent and which window it was intended for. Once a message is retrieved, *DispatchMessage* calls the window procedure corresponding to the window class of the window specified in the message structure.

GetMessage returns *TRUE* until it receives a *WM_QUIT* message from the message queue. The *GetMessage* function has the following syntax:

```
GetMessage(lpMsg, hWnd, wParamFilterMin, wParamFilterMax);
```

The *lpMsg* parameter to *GetMessage* is a pointer to a message structure that receives the information about the message retrieved from the message queue. *hWnd* specifies the window for which messages are to be retrieved. In other words, only messages sent to the specified window are removed from the message queue. If this parameter is *NULL*, messages for any window created by the calling thread are retrieved. *wParamFilterMin* and *wParamFilterMax* specify a range of window message identifiers to look for. Setting these both to zero tells *GetMessage* to look for all messages.

We skipped over the *TranslateMessage* step. This function converts virtual key messages into regular key messages before they are dispatched by *DispatchMessage*. We glossed over this because the main point of the discussion is how messages get pulled from the message queue and sent off to the proper window procedure. Once translated, virtual key messages get handled just like any other messages. The use of the *TranslateMessage* function is discussed when we introduce the concept of accelerator tables in Chapter 4.

Adding an asynchronous message to the message queue is called *posting* the message. As was mentioned above, Windows CE posts many messages to an application. However, applications can also post messages to an application or thread using the *PostMessage* function.

The MSG Structure

The `MSG` structure used by functions like `GetMessage` contains all of the information about a message that was posted to an application's message queue. The structure is defined as:

```
typedef struct tagMSG { // msg
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG;
```

The first four members of this structure are the same parameters that ultimately get sent to the window procedure. *time* specifies the time at which the message was posted. *pt* takes on a meaning under Windows CE that is somewhat different from its meaning under other Win32 platforms. Since Windows CE devices don't use a mouse, there is no concept of a current point. Therefore, rather than indicating the current cursor position, *pt* indicates the last point touched by the user on the touch screen before the message was posted.

`PostMessage` has the same argument list as `SendMessage`. The message posted is added to the message queue of the thread that created the window specified by the *hwnd* parameter.

The Template Application

Now that we have a basic understanding of how windows are represented in Windows CE and how they respond to messages, we can present the template application, shown in Figure 2.1.

This application creates a main application window and nothing more. Pressing the left mouse button when the cursor is anywhere inside the window while running the application in emulation (or tapping on it with the stylus if it's running on a real device) terminates the application. Terminating the application is accomplished with a call to the Windows CE API `PostQuitMessage`.

You can use this application as the foundation for other real Windows CE applications that you write. It contains all of the boilerplate needed

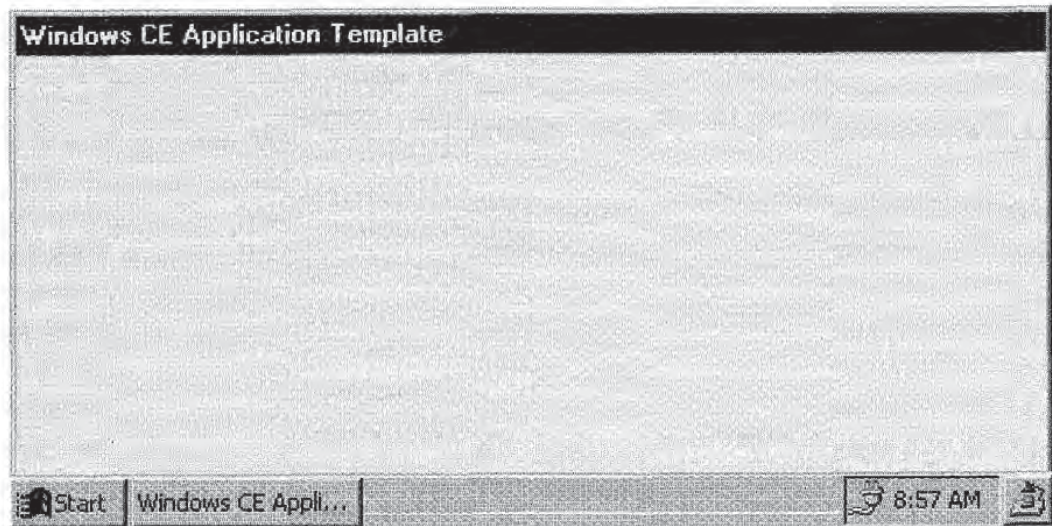


Figure 2.1 The Windows CE Template application.

for any Windows CE application: a *WinMain* function, a message loop, and a main window procedure. Of course you will probably replace the `WM_LBUTTONDOWN` handler with code of your own.

The project files for this application can be found in the `\Samples\template` directory of the companion CD. The application that results from building the project is called `TEMPLATE.EXE`. The complete source code for the template application is shown in Figure 2.2.

template.h

```
#ifndef __TEMPLATE_H_
#define __TEMPLATE_H_
TCHAR szAppName[] = TEXT("TEMPLATE");
TCHAR szTitle[] = TEXT("Windows CE Application Template");
/* Define the global application HINSTANCE */
HINSTANCE ghInst;
/* Define the HWNDs used by this application
   hwndMain -> Main application window
*/
HWND hwndMain;
/* Define the main application window procedure */
LRESULT CALLBACK WndProc(HWND hwnd,
                        UINT message,
```

```
        WPARAM wParam,  
        LPARAM lParam);  
#endif
```

main.cpp

```
#include <windows.h>  
#include "template.h"  
int WINAPI WinMain(HINSTANCE hInstance,  
                  HINSTANCE hPrevInstance,  
                  LPTSTR lpCmdLine,  
                  int nCmdShow)  
{  
    MSG msg;  
    WNDCLASS wndClass;  
    /* Save application instance in ghInst for  
    possible use by other functions, such as  
    the main window's window procedure.  
    */  
    ghInst = hInstance;  
    /* Register the main window class */  
    wndClass.style = 0;  
    wndClass.lpfnWndProc = WndProc;  
    wndClass.cbClsExtra = 0;  
    wndClass.cbWndExtra = 0;  
    wndClass.hInstance = hInstance;  
    wndClass.hIcon = NULL;  
    wndClass.hCursor = NULL;  
    wndClass.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);  
    wndClass.lpszMenuName = NULL;  
    wndClass.lpszClassName = szAppName;  
    RegisterClass(&wndClass);  
    /* Create the application's main window */  
    hwndMain = CreateWindow(szAppName,  
                           szTitle,  
                           WS_VISIBLE|WS_OVERLAPPED,  
                           0,0,  
                           GetSystemMetrics(SM_CXSCREEN),  
                           GetSystemMetrics(SM_CYSCREEN),  
                           NULL,  
                           NULL,  
                           hInstance,  
                           NULL);  
    while (GetMessage(&msg, NULL, 0, 0) == TRUE)  
    {  
        TranslateMessage(&msg);  
        DispatchMessage(&msg);  
    }  
    /* Return the wParam associated with the WM_QUIT message  
    that got us out of the while loop above. This wParam
```

```
        contains the exit code passed to PostQuitMessage.
    */
    return (msg.wParam);
}
LRESULT CALLBACK WndProc(HWND hwnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    switch(message)
    {
    case WM_LBUTTONDOWN:
        DestroyWindow(hwnd);
        PostQuitMessage(0);
        return (0);
    default:
        return (DefWindowProc(hwnd, message,
            wParam, lParam));
    }
}
```

Figure 2.2 TEMPLATE.EXE source code.

Concluding Remarks

In this chapter we have covered the basic window and message handling concepts you need to understand in order to write Windows CE applications. To make your applications more useful, you will want to use Windows CE controls and dialog boxes to allow users to interact with your applications. The next chapter describes how to program Windows CE controls and various kinds of dialog boxes.

Controls and Dialog Boxes

In this chapter, we look at how to program some of the most fundamental ingredients of Windows CE applications. Specifically, this chapter introduces Windows CE child and common control programming. After that, it describes how to include dialog boxes in your applications. It finishes with an introduction to the Windows CE common dialog library.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

Program child controls

Program common controls

Use modal and modeless dialogs

Use the Windows CE common dialogs

Programming Child Controls

A crucial part of almost every Windows CE user interface is the set of *child controls* with which users interact. Windows CE child controls

include push buttons, list boxes, and edit controls, as well as all of the other control types listed in Table 3.1.

Child controls are created by calling *CreateWindow* or *CreateWindowEx*, just as top-level application windows are created. The window class name to use for a given control type is given in Table 3.1.

Child controls communicate with their parent window by sending *WM_COMMAND* messages. These messages include the *control identifier* associated with the control, and a *notification code* that indicates what sort of action the user performed with the control. All of this information is sent to the parent window in the *WM_COMMAND* message parameters, as shown in Table 3.2.

The command identifier is an integer specified in the *hMenu* parameter of the *CreateWindow* call. For example, to create a push-button control with command identifier *IDC_BUTTON* and the string "Exit" inside, you could write:

```
#define IDC_BUTTON 1028
HWND hwndButton;
hwndButton = CreateWindow(
    TEXT("BUTTON"),           //Control class name
    TEXT("Exit"),            //Button text
    WS_CHILD|WS_VISIBLE|WS_PUSHBUTTON, //Button styles
    0,0,75,35,               //x, y, width, height
    hwndParent,              //Parent window
    (HMENU)IDC_BUTTON,       //Command identifier
    hInstance,               //Application HINSTANCE
    NULL);
```

Note that both the control window class name and the button text specified are Unicode strings.

Table 3.1 Windows CE Child Control Classes

CONTROL	WINDOW CLASS NAME
Button control	BUTTON
Edit control	EDIT
Combo box control	COMBOBOX
List box control	LISTBOX
Scroll bar control	SCROLLBAR
Static control	STATIC

Table 3.2 The WM_COMMAND Message

PARAMETER	MEANING
HIWORD(wParam)	Notification code. If the message is from a menu item, this value is 0.
LOWORD(wParam)	Specifies the command identifier of the control (or menu item) sending the WM_COMMAND message.
(HWND)lParam	The HWND of the control sending the message. If the WM_COMMAND message is not sent by a control, this value is NULL. For example, this is the case if the message is sent by a menu item.

Responding to WM_COMMAND Messages

As noted earlier, controls send WM_COMMAND messages to their parent windows to tell them that some action has taken place. For example, when a button is pushed, the button sends a WM_COMMAND message with a notification code BN_CLICKED.

The window procedure of the parent window of the button IDC_BUTTON we created in the previous section might include the following code. This code implements the window's response to the button being pressed:

```

LRESULT CALLBACK WndProc(
    HWND hwnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    switch(message)
    {
        //Other window messages
        //...
        case WM_COMMAND:
            UINT nID;
            nID = LOWORD(wParam);
            switch(nID)
            {
                case IDC_BUTTON:
                    DestroyWindow(hwnd);
                    PostQuitMessage(0);
                    break;
                default:
                    break;
            }
            //End of switch(nID) statement
    }
}

```



```

    return (0);
default:
    return (DefWindowProc(hwnd, message, wParam, lParam));
} //End of switch(message) statement
)

```

The identifier of the control sending the WM_COMMAND message is assigned to *nID*. The switch statement that follows then tests for the identity of the control, and performs whatever action that control specifies. In this case, the Exit button terminates the application.

From the example above, you can see that applications should return zero if they handle the WM_COMMAND message.

Notice that we ignored the notification code. Although not perfectly legitimate, this is usually the only button control notification of interest to applications. It is therefore common practice to ignore the notification code in the case of button WM_COMMAND messages.

At other times the notification code is important. For example, assume that you want to know when an edit control receives input focus. Assuming the edit control command identifier is specified by IDC_EDIT, your application could test for this in the parent window procedure as follows:

```

LRESULT CALLBACK WndProc(
    HWND hwnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    switch(message)
    {
        //Other window messages
        //...
        case WM_COMMAND:
            UINT nID, nNotify;
            nID = LOWORD(wParam);
            switch(nID)
            {
                case IDC_EDIT:
                    nNotify = HIWORD(wParam);
                    if (nNotify == EN_SETFOCUS)
                    {
                        //Do something
                    }
                    break;
            }
        default:
            break;
    }
}

```

```

    } //End of switch(nID) statement
    return (0);
default:
    return (DefWindowProc(hwnd, message, wParam, lParam));
} //End of switch(message) statement
}

```

For a complete list of the notification codes sent by the various Windows CE child controls, refer to the Microsoft Developer Studio Windows CE on-line documentation.

The complete sample application from which the examples in this section come can be found on the companion CD in the directory \Samples\controls.

Programming Common Controls

Back in the days before there were 32-bit versions of Windows, the Windows user interface was limited to the child controls. Custom controls could be implemented by ambitious programmers. But the core control set was the child controls.

When Windows 95 and Windows NT came out, however, a brand new set of controls was included with the operating systems. The common control library contains the controls that were added to Windows for Windows 95 and NT. Today, this library is still around, and there is a version of it for Windows CE. The library includes controls like list view controls and tree view controls. Table 3.3 lists the Windows CE

Table 3.3 The Windows CE Common Control Classes

CONTROL	WINDOW CLASS NAME
Date Time Picker	DATETIMEPICK_CLASS
Header control	WC_HEADER
Month calendar control	MONTHCAL_CLASS
Progress bar	PROGRESS_CLASS
Rebar control	REBARCLASSNAME
Tab control	WC_TABCONTROL
Trackbar control	TRACKBAR_CLASS
Tree view control	WC_TREEVIEW

common control classes. Note that Table 3.3 does not include those controls that have their own unique API. Such controls are not created using *CreateWindow*, and hence the window class names of these controls are not included.

Programming these controls is very similar to using the child controls. However, you must link with the common control library and initialize this library from your application. Also, the common controls do not communicate with their parent windows via *WM_COMMAND* messages. These controls send notifications by means of the *WM_NOTIFY* message.

NOTE

WINDOWS CE COMMON CONTROLS PROGRAMMING DETAILS

This section is only intended as an introduction to the mechanism by which common controls communicate with applications. Refer to Chapter 5 for a more detailed description of programming the Windows CE common controls.

Using the Common Controls Library

To use any of the Windows CE common controls, an application must link with the library *COMMCTRL.LIB*. It must then initialize the library by calling the function *InitCommonControls*.

Calling *InitCommonControls* further requires the application to include the header file *COMMCTRL.H*. *InitCommonControls* is typically called in an application's *WinMain* function.

Responding to Common Control Notifications

Common control notifications are a bit more complex than notifications sent by child controls. Notifications are sent in the form of the *WM_NOTIFY* message. This message includes a pointer to an *NMHDR* structure, which contains information about the notification being sent.

```
typedef struct tagNMHDR {
    HWND hwndFrom;
    UINT idFrom;
    UINT code;
} NMHDR;
```

hwndFrom is the window handle of the common control sending the notification. *idFrom* is the identifier of the control.

code indicates the notification code identifying the particular notification being sent. This value is used like the notification code sent by a child control.

Let's look at a typical example of how an application responds to common control notifications. Assume that a main window wants to know when the selected tab of a tab control in that window is changed. The main window procedure would respond to the WM_NOTIFY message as follows:

```
LRESULT CALLBACK WndProc(
    HWND hwnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    switch(message)
    {
        //Other window messages
        //...
        case WM_NOTIFY:
            LPNMHDR lpmhdr;
            lpmhdr = (LPNMHDR)lParam;

            switch(lpmhdr->idFrom)
            {
                case IDC_TAB:
                    switch(lpmhdr->code)
                    {
                        case TCN_SELCHANGE:
                            //Perform some action
                            break;
                        default:
                            break;
                    } //End of switch(lpmhdr->code) statement
                    break;
                default:
                    break;
            } //End of switch(lpmhdr->idFrom) statement
            return (0);
        default:
            return (DefWindowProc(hwnd, message, wParam, lParam));
    } //End of switch(message) statement
}
```

The window procedure determines which control is sending the notification by looking at the command identifier in *lpmhdr->idFrom*. It then checks the notification code and responds accordingly to the tab control notification it is interested in.

The complete sample application from which this code sample comes is found on the companion CD in the directory \Samples\lab.

Dialog Boxes

Many operations in Windows CE applications require user input to perform properly. Opening or saving files generally requires that the user specify a file name. To search a file for a specific string, a user enters text that the application uses to perform the search. A *dialog box* is a window through which users enter information required by an application to perform some task.

Dialog boxes can be either *modal* or *modeless*. A modal dialog box is displayed temporarily to accept user input (Figure 3.1). An important characteristic of a modal dialog box is that its owner window is disabled while the dialog box is present. Thus a modal dialog gives the effect of suspending an application until the user dismisses it.

A modeless dialog box, on the other hand, does not prevent users from interacting with the owner window (Figure 3.2). Modeless dialog boxes are often used in cases where the application may require frequent user input. Reopening the dialog box in such cases would be needlessly inconvenient. The dialogs that many applications use to provide text searching capability are often implemented as modeless dialog boxes.

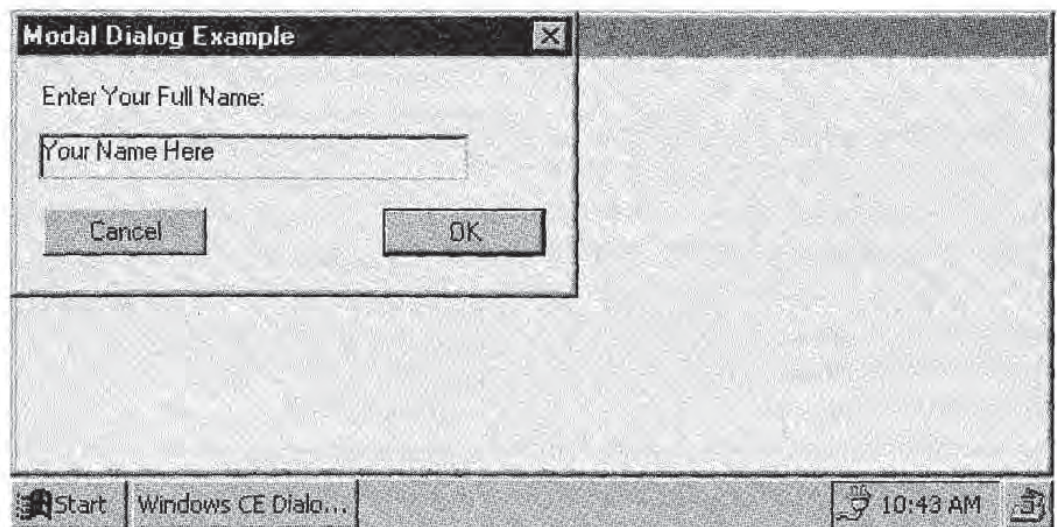


Figure 3.1 A modal dialog box example.

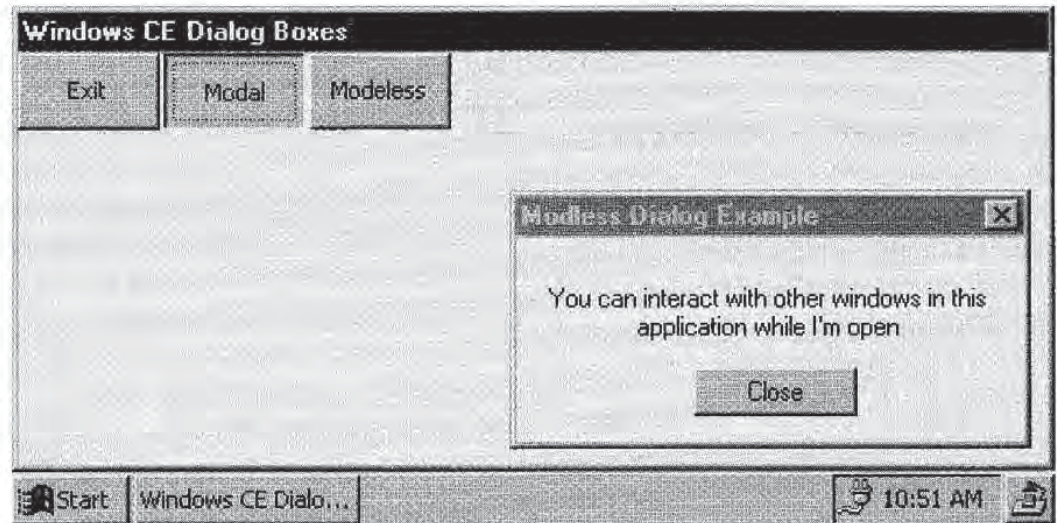


Figure 3.2 A modeless dialog box example.

Despite their differences, adding modal or modeless dialog boxes to your Windows CE applications involves the same basic steps. In the case of modeless dialogs, there are some additional programming requirements, but the basic idea is the same. Adding dialog boxes to an application involves these basic steps:

- Designing the dialog box and adding its dialog resource definition to the application's .rc file
- Programming the dialog procedure that handles Windows CE messages sent to the dialog box
- Invoking the dialog box at the appropriate times from the application

This chapter includes a sample application on the companion CD that demonstrates many of the concepts covered in this chapter. The application is called DIALOGS.EXE and can be found in the directory \Samples\dialogs. This application includes the implementations of the dialog boxes shown in Figures 3.1 and 3.2.

Dialog Box Resources

The appearance of the dialog boxes in Figures 3.1 and 3.2 was defined in a dialog box resource. The dialog box resource definition determines where the various controls appear. It also specifies the dialog box caption text, the dialog box size, and the font used when rendering text in the dialog box.

Dialog box resource definitions appear in the resource file (.RC file) of an application. Dialog box resources can be created manually or by means of the Microsoft Developer Studio Dialog Editor. The resource definition of the modal dialog box in Figure 3.1 is:

```
IDD_MODAL_DIALOG 0, 0, 170, 66
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Modal Dialog Example"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON "OK", IDOK, 112, 43, 50, 14
    PUSHBUTTON "Cancel", IDCANCEL, 8, 43, 50, 14
    LTEXT "Enter Your Full Name:", IDC_STATIC, 7, 7, 70, 8
    EDITTEXT IDC_NAME, 7, 22, 132, 14, ES_AUTOHSCROLL
END
```

The first line of this definition contains the resource identifier of the dialog (IDD_MODAL) and the dimensions of the dialog box in *dialog units*. Dialog units are defined in terms of the system font used by the particular Windows CE device. These units are interpreted by the system such that the dimensions of a dialog box are the same regardless of the resolution of the display.

Dialog box and dialog box child control identifiers are typically defined in the application's RESOURCE.H file. Some standard command identifier definitions, such as IDOK and IDCANCEL, are defined in the Windows CE header file WINUSER.H.

The next line defines the styles assigned to the dialog. Line 3 specifies the dialog box caption text, and line 4 specifies the font to be used by the dialog box.

The most interesting part of the dialog box resource definition lies between the BEGIN and END statements. The lines that appear between these statements specify the Windows CE controls that will appear in the dialog box.

The most general way to define a dialog box control is with the CONTROL statement:

```
CONTROL text, control identifier, class name,
        style, x, y, width, height
```

As an example, a push-button control with control identifier IDC_EXIT might be defined as follows:

```
CONTROL "Exit", IDC_EXIT, "BUTTON",  
    WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,  
    0, 0, 65, 35
```

Windows CE also defines a number of aliases for many of the child controls commonly used in dialog boxes. For example, in the case of the resource definition of the dialog in Figure 3.1, `DEFPUSHBUTTON` specifies a push button control that is the default button. The `PUSHBUTTON` control specifies a button control with style `BS_PUSHBUTTON`. The on-line documentation defines the full set of resource definition keywords.

The Dialog Box Procedure

To Windows CE, a dialog box is like any other window. Whenever an application creates a dialog box, deep in the implementation of Windows CE a call to `CreateWindowExW` is made. Similarly, dialog boxes respond to Windows CE messages. This implies that dialog boxes have a function for responding to messages similar to the window procedure discussed in the previous chapter.

A *dialog procedure* is an application-defined function that is assigned to a dialog box for responding to Windows CE messages. Programming a dialog procedure is very similar to coding the window procedure of a standard window. There are, however, a few subtle and very important differences.

When you use dialog boxes in your applications, you generally do not define a window class for the dialogs as you do for other windows. The dialog box window class for most types of dialog boxes available in Windows CE is defined by the operating system. This means that the true window procedure used by a dialog box is defined and provided by Windows CE.

The dialog procedure that you define as an application programmer and assign to a dialog box is simply a hook into the real dialog box window procedure. The dialog box window procedure defined in Windows CE receives all dialog box messages. The operating system then passes the messages to the application-defined dialog procedure, giving the application the first opportunity to handle them. If the application does not handle a particular message, it is handled by Windows CE.

A dialog procedure has the following prototype:

```
BOOL CALLBACK DlgProc(HWND hwndDlg, UINT message,  
    WPARAM wParam, LPARAM lParam);
```

This is almost exactly the same as the standard window procedure definition. The only difference is the return type. Dialog procedures return a `BOOL` instead of an `LRESULT`. This return value tells Windows CE whether or not to pass handling of a particular message on to the default dialog box message handler. Returning `FALSE` means that your dialog procedure did not handle a message. `TRUE` means it did.

This brings us to another important difference between dialog procedures and regular window procedures. A regular window procedure typically calls *DefWindowProc* to make Windows CE perform default processing for any unhandled Windows CE messages. A dialog procedure should return `FALSE` for any unhandled messages. The Windows CE API does include the function *DefDlgProc* for performing default message processing. This function should only be used in conjunction with private dialog classes.

Private Dialog Classes

The window classes associated with the majority of the dialog boxes you will use in your applications are defined by Windows CE. Invoking a modal dialog by calling *DialogBox* or displaying a message dialog with a call to *MessageBox* creates an instance of an operating system-defined window class.

This is generally adequate for most applications. Programmers create customized dialog boxes by specifying the contents and appearance of a dialog via a resource template. Custom behavior is provided by the dialog procedure implementation.

It is possible to completely define the dialog box window class within an application, however. *Private dialog class* is the term used to describe any such dialog class. A private dialog class is similar to a regular window class. The application defines the dialog procedure and specifies how all messages are handled. The Windows CE-defined dialog procedure plays no part in processing private dialog class messages.

Using private dialog classes in your applications is the only time you should ever call *DefDlgProc*. This function is the dialog box equivalent of *DefWindowProc*. It performs default processing for unhandled dialog messages. Calling this function from any other dialog procedure will cause your application to misbehave in very unexpected ways.

Table 3.4 The WM_INITDIALOG Message

PARAMETER	MEANING
(HWND)wParam	Window handle of the control to which Windows CE will assign default focus. Focus is assigned only if the dialog procedure returns TRUE in response to this message.
lParam	Initialization parameter passed by DialogBoxParam or DialogBoxIndirectParam.

The WM_INITDIALOG Message

A dialog box procedure typically implements custom responses to fewer messages than a standard window procedure. Most message processing for a typical dialog box is performed by Windows CE.

Since most dialog boxes are used for collecting user input, they generally respond to WM_COMMAND messages sent by their child controls. They also typically handle a message that is only sent to dialog boxes called WM_INITDIALOG. This message is used to initialize the contents of a dialog box, and is sent by Windows CE after the dialog box window is created but right before the dialog box is displayed. WM_INITDIALOG can be thought of as the dialog box equivalent of WM_CREATE. Table 3.4 lists the WM_INITDIALOG parameters and their meanings.

Returning FALSE after processing WM_INITDIALOG tells Windows CE not to set the default focus. *lParam* can contain an application-specific parameter used to initialize the dialog box. We will discuss this in greater detail later when we talk about how to invoke a dialog box.

An Example

Let's take a look at an example modal dialog procedure. This example comes from the sample application DIALOGS.EXE. The dialog box that corresponds to this code is shown in Figure 3.1. IDC_NAME is the command identifier of the edit control.

```
#define MAX_STRING_LENGTH 129
TCHAR pszUserText[MAX_STRING_LENGTH];
BOOL CALLBACK ModalDlgProc(HWND hwndDlg,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
```

```

UINT nID;
HWND hwndEdit;
switch(message)
{
case WM_INITDIALOG:
    hwndEdit = GetDlgItem(hwndDlg, IDC_NAME);
    SetWindowText(hwndEdit, TEXT("Your Name Here"));
    return (FALSE);
case WM_COMMAND:
    nID = LOWORD(wParam);
    switch(nID)
    {
    case IDOK:
        hwndEdit = GetDlgItem(hwndDlg, IDC_NAME);
        GetWindowText(
            hwndEdit,
            pszUserText,
            MAX_STRING_LENGTH);
    case IDCANCEL:
        EndDialog(hwndDlg, nID);
        break;
    default:
        break;
    } //End of switch(nID) block
    return (TRUE);
default:
    return (FALSE);
} //End of switch(message) block
}

```

This dialog procedure is pretty simple. The `WM_INITDIALOG` handler sets the edit control text to a default string. It uses the `GetDlgItem` function:

```
GetDlgItem(hwndDlg, nIDDlgItem);
```

This function returns the window handle of the control with control identifier `nIDDlgItem` contained by the dialog box specified by the window handle `hwndDlg`.

The `WM_COMMAND` handler responds to the OK button by reading the edit control text into the string `pszUserText` by calling `GetWindowText`. Both the OK and Cancel buttons end up closing the dialog box by calling `EndDialog`. More on this function in the next section.

The `WM_COMMAND` handler returns `TRUE` to tell Windows CE that this particular message has been handled. You will often see dialog box procedures return `FALSE` at the end of their `WM_COMMAND` handlers, instead of `TRUE`. Many programmers prefer this, since

returning `FALSE` is consistent with returning zero from a standard window procedure. And since the default `WM_COMMAND` processing is to do nothing, returning `FALSE` instead of `TRUE` produces no ill effects.

Invoking and Destroying Modal Dialogs

We have yet to discuss how an application displays a modal dialog box. Windows CE provides four functions for invoking modal dialog boxes. We will discuss the two most common of these functions.¹

The first of these functions is aptly named *DialogBox*:

```
DialogBox(hInstance, lpTemplate, hWndParent, lpDialogFunc);
```

This function displays the modal dialog specified by the resource template *lpTemplate*. *hInstance* is an `HINSTANCE` identifying the module that contains the dialog resource definition. *hWndParent* is the dialog box parent window. *lpDialogFunc* is a pointer to the dialog procedure to use with the modal dialog.

The second function is *DialogBoxParam*. This function works just like *DialogBox*, except that it includes an extra parameter:

```
DialogBoxParam(hInstance, lpTemplate, hWndParent,  
lpDialogFunc, lpInitParam);
```

The additional parameter *lpInitParam* is passed to the dialog procedure *lpDialogFunc* as the *lParam* of the `WM_INITDIALOG` message. This parameter can be used to pass application-specific initialization information to the modal dialog box.

A modal dialog is destroyed by calling *EndDialog*:

```
EndDialog(hDlg, nResult);
```

hDlg is the window handle of the dialog box. *nResult* specifies the value to be returned to the application by the function that invoked the dialog, such as *DialogBox* or *DialogBoxParam*. In other words, whatever value is passed to *nResult* is the value returned by *DialogBox* or *DialogBoxParam*.

¹The other two functions are *DialogBoxIndirect* and *DialogBoxIndirectParam*. These functions are used to display modal dialog boxes defined by dialog templates contained in program memory instead of in a resource file. This technique is uncommon enough to leave out of our discussion with no disservice to the reader.

This return value is typically used to tell the application which dialog box button was pressed. For example, here is how DIALOGS.EXE calls *DialogBox* to invoke the modal dialog whose resource identifier is `IDD_MODAL`. The return value is used to determine what action to perform after the dialog box is closed.

```
if (IDOK==DialogBox(ghInst, MAKEINTRESOURCE(IDD_MODAL),
    hwnd, (DLGPROC)ModalDlgProc))
{
    //Do something
}
```

Modeless Dialog Boxes

Modal dialogs are designed to collect user input and to not go away until the user dismisses them in some way. This behavior is no accident. When an application calls *DialogBox* (or any of the other functions that invoke a modal dialog box), Windows CE creates a new message loop specifically for the dialog box. This loop does not exit until the *EndDialog* is called for the dialog box. Hence the *DialogBox* call does not return until the dialog box is closed.

The fundamental difference between modeless dialogs and modal dialogs is that modeless dialog messages are put on the message queue of the thread that creates them. Therefore, modeless dialog box messages get dispatched by the message loop of the same thread that creates the dialog. In most cases, this means that the message loop in your application's *WinMain* function processes modeless dialog box messages.

The *IsDialogMessage* Function

Windows CE does not create a separate message loop for modeless dialogs or take care of the details of dispatching messages intended for modeless dialogs. So to use modeless dialogs, your application will clearly have to do more work than it does for modal dialogs.

This is where the *IsDialogMessage* function comes in. This function determines if a particular message is intended for the specified dialog. If so, *IsDialogMessage* processes the message.

```
IsDialogMessage(hwnd, lpMsg);
```

hwnd is the `HWND` of a modeless dialog, and *lpMsg* is a pointer to a `MSG` structure. If *IsDialogMessage* processes the message, it returns `TRUE`. Otherwise it returns `FALSE`.

This leads us to the time-tested technique for handling modeless dialog messages in message loops. Let's assume that a Windows CE application has created a modeless dialog with window handle *hwndModeless*. (We'll get to how to create modeless dialogs in the next section.) So that the application and the modeless dialog can share the same message loop, the message loop is rewritten as shown in Figure 3.3:

```
while (GetMessage(&msg, NULL, 0, 0) == TRUE)
{
    if (!hwndModeless ||
        !IsDialogMessage(hwndModeless, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Figure 3.3 Modifying the message loop to accommodate a modeless dialog box.

We have given *IsDialogMessage* the first chance to handle each and every message on the application's message queue. If the modeless dialog is not NULL and the message was indeed meant for the dialog, *IsDialogMessage* will process it and return TRUE.

IsDialogMessage evaluating to TRUE means that the code inside the if statement is skipped. If it returns FALSE, the message is processed in the normal way by *TranslateMessage* and *DispatchMessage*.

Creating and Destroying Modeless Dialog Boxes

The message loop modification discussed above is the most important thing to grasp about programming modeless dialog boxes. But we still have to see how to invoke and dismiss them.

There are four functions available for creating modeless dialogs, as there were for creating modal dialogs. We will discuss *CreateDialog* and *CreateDialogParam*. (*CreateDialogIndirect* and *CreateDialogIndirectParam* are analogous to *DialogBoxIndirect* and *DialogBoxIndirectParam*. These functions are not covered here. See the footnote in the section "Invoking and Destroying Modal Dialogs" for the reason.)

CreateDialog creates a modeless dialog and returns the HWND of the dialog if successful:

```
CreateDialog(hInstance, lpTemplate, hWndParent, lpDialogFunc);
```

The arguments have exactly the same meaning as they do in the function *DialogBox*.

Similarly, *CreateDialogParam* is used just like *DialogBoxParam*:

```
CreateDialogParam(hInstance, lpTemplate, hWndParent,
    lpDialogFunc, lpInitParam);
```

Again the difference is the return value. If successful, *CreateDialogParam* returns the HWND of the dialog it creates. Both *CreateDialog* and *CreateDialogParam* return NULL if they fail.

Dismissing modeless dialogs is done by calling *DestroyWindow*. An application should never use *EndDialog* to destroy a modeless dialog.

We can now see how to get the window handle of the modeless dialog required for the message loop. When using a modeless dialog, an application typically maintains a global HWND variable that is set to NULL as long as the modeless dialog does not exist. Once the dialog is created, the global variable is assigned the dialog's window handle.

The application DIALOGS.EXE provides an example. The application initializes *hwndModeless* to NULL. The if statement in the message loop of Figure 3.3 therefore never gets to call *IsDialogMessage*. All messages posted to the application's message queue therefore get handled the usual way.

But once the modeless dialog is created, *hwndModeless* is no longer NULL. The modeless dialog IDD_MODELESS is displayed when a user presses the Modeless button in the main application window. The portion of the main window procedure that responds to the corresponding WM_COMMAND message is:

```
HWND hwndModeless; //Modeless dialog window handle
//Inside the main window WM_COMMAND handler
case IDC_MODELESS_BUTTON:
    hwndModeless = CreateDialog(
        ghInst,
        MAKEINTRESOURCE(IDD_MODELESS),
        hwnd,
        (DLGPROC)ModelessDlgProc);
```

Messages intended for the dialog box are now processed by *IsDialogMessage*.

To dismiss the modeless dialog, the user taps the Close button. Here is how the modeless dialog procedure responds:

```
//Inside the dialog procedure WM_COMMAND handler
case IDCANCEL:
    DestroyWindow(hwndDlg);
    hwndModeless = NULL;
    break;
```

After destroying the actual dialog window, the global variable *hwndModeless* is set back to NULL. Thus the application's message loop doesn't send messages to *IsDialogMessage* until a user again creates the modeless dialog box.

The Windows CE Common Dialogs

If you have been using applications written for the various versions of Windows for awhile, you have no doubt noticed that certain features, such as opening and saving files, are often the same from one application to another.

In fact, software users have come to expect that the user interfaces for performing such operations will look the same. The widespread acceptance of computers is in part due to the fact that most users can figure out how to use new software quickly because the user interfaces for such fundamental features are often the same.

The common dialogs provide a way for Windows CE applications to quickly include a standard user interface for various operations common to many applications. Specifically, the standard dialog boxes for opening and saving files and choosing colors can be easily included in Windows CE applications.

Each of these dialogs has come to be known by a unique name. For example, the dialog for opening files is called the File Open dialog. Table 3.5 lists the common names for the dialogs in the Windows CE common dialog library.

Figure 3.4 shows an example of the File Open dialog in use.

Table 3.5 The Windows CE Common Dialogs

DIALOG NAME	USE
File Open dialog	Dialog used for opening files.
Save As dialog	Dialog used to save a file under a new name.
Color dialog	Dialog providing a user interface for selecting a color.

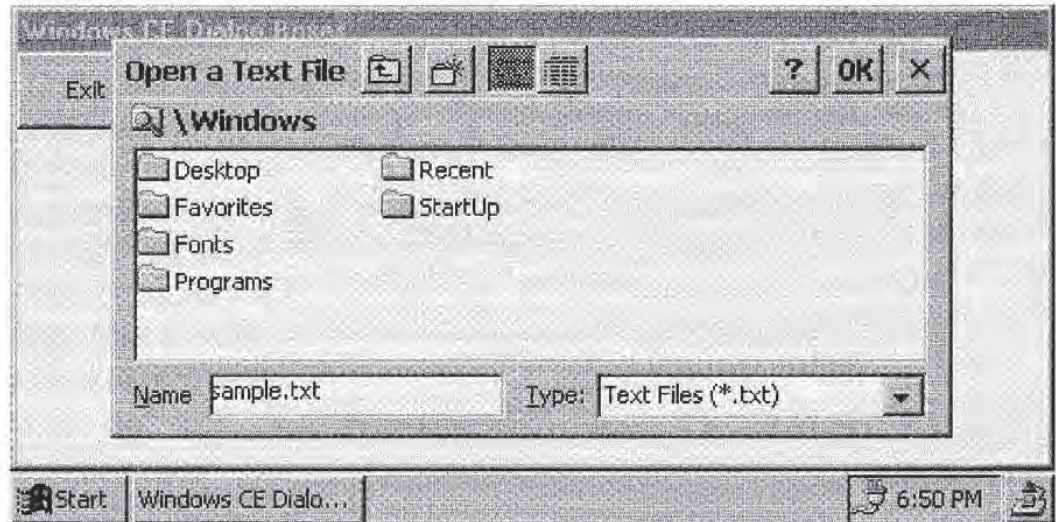


Figure 3.4 The File Open common dialog.

NOTE

COMMDLG.DLL

The Windows CE common dialog library is implemented in **COMMDLG.DLL**. To use the dialogs, an application must include the header file **COMMDLG.H** and link with **COMMDLG.LIB**. Under Windows CE emulation, the DLL is called **COMMDLGM.DLL**, and the import library to link with is **COMMDLGM.LIB**.

These dialogs are implemented in a library called the common dialog library, which is a part of the Graphics, Windowing, and Event Subsystem.

Each of the common dialogs is invoked by a single function call. Furthermore, each common dialog has an associated data structure that an application fills with parameters specifying various attributes about the dialog to be displayed.

For example, an application invokes the File Open dialog by filling an **OPENFILENAME** structure and then calling *GetOpenFileName*.

NOTE

COMPATIBILITY: *FINDTEXT* AND *CHOOSEFONT* ARE NOT SUPPORTED

The functions *FindText* and *ChooseFont* are not currently supported under Windows CE. This means that the corresponding Find Replace and Font Chooser common dialogs are not available.

NOTE**COMMON DIALOGS ARE MODAL**

All of the common dialogs supported under Windows CE are modal dialogs. The functions that invoke them therefore do not return until the user dismisses them by pressing the OK or Cancel (X) button.

Common Dialog Programming

The following sections give the details of how to program each of the common dialogs supported under Windows CE.

The File Open and Save As Dialogs

The File Open and Save As common dialogs under Windows CE look very similar. The difference between them is how your applications use the information that they provide.

The File Open common dialog is invoked by calling the function *GetOpenFileName*:

```
GetOpenFileName(lpofn);
```

The Save As dialog is invoked with *GetSaveFileName*:

```
GetSaveFileName(lpofn);
```

Each of these functions displays the respective dialog box. The functions do not return until the user dismisses the dialog by pressing the OK or Cancel (X) button.

The single argument to each of these functions is a pointer to an OPENFILENAME structure. This structure contains information used by *GetOpenFileName* or *GetSaveFileName* to control the appearance and behavior of the resulting dialog box.

The OPENFILENAME structure is defined as follows:

```
typedef struct tagOFN
{
    DWORD lStructSize;
    HWND hwndOwner;
    HINSTANCE hInstance;
    LPCSTR lpstrFilter;
    LPSTR lpstrCustomFilter;
```

```

    DWORD nMaxCustFilter;
    DWORD nFilterIndex;
    LPSTR lpstrFile;
    DWORD nMaxFile;
    LPSTR lpstrFileName;
    DWORD nMaxFileName;
    LPSTR lpstrInitialDir;
    LPCSTR lpstrTitle;
    DWORD Flags;
    WORD nFileOffset;
    WORD nFileExtension;
    LPCSTR lpstrDefExt;
    DWORD lCustData; //
    LPOFNHOOKPROC lpfnHook;
    LPCSTR lpTemplateName;
} OPENFILENAME;

```

This structure looks pretty complicated, but in reality you generally do not use all of the members. In fact four of them, *lCustData*, *lpfnHook*, and *lpTemplateName*, and *hInstance*, are never used by Windows CE. These members appear in the Windows CE definition of the OPENFILENAME structure because the definition was taken from the Windows NT implementation.

While reading the description below, keep in mind that this structure is used for both the File Open and File Save dialogs.

lStructSize must contain the size in bytes of the OPENFILENAME structure.

hwndOwner specifies the owner of the dialog box. This can be NULL if the dialog box has no owner.

lpstrFilter is used to specify the *filter strings* to be used when displaying the dialog box. These strings determine the contents of the Type: combo box (see Figure 3.4). They also tell the dialog box to display only those files with the specified extensions. This member can be NULL. You specify *lpstrFilter* as pairs of NULL-terminated strings. The first string in each pair is the descriptive text, such as the text "Text Files (*.txt)" that appears in the Type: field in Figure 3.4. The second string is the filter pattern, such as "*.htm".

lpstrCustomFilter is used by *GetOpenFileName* or *GetSaveFileName* to return the filter chosen by the user. This can be NULL. If not NULL, you must specify the length of the buffer that you are passing in the *lpstrCustomFilter* member in the *nMaxCustFilter* member.

nFilterIndex specifies the 1-based index of the pair of *lpstrFilter* strings with which to initialize the Type: combo box.

lpstrFile is used to specify the file name to initially display in the Name field. It also returns the name of the file specified by the user if the user presses the OK button.

nMaxFile specifies the size of the buffer passed in *lpstrFile*.

Similarly, *lpstrFileTitle* defines a buffer that receives the name of the selected file without the full path name. *nMaxFileTitle* is passed to specify the size of the buffer pointed to by *lpstrFileTitle*. *lpstrFileTitle* can be NULL, in which case *nMaxFileTitle* is ignored.

The *lpstrInitialDir* member is a string that specifies the directory whose contents are displayed by the File Open or Save As dialog when it first appears. This member can be NULL.

lpstrTitle is the string used as the dialog box caption text. This member can be NULL.

nFileOffset specifies a zero-based offset from the beginning of the path to the file name in the string pointed to by *lpstrFile*. Similarly, *nFileExtension* is the zero-based offset to the file extension.

lpstrDefExt is a string containing the default file extension. This is the extension that is appended to the selected file name if, for example, a user types a file name into the Name field but does not supply an extension. This member can be NULL.

We left out the *Flags* member. This member is used to specify a set of bit flags that determine various attributes of the File Open dialog box. This member is also used as a return value. *GetOpenFileName* returns one or more flags to report information about the user's input.

There are many flags that can be specified in the *Flags* member of an *OPENFILENAME* structure. The most common flags and their meanings are given in Table 3.6.

An Example

All of the foregoing has no doubt caused your eyes to glaze over. Let's take a look at how the *DIALOGS.EXE* application creates a File Open dialog to make it all more clear.

Table 3.6 OPENFILENAME Flag Values

FLAG	MEANING
OFN_CREATEPROMPT	If the user specifies a file that does not exist, this flag causes the dialog box to prompt the user for permission to create the file. If the user chooses to create the file, the dialog box closes and the function returns the specified name. Otherwise, the dialog box remains open. This flag is only used with <code>GetSaveFileName</code> .
OFN_EXTENSIONDIFFERENT	Specifies that the user typed a file-name extension that differs from the extension specified by <code>lpstrDefExt</code> . The function does not use this flag if <code>lpstrDefExt</code> is NULL.
OFN_FILEMUSTEXIST	Specifies that the user can type only names of existing files in the File Name entry field. If this flag is specified and the user enters an invalid name, the dialog box procedure displays a warning in a message box. If this flag is specified, the <code>OFN_PATHMUSTEXIST</code> flag is also used.
OFN_HIDEREADONLY	Hides the File Open dialog's read-only check box.
OFN_NOCHANGEDIR	Restores the current directory to its original value if the user changed the directory while searching for files.
OFN_NODEREFERENCELINKS	Directs the dialog box to return the path and file name of the selected shortcut (.LNK) file. If this value is not given, the dialog box returns the path and file name of the file referenced by the shortcut.
OFN_NONETWORKBUTTON	Hides and disables the Network button.
OFN_OVERWRITEPROMPT	Causes the Save As dialog box to generate a message box if the selected file already exists. The user must confirm whether to overwrite the file.
OFN_PATHMUSTEXIST	Specifies that the user can type only valid paths and file names. If this flag is used and the user types an invalid path and file name in the File Name entry field, the dialog box function displays a warning in a message box.
OFN_SHOWHELP	Causes the dialog box to display the Help button.

The most important part of creating the File Open dialog is filling an `OPENFILENAME` structure with the right values. To render the dialog in Figure 3.4, `DIALOGS.EXE` executes the following code in the function `OnFileOpen`:

```
void OnFileOpen(HWND hwnd)
{
    OPENFILENAME ofn;
```

```
TCHAR pszName[256];
DWORD dwSize;
dwSize = sizeof(ofn);
lstrcpy( pszName, TEXT("sample.txt"));
memset(&ofn, 0, dwSize);
ofn.hwndOwner = hwnd;
ofn.lStructSize = dwSize;
ofn.lpstrFilter = TEXT("Text Files (*.txt)\\0*.txt\\0");
ofn.nFilterIndex = 1;
ofn.lpstrFile = pszName;
ofn.nMaxFile = 256;
ofn.lpstrTitle = TEXT("Open a Text File");
ofn.lpstrInitialDir = TEXT("\\Windows");
ofn.lpstrDefExt = TEXT("txt");
GetOpenFileName(&ofn);
}
```

The *hwnd* parameter of *OnFileOpen* is the main application window. This value is the owner of the File Open dialog. Hence the *hwndOwner* member of *ofn* is set to this window.

The filter strings are assigned to *ofn.lpstrFilter*. The first null-terminated string is used as the contents of the dialog Type: field. The second specifies that the File Open dialog should only display files with the .txt extension.

nFilterIndex specifies the 1-based index of the filter string pair to initially use with the dialog. Setting *nFilterIndex* to 1 tells the dialog box to use the first filter string pair, the string "Text Files (*.txt)\\0*.txt\\0."

Assigning a string to *ofn.lpstrFile* means that the File Open dialog will be initialized with that string in the Name field of the dialog box. In this case, the file name "sample.txt" will initialize this field.

Finally, the string "Open a Text File" will be the File Open dialog box title, and "txt" will be used as the default file name extension. If a user types a file name in the name field without an extension, .txt will be automatically appended.

This is all the hard work. To display the dialog box, the application simply calls *GetOpenFileName*. When *GetOpenFileName* returns, the *lpstrFile* member of *ofn* contains the fully qualified file name of the selected file.

OnFileOpen doesn't do anything once a user selects a file. It just shows you how to use the File Open dialog. A real application would proceed by, for example, reading the file and displaying the contents in the main application window.

Programming the Save As common dialog is very similar. An OPENFILENAME structure is filled as in this example. The application then simply calls *GetSaveFileName* to display the dialog.

The Color Dialog

Now we take a very brief look at using the Color dialog. Figure 3.5 shows an example of the Color dialog. Applications use this dialog as a user interface for making color selections. Pressing the Define button causes the dialog to expand into the form shown in Figure 3.6.

This dialog box is created with one function call, just like all the other common dialogs. In the case of the Color dialog, you call the *ChooseColor* function:

```
ChooseColor(lpcc);
```

The single argument in this case is a pointer to a CHOOSECOLOR structure. The various members of this structure are initialized to control the appearance of the Color dialog.

To create the Color dialog in Figure 3.5, an application only needs six lines of code:

```
CHOOSECOLOR cc;
COLORREF cr[16];
memset(&cc, 0, sizeof(cc));
cc.lStructSize = sizeof(cc);
```

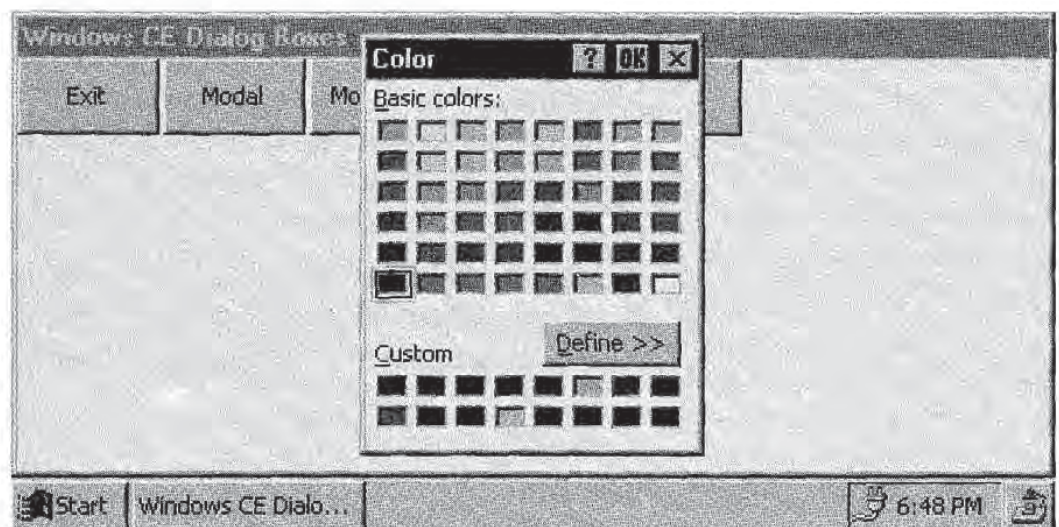


Figure 3.5 The Color common dialog.

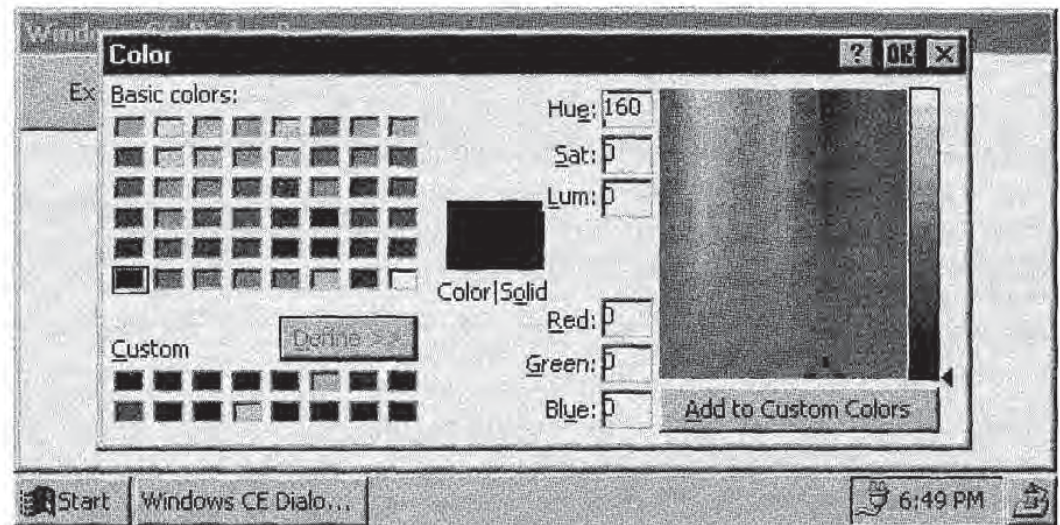


Figure 3.6 A fully expanded Color dialog.

```
cc.lpCustColors = cr;
ChooseColor(&cc);
```

The *lpCustColors* member of the `CHOOSECOLOR` structure can be used to specify the 16 colors displayed in the custom colors fields at the bottom of the dialog box in Figure 3.5. This allows the application to specify colors of its own for users to choose from other than the basic colors at the top.

Note that *lpCustColors* cannot be `NULL`. Since this member is used to return any custom colors specified by the user, it must be initialized with a valid `COLORREF` array.

These custom colors can also be changed by the user. If the `Define` button is pressed, the user sees the fully expanded Color dialog like the one in Figure 3.6. Colors can be entered in the Red, Green, and Blue edit boxes. Or the user can tap points on the large field above the `Add To Custom Colors` button to select colors. The hue, saturation, and luminance of the colors in this field can also be changed via the corresponding edit boxes.

The point of all of this is that if the user taps the `Add To Custom Colors` button, the color currently selected in the right half of the dialog is added to the *lpCustColors* array. This provides a mechanism for the application to store the user's custom color selections and put them in the custom color fields the next time the dialog is created.

When the user presses the OK button to close the Color dialog, the color selected by the user is returned to the application in the *rgbResult* member of the *CHOOSECOLOR* structure. This member can also be assigned before calling *ChooseColor* to specify which color is initially selected in the dialog.

Concluding Remarks

The two chapters you have just finished reading have presented an introduction to programming some of the most basic building blocks of any Windows CE application. We have seen how to write the fundamental message processing infrastructure of an application. We know how to add child and common controls, as well as various types of dialog boxes.

In the rest of Part I, we add to this knowledge by investigating how menus are added to Windows CE applications. And we look at how to program some of the common controls that are of particular interest to Windows CE applications.

Menus and the Windows CE Command Bar

One of the biggest challenges that Windows application programmers have traditionally faced is designing a user interface that is intuitive and easy to use. Keeping the computer screen organized and free of the clutter of lots of buttons and other controls gets more difficult as Windows programs become more complex and feature-rich.

If this is a challenge on desktop Windows platforms, it is even more problematic when designing applications for Windows CE-based devices. With screen sizes that are typically a mere fraction of their desktop computer siblings, Windows CE devices are much more susceptible to problems of screen clutter and confusing user interfaces.

Fortunately, Windows CE provides extensive support for including menus and menu accelerators in your applications. Menus provide the application programmer and user interface designer with a convenient way to include a large number of user command options in a small amount of screen real estate. Accelerators allow applications to translate simple keyboard actions into menu item equivalents. Of course, Windows CE devices are not required to have a keyboard. Palm-size PCs have no use for menu accelerators. But the support for accelerators is provided by Windows CE for use by those devices that do include a keyboard.

Central to the discussion of Windows CE menus is the command bar control. In this chapter we will introduce this control, which is used for holding menus as well as other child controls. Command bars give Windows CE application programmers a way to include menus and other controls in an application without using large amounts of screen space.

I Repeat Myself When under Stress . . .

It cannot be stressed enough here that menus under Windows CE are fundamentally different from menus under Win32. Menus in Windows CE applications can still be defined in terms of menu resources just as they can on desktop Windows platforms. However, there is no concept of a menu bar, and Windows CE menus are not part of the non-client area of a window. Menus under Windows CE must be embedded within a control, most commonly a command bar. (Menus can also be inserted into command bands, a control discussed in the next chapter.) The controls that contain Windows CE menus are child controls, and therefore reside in their parent window's client area.

Another implication of the absence of menu bar support under Windows CE is that the *hMenu* parameter of *CreateWindow* and *CreateWindowEx* has no meaning for a top-level window. Therefore code like the following will fail:

```
HWND hwndMain;
HMENU hMenu;
/* Assume that IDR_MENU identifies a legitimate menu
   resource contained by the module hInstance.
*/
hMenu = LoadMenu(hInstance, MAKEINTRESOURCE(IDR_MENU));
hwndMain = CreateWindow(
    TEXT("MYWNDCLASS"),
    NULL,
    WS_VISIBLE|WS_OVERLAPPED,
    0,0,100,100,
    NULL,
    hMenu,
    hInstance,
    NULL);
```

CreateWindow in this case will most assuredly return NULL. Menus for top-level windows are not supported.

For the same reason, creating instances of a window class registered with a non-NULL *lpszMenuName* WNDCLASS member works, but the *lpszMenuName* attribute has no effect.

Of course you can still use non-NULL values for the *hMenu* parameter of *CreateWindow* and *CreateWindowEx* to specify the identifier of child windows. For example, as we saw in the previous chapter, whenever you create a Windows CE control, you use the *hMenu* parameter to specify that control's identifier.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

- Create a command bar control and insert it into a window**
- Add a menu to a command bar**
- Add controls like buttons and combo boxes to a command bar**
- Add adornments to a command bar**
- Add tool tips to a command bar**
- Add menu accelerators to a command bar menu**
- Add a window menu (system menu) to a window**

The Command Bar Control

On desktop Windows platforms, menus are contained in a *menu bar*. Windows CE does not support the concept of a menu bar. Menus are instead contained by a Windows CE control called the *command bar* control.

One exception to this is pop-up menus. Pop-up menus in Windows CE are implemented just as they are on desktop Windows platforms. We will see an example using pop-up menus later in this chapter.

Command bars are one of the common controls. Their implementation lives in the COMMCTRL.DLL dynamic link library. To use them, an application must therefore initialize the common control library with a call to *InitCommonControls* or *InitCommonControlsEx*. Furthermore,

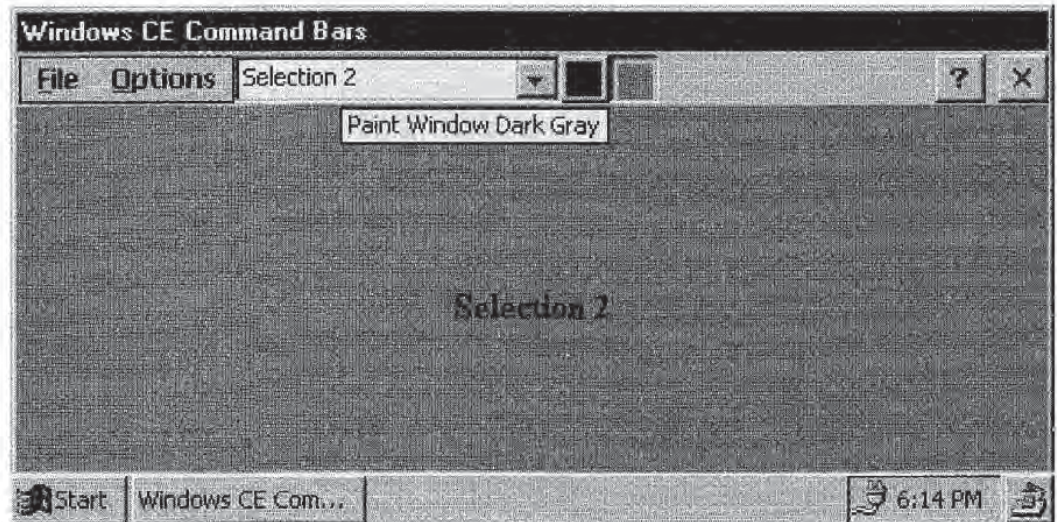


Figure 4.1 A command bar with menu, controls, and adornments.

applications must include the file `COMMCTRL.H`. This header file contains the definitions of the `InitCommonControls` and `InitCommon-ControlsEx` functions. It also defines all of the command bar API functions that we will use in this chapter.

Since command bars often contain menus, it is very easy to think of them as menu bars. But it is important to keep in mind that command bars, just like buttons or list boxes, are child controls. This means that a command bar is part of the client area of the window that owns it. Your application needs to account for the space used up by any command bars in the client area. The sidebar that follows points out a good way to do this.

Command bar controls are actually a type of toolbar control. As such, you can use any of the toolbar messages, styles, and the like with command bars. Toolbar messages can be sent directly using `SendMessage`. Toolbar styles can be added with `SetWindowLong`.

The details of creating and working with command bars will be introduced later. We first describe some menu basics. Because Windows CE menu concepts are similar to their desktop Windows counterparts, this discussion will be brief.

The sample application in this chapter, shown in Figure 4.1, wins “The World’s Most Useless Windows CE Application” award. It implements a command bar with a menu, combo box, and two buttons that change

What Happened to the Button?

Command bars are part of the client area of the windows that contain them. This point can be driven home by the following code example:

```

/* IDC_B_MAIN is the command ID of the command bar
   IDC_BUTTON is the command ID of a button
   hwndMain is the main application window
   hInstance is the application instance
*/
HWND hwndCB;      //Command bar HWND
HWND hwndButton; //Button HWND
// . . .other WinMain application code . . .
hwndCB = CommandBar_Create(hInstance, hwndMain, IDC_B_MAIN);
hwndButton = CreateWindow(TEXT("BUTTON"),
                          TEXT("Button"),
                          WS_VISIBLE|WS_CHILD|
                          BS_PUSHBUTTON,
                          0,0,65,35,
                          hwndMain,
                          HMENU(IDC_BUTTON),
                          hInstance,
                          NULL);

```

The button will be obscured by the command bar because the command bar takes up client area just like any other control.

Windows CE applications must account for the space used up by command bars. This is done using the *CommandBar_Height* function. This function returns the height of the specified command bar. This value can be used to offset any y dimension in the client area.

More generically, you can define the following macro:

```
#define MAKEKEY(y, hwndCB) (y+CommandBar_Height(hwndCB))
```

The button could then be created like this:

```

hwndButton = CreateWindow(TEXT("BUTTON"),
                          TEXT("Button"),
                          WS_VISIBLE|WS_CHILD|
                          BS_PUSHBUTTON,
                          0,MAKEKEY(0, hwndCB),65,35,
                          ...);

```

The button control will be positioned just below the command bar control.

the color of the main window client area. The buttons also include *tool tips*, which are little pop-up windows containing strings that describe the button functionality. At least this application is useful in introducing command bar concepts. The application is in the directory \Samples\cmdbar on the companion CD, and generates an application named CMDBAR.EXE.

Windows CE Menu Basics

Windows CE applications can include four types of menus:

- drop-down menus
- cascading menus
- scrolling menus
- pop-up menus

Drop-down menus are the standard pull-down menus as shown in Figure 4.1. Cascading menus are menus that contain items that open menus of their own. An example is shown in Figure 4.2.

Windows CE introduces scrolling menus. A scrolling menu is simply a menu that can scroll vertically. If an application creates a menu that contains more items than fit in the vertical screen space, Windows CE makes the menu scrollable. Users can then scroll the menu to expose menu items that do not fit in the screen area.

Pop-up menus are similar to their desktop Windows counterparts. They can be used to implement menus that do not live on the command bar, but that temporarily appear as the result of some user action. An example of a pop-up menu is given later in this chapter.

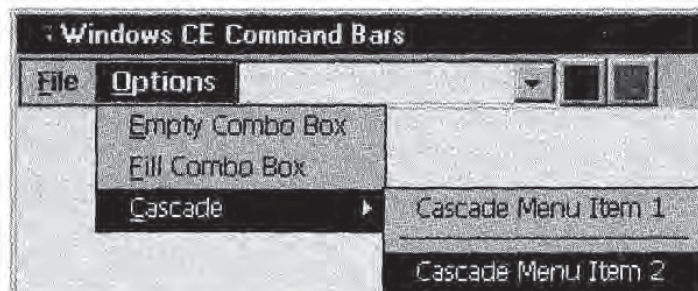


Figure 4.2 A cascading menu.

Menus can be defined in your applications by means of menu definitions in resource files. They can also be defined programmatically using the menu API. The examples in this chapter will all use the resource file approach. The complete set of Windows CE-supported menu functions is shown at the end of the chapter.

Windows CE also supports owner draw menus. We will not cover owner draw menus in this book, except to discuss the basic concepts of owner draw control programming in Chapter 9.

A Menu Resource Refresher

Just in case your memory is a bit rusty, here's a quick refresher course on how menu resources are described in resource files.

The general syntax of a menu resource is shown below. Items in uppercase are required keywords. Items in italics are supplied by the application programmer. Items in square brackets are optional values.

```
MenuName MENU [DISCARDABLE]
BEGIN
  POPUP SubMenu1Name [, options]
  BEGIN
    MENUITEM MenuItemName, id [, options]
    ... //More MENUITEMS
  END
  POPUP SubMenu2Name [, options]
  BEGIN
    MENUITEM MenuItemName, id [, options]
    ...
  END
END
```

MenuName is the name of the menu. Generally this is a resource identifier, but it can be a string menu name.

The optional DISCARDABLE keyword tells Windows CE that the menu resource can be discarded from memory automatically when no longer needed.

The POPUP keyword indicates that a new submenu is being defined. These POPUP definitions define the individual submenus. *SubMenu1Name*, *SubMenu2Name*, etc. are the names of the submenus. In the example in Figure 4.1, these would be "&File" and "&Options".

Cascading menus can be defined by nesting submenus within other submenus.

The `MENUITEM` keyword inserts a new item in the submenu. The *MenuItemName* values represent the individual pull-down menu choices.

id is the *command identifier* for the particular menu item. The use of this identifier is described in the next section, where we describe how applications respond to menu item selections.

The menu resource identifier and menu item command identifiers are typically defined in the application's `RESOURCE.H` file.

Various options can be specified for submenus and for menu items. More than one of these options can be specified by bitwise-ORing them together. For submenus, these options include:

GRAYED. The text is grayed; the menu is inactive, and does not generate `WM_COMMAND` messages.

INACTIVE. The text is displayed normally, but the menu is still inactive. No `WM_COMMAND` messages are generated.

MENUBREAK. This option is supported, but has no effect in the context of menus in command bars.

HELP. This option is supported, but has no effect in the context of menus in command bars.

Menu item options include the following:

CHECKED. Places a check mark to the left of the menu item text.

GRAYED. Same meaning as for submenus; see the list above.

INACTIVE. Same meaning as for submenus; see the list above.

MENUBREAK. The menu item and all items that follow appear in a new column in the menu.

MENUBARBREAK. Same as `MENUBREAK`, except that a vertical line separates the menu columns.

As an example, here is the menu resource definition for the cascading menu shown in Figure 4.2:

```
IDR_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
        BEGIN
            MENUITEM "&Exit", IDC_EXIT
        END
    POPUP "&Options"
```

```
BEGIN
MENUITEM "&Empty Combo Box", IDC_EMPTY
MENUITEM "&Fill Combo Box", IDC_FILL
  POPUP "&Cascade"
    BEGIN
      MENUITEM "Cascade Menu Item 1", IDC_SUBITEM1
      MENUITEM SEPARATOR
      MENUITEM "Cascade Menu Item 2", IDC_SUBITEM2
    END
  END
END
END
```

Responding to Menu Items

The usefulness of menus comes from the ability to map the various menu items in a menu to actions. This is done by assigning a command identifier to each of the menu items that an application is meant to respond to. The previous section described how a command identifier is assigned to a menu item.

This command identifier performs the same role as control identifiers for Windows CE controls. Specifically, whenever a menu item is selected, Windows CE sends a `WM_COMMAND` message to the parent of the command bar control. The low word of the `wParam` parameter contains the menu item identifier. The `WM_COMMAND` message notification code is zero, indicating that the message is sent as a result of a menu item selection.

Creating a Command Bar

Creating a command bar control is a little different from creating other Windows CE controls. Instead of using `CreateWindow` or `CreateWindowEx` as when creating child and common controls, Windows CE provides a separate API for creating and working with command bar controls.

To create a command bar control, an application calls `CommandBar_Create`:

```
CommandBar_Create(hInst, hwndParent, idCmdbar);
```

This function is a wrapper for the `CreateWindow` call that ultimately creates the actual command bar window. `hInst` is the application instance of the application in which the control is being created.

hwndParent is the command bar's parent window. *idCmdBar* is the control identifier of the command bar.

If *CommandBar_Create* is successful, it returns the HWND of the newly created command bar control. Otherwise the function returns NULL. Thus creating a command bar is semantically similar to creating any other Windows CE window or control.

Creating the command bar control is only the beginning. The command bar in Figure 4.1 contains a menu and various child controls. After calling *CommandBar_Create*, your application is left with nothing more than the HWND of an empty command bar control.

Adding useful things like menus and buttons to a command bar control requires using the various *CommandBar_Insert* functions.

Inserting a Menu into a Command Bar

An application adds a menu to a command bar with the function *CommandBar_InsertMenubar*. After all the caveats about how Windows CE does not support menu bars, the command bar API still thinks it is inserting a menu bar. In any of its incarnations, consistency in function naming has never been a strong point with Windows.

CommandBar_InsertMenubar takes four parameters:

```
CommandBar_InsertMenubar(hwndCB, hInst, idMenu, iButton);
```

hwndCB is the HWND of the command bar into which the menu is inserted. *hInst* is the application instance. *idMenu* is the resource identifier of the menu to insert.

iButton identifies where in the command bar the menu is to be inserted. A command bar can contain buttons and command bars as well as a menu. *iButton* is the zero-based index of the control to the left of which the menu is inserted. Since controls are typically inserted into command bars after the menu, *iButton* for menus is typically zero, putting the menu at the very left of the command bar.

TIP

MAKING SENSE OF *iButton*

iButton is best thought of as the position index of a control in a command bar. In the example shown in Figure 4.1, the menu has index 0, the combo box index 1, and so forth.

CommandBar_InsertMenubar internally loads the menu resource corresponding to *idMenu* from the module identified by *hInstance*.

CommandBar_InsertMenubar returns TRUE if successful. Otherwise it returns FALSE.

An alternative way to insert a menu into a command bar is with *CommandBar_InsertMenubarEx*. This function is exactly the same as *CommandBar_InsertMenubar*, except that the third parameter is not a menu resource identifier. Instead, this parameter can be passed the menu resource name, or a menu handle.

Here's an example of how you might insert a menu into a command bar. In this example, *IDR_MENU* is the resource identifier of a menu resource, defined in the file *RESOURCE.H*:

```
#define IDCB_MAIN 0 //Command bar control command ID
int WINAPI WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine,
    int nCmdShow)
{
    HWND hwndCB; //Command bar window
    HWND hwndMain; //Main application window
    /* Register main window class and create main window here
    ... */
    /* Create the command bar control */
    hwndCB = CommandBar_Create(hInstance, hwndMain, IDCB_MAIN);
    /* Insert the menu into the command bar */
    if (hwndCB)
    {
        CommandBar_InsertMenubar(hwndCB, hInstance, IDR_MENU, 0);
    }
    /* The rest of the WinMain code here
    ... */
}
```

Adding Controls to a Command Bar

The example in Figure 4.1 shows a command bar with a menu as well as a combo box and a set of buttons. Also, at the right of the command bar you can see a small Close button and Help button. How did the Windows CE child controls get there?

This section presents the command bar API functions that are used to insert controls into the command bar. It also describes how to add

Table 4.1 Command Bar Functions for Adding Controls and Adornments

FUNCTION	MEANING
<code>CommandBar_AddAdornments</code>	Used to add OK, Close, and/or Help buttons to a command bar.
<code>CommandBar_AddBitmap</code>	Adds images to a command bar to use with command bar buttons.
<code>CommandBar_AddButtons</code>	Inserts one or more buttons (not adornments) to a command bar.
<code>CommandBar_AddTooltips</code>	Inserts tool tip strings into a command bar for use with command bar buttons.
<code>CommandBar_InsertButton</code>	Inserts a single button into a command bar.
<code>CommandBar_InsertComboBox</code>	Inserts a combo box into a command bar.

adornments, the Close, Help, and OK buttons that often appear in command bars. Adding tool tips to command bar buttons is also described.

As a quick reference, Table 4.1 lists these functions and their use.

Inserting Buttons

Buttons can be inserted into a command bar with two different functions. `CommandBar_AddButtons` inserts one or more buttons into the specified command bar. `CommandBar_InsertButton` is the same except that it inserts only one button at a time.

The buttons that you insert into a command bar control send `WM_COMMAND` messages just like standard Windows CE child control buttons. The only difference is that instead of sending the message to their parent, which would be the command bar control, the message is sent to the command bar control's parent. In this way, your window procedure can handle `WM_COMMAND` messages from command bar buttons just as it responds to such messages from other buttons.

In order to add any buttons to a command bar, an application must define an appropriate button structure for each button to be added. The button structure used for this purpose is `TBBUTTON`. This is the same structure used in toolbar controls to describe toolbar buttons.

```
typedef struct _TBBUTTON
{
```

```

int iBitmap;
int idCommand;
BYTE fsState;
BYTE fsStyle;
DWORD dwData;
int iString;
} TBBUTTON, NEAR* PTBBUTTON, FAR* LPTBBUTTON;

```

iBitmap is the zero-based index of the bitmap to use with the button. *idCommand* is the button control identifier. This value is the value of the control identifier that Windows CE sends with WM_COMMAND messages whenever this particular button is pressed.

fsState defines the *button state*. This can be one or more of the values defined in Table 4.2. Some of these states refer to toolbar button styles, which are described in Table 4.3.

The *fsStyle* member of TBBUTTON specifies the various styles of a command bar button. This member can be one or more of the values in Table 4.3.

dwData is used to store application-defined data with the button. This member can be zero.

Finally, *iString* is the zero-based index of the string to use as the button's text. As command bar buttons do not use text, this member is ignored.

Let's see how to use the TBBUTTON structure to insert buttons into a command bar. The sample application for this chapter has a command bar with two buttons (see Figure 4.1). They are placed to the right of the command bar menu. One has a black square bitmap, the other a

Table 4.2 Command Bar/Toolbar Button States

STATE	MEANING
TBSTATE_CHECKED	The button is checked. Button must have the TBSTYLE_CHECK style to support this state.
TBSTATE_ENABLED	The button accepts user input, i.e., is not disabled.
TBSTATE_HIDDEN	The button is not visible.
TBSTATE_INDETERMINATE	The button is grayed out.
TBSTATE_PRESSED	The button is pressed.
TBSTATE_WRAP	A line break follows the button. The button must also have the TBSTATE_ENABLED state.

Table 4.3 Command Bar/Toolbar Button Styles

STYLE	MEANING
TBSTYLE_BUTTON	Specifies a standard push-button-style button.
TBSTYLE_CHECK	Specifies a button that looks like a push button, but that behaves like a check button. That is, it toggles between the pressed and unpressed states each time it is tapped by the user.
TBSTYLE_GROUP	Specifies a group of TBSTYLE_BUTTON buttons.
TBSTYLE_CHECKGROUP	Specifies a group of TBSTYLE_CHECK buttons. If a button in the group is pressed, it stays pressed until another button in the group is tapped. Unlike standard controls, all buttons in toolbar button group or check group must have the TBSTYLE_GROUP or TBSTYLE_CHECKGROUP style.

dark gray square bitmap. Pressing either of these buttons changes the background color of the main window's client area to the color shown in the button. The WM_COMMAND handler code for the buttons is not shown.

```

/* Define the command bar button command IDs */
#define IDC_CMDBAR_BUTTON1  1028
#define IDC_CMDBAR_BUTTON2  1029
/* Define the button bitmap image indices */
#define IDI_BUTTON1  0
#define IDI_BUTTON2  1
HWND hwndCB;  //The command bar HWND
/* Define the button structures associated with the
command bar buttons
*/
TBUTTON tb[2] = { {IDI_BUTTON1, IDC_CMDBAR_BUTTON1,
    TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0},
    {IDI_BUTTON2, IDC_CMDBAR_BUTTON2,
    TBSTATE_ENABLED, TBSTYLE_BUTTON, 0, 0}};
//Create the command bar control
hwndCB = CommandBar_Create(...);
if (hwndCB)
{
    //Other command bar code
    //...
    CommandBar_AddBitmap(hwndCB, hInstance, IDB_BUTTONS,
    2, 0, 0);
    CommandBar_AddButtons(hwndCB, 2, &tb);
    //...
}

```

The pertinent pieces of the previous sample are the definition of the `TBBUTTON` array *tb*, and the `CommandBar_AddBitmap` and `CommandBar_AddButtons` calls.

The array *tb* contains two `TBBUTTON` structures, one describing each of the buttons to be inserted into the command bar. The buttons are both enabled, push-button-style command bar buttons. The control identifiers are `ID_CMDBAR_BUTTON1` and `ID_CMDBAR_BUTTON2`, respectively.

The only part that still might need some clarification is the *iBitmap* value of the elements of the array.

Each command bar control maintains its own internal image list. This image list is what the control uses to figure out what bitmap to display on a particular command bar button. The *iBitmap* member of the `TBBUTTON` structure used to describe a particular button is the index into the image list.

An application sets the bitmap of images in this image list by calling `CommandBar_AddBitmap`:

```
CommandBar_AddBitmap(hwndCB, hInst, idBitmap, iNumImages,  
                    iReserved, iReserved);
```

hwndCB and *hInst* have the usual meanings. *idBitmap* is the resource identifier of the bitmap to add. *iNumImages* contains the number of 16-by-16-pixel button bitmap images contained in the bitmap referred to by *idBitmap*.

The last two parameters of the function are reserved and should be zero.

The bitmap referred to by the resource identifier `IDB_BUTTONS` is shown in Figure 4.3.

After the bitmap has been added to the command bar, the `CommandBar_AddButtons` call adds the buttons to the command bar:

```
CommandBar_AddButtons(hwndCB, uNumButtons, lpButtons);
```

The second parameter specifies the number of buttons to be added. *lpButtons* is a pointer to the array of `TBBUTTON` structures that define the command bar buttons.

Note that no matter how many buttons are added to a command bar control, only one bitmap resource is specified in the call to `Command-`

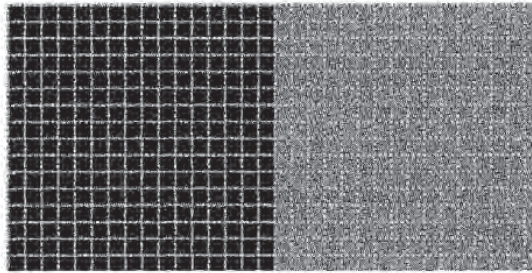


Figure 4.3 The CMDBAR application button bitmap.

Bar_AddBitmap. Each command bar button bitmap is expected to be 16 by 16 pixels, and *iNumImages* specifies the number of such images. The *iBitmap* value in a given TBBUTTON description can then reliably identify which 16-by-16 bitmap to associate with the particular button.

When a button is added with *CommandBar_AddButtons* or *CommandBar_InsertButton*, the command bar looks at the *iBitmap* member of the button's TBBUTTON definition. If this value is 2, for example, the command bar uses the second 16-by-16 section of bits from the bitmap resource added to the control by the *CommandBar_AddBitmap* call as the button image.

Inserting Combo Boxes

Combo boxes can be inserted into command bar controls using the function *CommandBar_InsertComboBox*:

```
CommandBar_InsertComboBox(hwndCB, hInst, iWidth, dwStyle,
    idComboBox, iButton);
```

hwndCB and *hInst* are the same as with the other command bar functions we've seen. *iWidth* specifies the width, in pixels, of the combo box control to be inserted. *dwStyle* defines the style of the combo box. This value can be one or more of the styles used for other combo box controls. *idComboBox* is the control identifier of the combo box, and *iButton* specifies where to put the control.

If *CommandBar_InsertComboBox* is successful, it returns the HWND of the combo box that is created. If the function fails, the return value is NULL.

Applications interact with combo boxes in command bars just as they do with other combo box controls. All messages and notifications that

are generated by a command bar combo box are sent to the command bar control's parent window. The application's main window procedure therefore handles these events in its WM_COMMAND handler.

Inserting Adornments

Adornments are the Help, OK, and Close buttons that are often found in command bars. The Help button is used as a standard way to invoke help features in an application. The Close button closes the window that contains the command bar. The OK button sends a WM_COMMAND message to the parent of the command bar. The control identifier sent with this message (i.e., the LOWORD of *wParam*) in this case is IDOK.

A command bar can include one or more adornment buttons. Any command bar that has adornments must have a Close button. You have no choice in this, and Windows CE will add it for you automatically. The Help and OK buttons can be specified optionally.

The function for doing all of this is *CommandBar_AddAdornments*:

```
CommandBar_AddAdornments (hwndCB, dwFlags, dwReserved);
```

hwndCB identifies the command bar, and *dwReserved* is reserved and must be set to zero.

That leaves the *dwFlags* parameter. This parameter is used to specify which optional adornment buttons to add to the command bar. This parameter can be CMDBAR_HELP, CMDBAR_OK, or both. CMDBAR_HELP adds the Help button, and CMDBAR_OK adds the OK button.

As an example, here's how an application would add the Help and OK buttons to a command bar with an HWND identified by *hwndMyCB*:

```
CommandBar_AddAdornments (hwndMyCB, (CMDBAR_HELP|CMDBAR_OK), 0);
```

If the function is successful, it returns TRUE. Otherwise it returns FALSE.

NOTE

COMMANDBAR_ADDADORNMENTS MUST BE LAST!

Any call to *CommandBar_AddAdornments* must come after all other functions that insert menus or controls into a particular command bar.

Table 4.4 summarizes the Windows CE messages generated when the various adornment buttons are pressed. These messages are sent to the command bar control's parent window.

Adding Tool Tips to Command Bar Buttons

Windows CE supports tool tips in command bar buttons (see Figure 4.4). A tool tip is a little pop-up window that is displayed when a user presses a command bar button and holds it down for more than half of a second. The tool tip contains an application-specified Unicode string that is used as a description of the command bar button. Tool tips are a good way to provide users of your Windows CE applications with a description of what action is performed by command bar buttons without taking up a lot of valuable screen space.

Tool tips are inserted with the function *CommandBar_AddToolTips*:

```
CommandBar_AddToolTips(hwndCB, uNumToolTips, lpToolTips);
```

uNumToolTips specifies the number of tool tip strings in *lpToolTips*. *lpToolTips* is an array of null-terminated Unicode strings. One of these strings is displayed for each command bar button.

This sounds pretty simple, but there is a catch. Windows CE does not allow tool tips for menus or combo boxes in command bars. However, it does assume that *lpToolTips* contains a string pointer for each item in the command bar. This is strange indeed. In order to add tool tips to two command bar buttons that come after a menu and combo box, *uNumToolTips* would have to be 4, and *lpToolTips* would have to contain two NULL pointers for the menu and combo box.

Table 4.4 Adornment Button Messages

BUTTON	MESSAGE GENERATED
Help	WM_HELP
OK	WM_COMMAND, with IDOK as the command identifier
Close	WM_CLOSE



Figure 4.4 A command bar with Help, OK, and Close button adornments.

To be more specific, here's how the sample application for this chapter (shown in Figure 4.1) defines the tool tips it uses:

```
TCHAR* pszTips[] = {NULL,  
    NULL,  
    TEXT("Paint Window Black"),  
    TEXT("Paint Window Dark Gray")};
```

The first two string pointers are NULL. These correspond to the menu and combo box. The next two strings are the command bar button tool tip strings. If not defined this way, *CommandBar_AddToolTips* will produce unexpected results.

With this definition for *pszTips*, the application adds the tool tips with this function call:

```
CommandBar_AddToolTips(hwndCB, 4, pszTips);
```

The basic rule of thumb for adding tool tips is that you must specify as many strings as components in your command bar (menus, combo boxes, and command bar buttons). Additionally, these strings must be specified in *lpToolTips* in the same order as the command bar components.

The *CommandBar_AddToolTips* function returns TRUE if successful. Otherwise, it returns FALSE.

Other Command Bar Functions

There are some other command bar functions we have not yet covered. These remaining functions provide functionality for such things as showing or hiding command bars, determining if a command bar is visible, and so on.

The remaining functions are pretty self-explanatory. They are listed in Table 4.5.

Table 4.5 Miscellaneous Command Bar Control Functions

FUNCTION	MEANING
CommandBar_Destroy	Destroys the specified command bar without destroying the parent window.
CommandBar_DrawMenuBar	Used to redraw or reposition the menu in the specified command bar.
CommandBar_GetMenu	Retrieves the menu handle (HMENU) of the specified command bar menu.
CommandBar_Height	Gets the height of the specified command bar.
CommandBar_IsVisible	Determines if the specified command bar is visible or not.
CommandBar_Show	Shows or hides the specified command bar.

Using Accelerators in Windows CE Applications

Desktop computers have relied heavily on keyboard accelerators for years. A *keyboard accelerator* is a keystroke combination that duplicates the behavior of a menu item or control.

Keyboard accelerators are a useful feature of many popular Windows applications. After becoming familiar with the keyboard equivalents of common menu selections, users can greatly increase the speed at which they use applications.

Windows CE provides the same keyboard accelerator support as desktop versions of Windows. Since Windows CE devices are not required to have a keyboard, accelerators don't make sense for all devices. But many devices do have keyboards, so briefly covering the subject of keyboard accelerators is worthwhile. And you never know when the Palm-size PC application that you write today will need to be ported to run on Handheld PCs.

Accelerator tables are pretty small, so the memory they consume is minimal. And compiling out the application code that enables them with preprocessor symbols is easy. As we will see, once the accelerators are defined, enabling them can be done with exactly five lines of code.

Accelerator Resources

Like menus, keyboard accelerators are a type of Windows CE resource. They are defined in a resource file as an *accelerator table*. An accelerator table has the following general syntax:

```

TableName ACCELERATORS [DISCARDABLE]
BEGIN
    [Accelerator definitions]
END

```

TableName is either the resource identifier or a string name identifying the resource. The accelerator table for this CMDBAR.EXE is defined as follows:

```

IDR_ACCELERATOR ACCELERATORS DISCARDABLE
BEGIN
    "E",      IDC_EMPTY,    VIRTKEY, CONTROL, NOINVERT
    "F",      IDC_FILL,    VIRTKEY, CONTROL, NOINVERT
    "Q",      IDC_EXIT,    VIRTKEY, CONTROL, NOINVERT
END

```

Each of the accelerator definitions identifies the keyboard key that must be pressed to invoke the accelerator.

The second item in each definition is the control or menu item identifier to which the accelerator corresponds. This is the command identifier that Windows CE sends with the WM_COMMAND message to the window that owns the accelerators.

The VIRTUAL keyword indicates that Windows CE is to use the virtual key code, not the ASCII key code, for the key specified in the accelerator definition.

CONTROL indicates that the Control key must also be pressed to invoke the accelerator. So the first definition means that the key combination Ctrl+E must be pressed. Other keywords of this type are SHIFT and ALT, indicating that the Shift or Alt key must be pressed. For example, to define an accelerator for the key combination Alt+Shift+X, the accelerator table would include this line:

```

"X", SomeID, VIRTKEY, SHIFT, ALT, NOINVERT

```

The NOINVERT keyword says that the menu containing the menu item corresponding to the accelerator's control identifier is not inverted (i.e., not highlighted) when the accelerator key combination is pressed. Leaving out this keyword forces Windows CE to try to invert the menu.

The identifiers such as `IDR_ACCELERATOR` and `IDC_EMPTY` are typically defined in the application's `RESOURCE.H` file.

Loading and Translating Accelerators

To use keyboard accelerators, an application must load the accelerator table resource. Windows CE represents keyboard accelerators using an *accelerator handle* of type `HACCEL`.

Accelerators are loaded using the *LoadAccelerators* function:

```
LoadAccelerators(hInstance, lpTableName);
```

hInstance specifies the application instance or dynamic link library instance that contains the accelerator table resource. *lpTableName* is the name of the accelerator table resource. If the accelerator table was given a string name when it was defined in the resource file, this is the string that you pass to *lpTableName*.

If, on the other hand, the table is identified by a resource identifier, you can use the Windows CE macro `MAKEINTRESOURCE` to convert the identifier into the suitable string value. For example, `CMDBAR.EXE` loads its accelerators as follows:

```
#define IDR_ACCELERATOR 102
HACCEL hAccel;
hAccel = LoadAccelerators(hInstance,
    MAKEINTRESOURCE(IDR_ACCELERATOR));
```

If *LoadAccelerators* is able to load the accelerator table, it returns a handle to the table. Otherwise the function returns `NULL`.

Once an accelerator table is loaded, an application needs to know how to respond to accelerator keystrokes. This is done by the function *TranslateAccelerator*:

```
TranslateAccelerator(hWnd, hAccelTable, lpMsg);
```

The parameters passed to *TranslateAccelerator* are an `HWND`, an accelerator table (`HACCEL`), and a pointer to a message structure. The function first determines if the message specified by *lpMsg* is a `WM_KEYDOWN` or `WM_SYSKEYDOWN` message. If it is, it looks in the accelerator table specified by *hAccelTable* to see if the virtual key code sent with the message corresponds to any of the accelerator keys. If so, the message is converted into a `WM_COMMAND` message and sent to the window procedure of the window specified by the *hWnd*

parameter, and then returns TRUE. Otherwise *TranslateAccelerator* returns FALSE.

So how does an application use this function to continually monitor the keyboard for accelerator keystrokes?

Handling accelerators is generally done by modifying an application's message loop. Consider what happens when the standard message loop code is changed by first checking for accelerators:

```
while (GetMessage(&msg, NULL, 0, 0) == TRUE)
{
    if (!TranslateAccelerator(hwndMain, hAccel, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

As described above, *TranslateAccelerator* turns any WM_KEYDOWN or WM_SYSKEYDOWN message that corresponds to an accelerator into the equivalent WM_COMMAND message and sends it off to the appropriate window procedure. For every message that gets into the application's message queue, this new message loop code first gives *TranslateAccelerator* a chance to process the message. If the message does not correspond to an accelerator keystroke (i.e., if *TranslateAccelerator* returns FALSE,) the message is processed in the usual way by *TranslateMessage* and *DispatchMessage*.

NOTE

COMPATIBILITY: WM_SYSCOMMAND MESSAGES

Unlike on desktop versions of Windows, under Windows CE *TranslateAccelerator* does not generate WM_SYSCOMMAND messages, only WM_COMMAND messages.

Using the Window Menu

The window menu (or system menu, as it used to be called) is the little menu that appears in some windows when you tap the window icon in the upper left corner of the title bar. A window with a window menu is shown in Figure 4.5.

You include a window menu in a window by specifying the WS_SYSMENU style when the window is created:

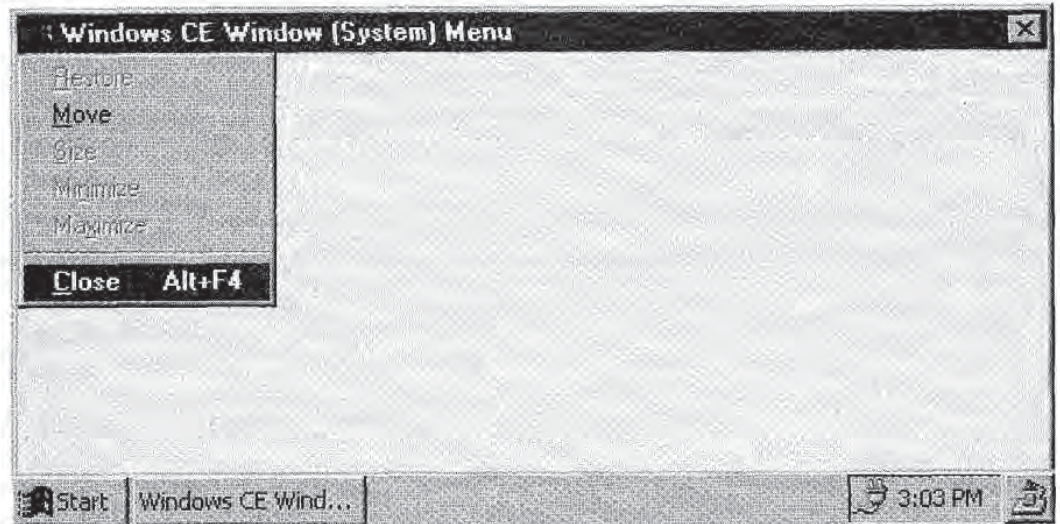


Figure 4.5 A Windows CE window (system) menu.

```

HWND hwndSysMenu; //Handle of window with a window menu
hwndSysMenu = CreateWindow(
    TEXT("MyWndClass"),
    TEXT("My Window"),
    WS_VISIBLE|WS_OVERLAPPED|WS_SYSMENU,
    ...);

```

The window menu notifies its parent window that an item has been selected from the window menu by sending WM_SYSCOMMAND messages to the window. This is analogous to the command bar menu behavior of sending WM_COMMAND messages when items are selected. Table 4.6 details the WM_SYSCOMMAND message parameters.

Table 4.6 WM_SYSCOMMAND Message Parameters

PARAMETER	MEANING
wParam	Specifies the system command. Value can be SC_CLOSE or SC_KEYMENU.
LOWORD(lParam)	Specifies the x component of the point where the stylus tapped the screen if the menu item was selected with the stylus.
HIWORD(lParam)	Specifies the y component of the point where the stylus tapped the screen if the menu item was selected with the stylus.

The `SC_CLOSE` system command indicates that the Close item was selected from the window menu. `SC_KEYMENU` means that the menu has been activated by a keystroke.

Notice in Figure 4.5 that including a system menu in a window also includes a Close button in the upper right corner of the title bar.

A window procedure should return zero for any `WM_SYSCOMMAND` message that is handled. All other `WM_SYSCOMMAND` messages should be passed on to *DefWindowProc*.

NOTE

SYSTEM COMMANDS

Under Windows NT and Windows 98, there are many more possible system command values that can be sent with `WM_SYSCOMMAND` messages. Under Windows CE, only the `SC_CLOSE` and `SC_KEYMENU` values are supported.

The Complete Windows CE Menu API

So far we have been describing menus that are created in Windows CE resource files. The typical use of menus has been to create the desired menu in a menu resource and then insert it into a command bar with *CommandBar_InsertMenubar*. The application then responds to menu item selections with the appropriate `WM_COMMAND` message handler.

Windows CE also provides a rich set of functions for working with menus more directly. This API is very similar to the traditional Win32 menu API. Since there are already numerous resources that describe these functions in detail, this section will simply summarize the Windows CE menu API and point out where particular functions differ from their Win32 siblings. We also demonstrate some of the functions by showing how to add a context-specific pop-up menu to the `CMDBAR.EXE` sample application.

Adding Pop-up Menus

Menus in Windows CE applications do not have to reside on a command bar menu. Applications often implement *pop-up menus* that are temporarily displayed when the user performs some specified action,

like tapping the screen while pressing the Alt key on the keyboard. Some devices such as Palm-size PCs have hardware navigation buttons that can be used in various combinations to invoke pop-up menus. For the example in this section, it is assumed that a keyboard is present.

Pop-up menus are extremely useful in Windows CE applications to present users with lists of options that depend on the context in which the menu is invoked. For example, in a word processing application a pop-up menu might contain one set of choices when a document is open in the application, a completely different set of menu items when the application has no documents open.

Pop-up menus behave pretty much like regular command bar menus. Once displayed, the user can select menu items as in any other menu. Pop-up menus notify their parent that an item has been selected via the WM_COMMAND message. When displayed, a pop-up menu generally stays open until the user makes a menu item selection or taps outside the menu.

In this section we add context-specific pop-up menus to CMDBAR.EXE. One of these menus is shown in Figure 4.6. The menus are invoked by tapping the client area of the main application window while pressing the Alt key. If the client area is black, the menu offers the choice of painting the client area gray, or reverting to white. If the

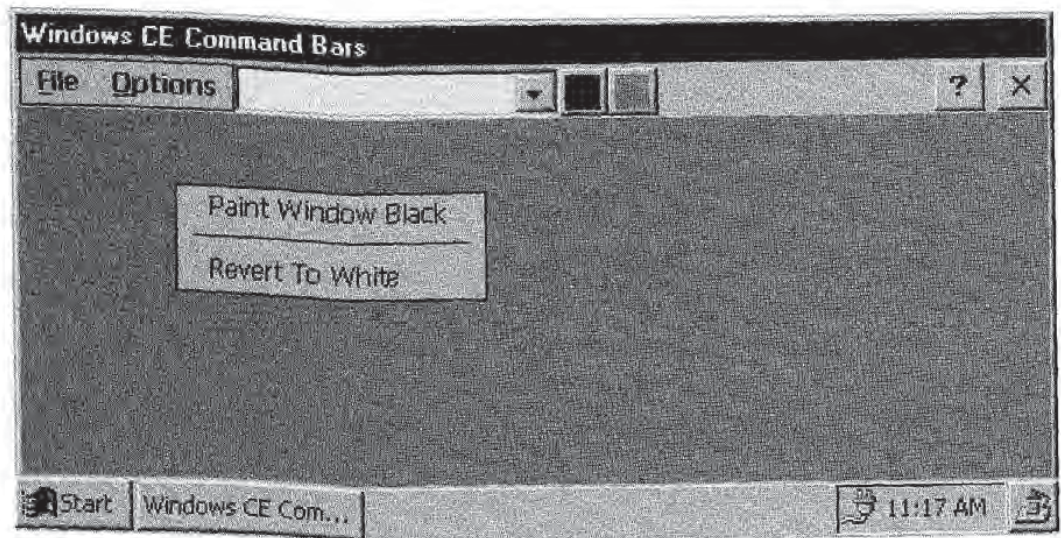


Figure 4.6 A Windows CE Pop-up menu.

client area is gray, the menu allows the user to paint it black or revert to white. If the client area is white, the pop-up menu offers the choice of painting black or gray, and the revert-to-white option is disabled.

We define the basic pop-up menus in this menu resource:

```
IDR_POPUPS MENU DISCARDABLE
BEGIN
  POPUP "Popup1"
  BEGIN
    MENUITEM "Paint Window Gray", IDC_SET_COLOR_GRAY
    MENUITEM SEPARATOR
    MENUITEM "Revert To White", IDC_CLEAR
  END
  POPUP "Popup2"
  BEGIN
    MENUITEM "Paint Window Black", IDC_SET_COLOR_BLACK
    MENUITEM SEPARATOR
    MENUITEM "Revert To White", IDC_CLEAR
  END
END
```

Each of the individual pop-ups in this menu definition will be used as the pop-up menu that is displayed for a particular application context, i.e., current client area color. Each pop-up statement in the resource definition above defines a *submenu*.

NOTE

MENUS CAN BE CREATED PROGRAMMATICALLY

The examples in this chapter all use menu resources to define the menus that they use. The Windows CE menu API also provides functions that allow applications to create menus programmatically. See Table 4.9 for a complete list of menu functions.

Programmatically, the process of creating and displaying a pop-up menu and then detecting a user's menu item selection can be summarized with these three steps:

- Loading the menu resource that contains the pop-up menu to be displayed
- Obtaining a menu handle to the appropriate submenu
- Tracking the user's menu item selection

The menu resource is loaded using the *LoadMenu* function:

```
LoadMenu(hInstance, lpMenuName);
```

LoadMenu returns a menu handle (HMENU) to the specified menu. If it fails, *LoadMenu* returns NULL.

hInstance is the HINSTANCE of the application or DLL that contains the specified menu resource. *lpMenuName* is the name of the menu resource to be loaded. As with all of the resource loading functions, this parameter can be obtained by passing the menu resource identifier to the macro MAKEINTRESOURCE. Refer to the section "Loading and Translating Accelerators" for details.

The next step in the process, obtaining a submenu handle, is done by calling *GetSubMenu*:

```
GetSubMenu(hMenu, nPos);
```

Like *LoadMenu*, *GetSubMenu* returns a menu handle to the specified submenu if it succeeds. Failure results in a NULL return value.

hMenu specifies the menu containing the submenu of interest. This value generally comes from a previous *LoadMenu* call. *nPos* is the zero-based index of the submenu to be extracted from *hMenu*.

For example, to get a menu handle to the Popup2 submenu defined in the resource definition above, an application would do this:

```
#define IDR_MYMENU 1028 //Resource ID of the menu,
                       //typically defined in
                       //resource.h
HMENU hMenu, hSubMenu; //Define the menu handles
hMenu = LoadMenu(hAppInstance, MAKEINTRESOURCE(IDR_MYMENU));
if (hMenu)
{
    hSubMenu = GetSubMenu(hMenu, 1);
}
```

After the application has a handle to the submenu it wants to use as a pop-up, all that is left to do is display that submenu and track user selections. Menu tracking is the menu behavior that includes displaying and hiding the menu and highlighting menu items that are pressed. Menu tracking is implemented by the operating system. All that an application needs to do to display a pop-up menu and track selections is to call *TrackPopupMenu*:

```
TrackPopupMenu(hMenu, uFlags, x, y, nReserved, hWnd, prcRect);
```

hMenu is the menu handle of the submenu to track. *x* and *y* determine where the menu is displayed, specifying the *x* and *y* coordinates of the top left corner of the menu. These coordinates are assumed to be given

in screen coordinates, not in client coordinates. *nReserved* must be set to zero.

hWnd specifies the HWND of the menu's parent window. *prcRect* points to a RECT that specifies the area of the screen which the user can tap without closing the pop-up menu. If this parameter is NULL, the pop-up menu is always closed if the user taps anywhere outside the open menu.

uFlags is a UINT that specifies various flags controlling the position and behavior of the pop-up menu. *uFlags* can be one or more of the values in Table 4.7. The *uFlags* parameter of *TrackPopupMenu* can include only one of the values TPM_CENTERALIGN, TPM_LEFTALIGN, and TPM_RIGHTALIGN. These values are used to specify the horizontal alignment of the pop-up menu. Likewise, only one of the values TPM_BOTTOMALIGN, TPM_TOPALIGN, and TPM_VCENTERALIGN may be specified for vertical alignment.

TrackPopupMenu does not return until a menu item is selected or the menu is closed by tapping a point on the screen not contained by the RECT in *prcRect*. If the TPM_RETURNCMD flag is set, the return value is the command identifier of the selected menu item. If this style is not set, the return value of *TrackPopupMenu* is treated like a BOOL:

Table 4.7 TrackPopupMenu Flags

FLAG	MEANING
TPM_CENTERALIGN	Centers the menu horizontally with respect to the x parameter.
TPM_LEFTALIGN	Positions the menu so that the left side is aligned with the x parameter of TrackPopupMenu.
TPM_RIGHTALIGN	Positions the menu so that the right side is aligned with the x parameter of TrackPopupMenu.
TPM_BOTTOMALIGN	Positions the menu so that the bottom edge is aligned with the y parameter of TrackPopupMenu.
TPM_TOPALIGN	Positions the menu so that the top edge is aligned with the y parameter of TrackPopupMenu.
TPM_VCENTERALIGN	Centers the menu vertically with respect to the y parameter of TrackPopupMenu.
TPM_RETURNCMD	If this style is set, TrackPopupMenu returns the identifier of the selected menu item.

That is, it returns TRUE if the function completes successfully, and FALSE otherwise.

NOTE

TRACKPOPUPMENU FLAGS

Windows CE does not support the TPM_NONOTIFY flag. Also, as Windows CE devices do not support a mouse, the TPM_LEFTBUTTON and TPM_RIGHTBUTTON flags are not supported.

The pop-up menus in CMDBAR.EXE are displayed by holding the Alt key and tapping the screen. All of the pop-up menu code is therefore implemented in the WM_LBUTTONDOWN message handler in the main window procedure. In the code below, *bWhite* and *bBlack* are BOOL global variables that indicate if the window is painted white or black, respectively.

```
case WM_LBUTTONDOWN:
    POINT pt;
    SHORT nState;
    int nSubMenuIndex;
    HMENU hPopupMenu, hSubMenu;
    pt.x = LOWORD(lParam);
    pt.y = HIWORD(lParam);
    ClientToScreen(hwnd, &pt);
    nState = GetKeyState(VK_MENU);
    if (nState & 0x80)
    {
        hPopupMenu = LoadMenu(ghInst,
            MAKEINTRESOURCE(IDR_POPUPS));
        if (hPopupMenu)
        {
            nSubMenuIndex = (bBlack ? 0 : 1);
            hSubMenu = GetSubMenu(hPopupMenu, nSubMenuIndex);
            /* Insert the menu item for painting the window
             gray. Also disable the revert to white
             menu item if the client area is already
             painted white.
             */
            if (bWhite)
            {
                InsertMenu(hSubMenu, 1, MF_BYPOSITION,
                    IDC_SET_COLOR_GRAY, TEXT("Paint Window Gray"));
                EnableMenuItem(hSubMenu, IDC_CLEAR,
                    MF_BYCOMMAND | MF_GRAYED);
            }
            TrackPopupMenu(hSubMenu, TPM_TOPALIGN | TPM_LEFTALIGN,
                pt.x, pt.y, 0, hwnd, NULL);
        }
    }
}
```

```
    }          //End of if (hMenuPopup) block
  }          //End of if (nState & 0x8000) block
  return (0);
```

The `WM_LBUTTONDOWN` message is sent with the client coordinates of the point where the screen was tapped in the window that receives the message. These coordinates are immediately converted to screen coordinates with a call to *ClientToScreen*. This is because the *TrackPopupMenu* call that comes later expects its *x* and *y* parameters in screen coordinates.

The next interesting part of this piece of code tells us if the Alt key is being pressed. The call to *GetKeyState* does this for us. This function takes a virtual key code as its only parameter. If the corresponding key is pressed, *GetKeyState* returns a `SHORT` whose high-order bit is 1.

If the Alt key is pressed, the code proceeds to load the pop-up menu resource and get a handle to the proper submenu, and the menu is displayed and tracked with *TrackPopupMenu*.

Two other interesting menu functions are demonstrated in the piece of code above. In the case that the window was already painted white, a menu item for painting the window gray is added to the pop-up menu with a call to *InsertMenu*. Otherwise the only choices will be for painting it black and reverting to white. Also, if the window is white, the application disables the "Revert To White" option by calling *EnableMenuItem*. The next two sections discuss how these features are implemented.

Inserting New Menu Items

New menu items can be inserted into existing menus at run-time with the function *InsertMenu*:

```
InsertMenu(hMenu, uPosition, uFlags, uIDNewItem, lpNewItem);
```

The *hMenu* parameter specifies the `HMENU` of the menu into which the new menu item is inserted. *uPosition* specifies the menu item which the new menu item is to be inserted before. This value is interpreted depending on the *uFlags* parameter.

uFlags must be either `MF_BYCOMMAND` or `MF_BYPOSITION`, combined with at least one of the values in Table 4.8. `MF_BYCOMMAND` means that *uPosition* gives the identifier of the menu item to be

Table 4.8 InsertMenu Flags

FLAG	MEANING
MF_CHECKED	Draws a check mark to the left of the menu item text.
MF_ENABLED	Enabled the menu item. Item can be selected and the item text is not grayed.
MF_GRAYED	Disables the menu item and grays the item text.
MF_MENUBREAK	Places the item in a new column.
MF_MENUBARBREAK	Same as MF_MENUBREAK, except columns are separated by a vertical line.
MF_OWNERDRAW	Specifies the menu item as owner draw.
MF_POPUP	Indicates that the menu item is a submenu.
MF_SEPARATOR	The item inserted is a horizontal menu item separator.
MF_STRING	Indicates that the <i>lpNewItem</i> parameter is a string.
MF_UNCHECKED	Opposite of MF_CHECKED, i.e., a check mark is not drawn next to the item text. This flag is set by default.

inserted. MF_BYPOSITION says that *uPosition* is the zero-based index of the new item. MF_BYCOMMAND is the default.

uIDNewItem indicates the command identifier of the new menu item. If *uFlags* includes the MF_POPUP flag, *uIDNewItem* is the menu handle of the menu or submenu to be inserted.

lpNewItem specifies the contents of the new menu item. Generally *lpNewItem* points to a null-terminated Unicode string used as the menu item text. This parameter can also contain information for drawing owner draw menu items. But as we are not covering owner draw menus in this book, we don't discuss this.

NOTE

INSERTMENUITEM

The function *InsertMenuItem* is not supported under Windows CE.

Enabling and Disabling Menu Items

We have described the pop-up menus that were added to the CMD-BAR application as context-specific. That means that the particular

pop-up menu that is displayed depends on the state of the application at the time the menu is displayed.

Individual menu items can also be displayed differently depending on the state of an application. For example, a word processor typically grays out the Cut and Copy menu items in the Edit menu if no text is selected in a document. But when text is selected, those menu items become enabled.

The CMDBAR.EXE application pop-up menus have a Revert To White menu item that is only enabled when the main window background is not already painted white. Menu items are enabled or disabled with the *EnableMenuItem* function:

```
EnableMenuItem(hMenu, uIDEnableItem, uEnable);
```

hMenu is the menu handle of the menu or submenu that contains the item to disable or enable.

uEnable is similar to the *uFlags* parameter of the *InsertMenu* function. It is a combination of one of the flags MF_COMMAND or MF_BYPOSITION, and one of the flags MF_GRAYED or MF_ENABLED. These flags have the same meanings as in the *InsertMenu* function.

uIDEnableItem indicates which menu item to enable or disable. As with the *uFlags* parameter of *InsertMenu*, *uIDEnableItem* specifies the command identifier of the menu item if *uEnable* includes the MF_COMMAND flag. If *uEnable* instead contains MF_BYPOSITION, *uIDEnableItem* is the zero-based index of the menu item to enable or disable.

NOTE

MF_DISABLED NOT SUPPORTED

Under Windows CE, the menu flag MF_DISABLED is not supported. To disable menu items using functions like *InsertMenu* and *EnableMenuItem*, applications must use the MF_GRAYED flag.

The Complete Windows CE Menu API

The Windows CE menu API includes many more functions than those few detailed above. However, their usage is generally similar to those functions which we have discussed in detail.

Table 4.9 The Windows CE Menu Functions

FUNCTION	MEANING
AppendMenu	Inserts a new menu item at the end of the specified menu.
CheckMenuItem	Used to add or remove a check mark from a menu item.
CheckMenuRadioItem	Draws a bullet next to the specified menu item and removes any previously drawn bullets from all other items in the menu item group.
CreateMenu	Creates an empty menu.
CreatePopupMenu	Creates an empty pop-up menu.
DeleteMenu	Deletes an item from the specified menu.
DestroyMenu	Destroys the specified menu and frees any memory used by the menu resource. The menu analogue of DestroyWindow.
DrawMenuBar	Redraws the menu in the specified window. The window is a command bar window.
EnableMenuItem	Enables or disables the specified menu item.
GetMenuItemInfo	Gets information about the specified menu item in the form of a MENUITEMINFO structure.
GetSubMenu	Gets a handle to the specified submenu.
GetSystemMenu	Gets a handle to the window menu (system menu) in the specified window.
InsertMenu	Inserts a new menu item into the specified menu.
LoadMenu	Loads the specified menu resource.
RemoveMenu	Deletes a menu item from the specified menu.
SetMenuItemInfo	Changes menu item information.
TrackPopupMenu	Displays a pop-up menu and tracks user selections.
TrackPopupMenuEx	Similar to TrackPopupMenu, but passes the exclusion RECT in a TPMPARAMS structure instead of as an individual LPRECT.

Table 4.9 can be used as a quick reference for the menu operations provided by the operating system. Now that you have a good understanding of Windows CE menu basics, understanding how to use these functions when needed should be straightforward with the help of the Windows CE on-line documentation.

The Complete CMDBAR Sample Application

All of the concepts presented in this chapter are pulled together in the sample application, CMDBAR.EXE. Complete source code for this application is included on the companion CD under the directory \Samples\cmdbar. The command bar button bitmap file and all of the project files needed to build the application are included there as well.

Concluding Remarks

In this chapter, we discussed how to add menus and accelerators to Windows CE applications. We introduced the Windows CE command bar control, and showed how menus, controls, and tool tips can be added to command bars. This chapter also presented the Windows CE menu API.

At this point, you should be able to write some fairly complex applications that include menus, modal and modeless dialogs, and the standard Windows CE child controls. In the next chapter, we will explore programming the Windows CE common control library in greater detail. You will then be able to add even more rich features, such as calendar functionality, to your applications very easily.

Windows CE Common Controls

This chapter discusses programming Windows CE common controls. It concentrates on the *month calendar* control, the *date time picker* control, *rebar* controls, and *command bands* (Table 5.1). But the basic common control programming concepts covered here, such as how to respond to common control notifications, can be applied to programming all Windows CE common controls.

Like the other common controls, each of the controls listed in Table 5.1 resides in COMMCTRL.DLL. To use one or more of them in an application, the COMMCTRL.DLL must be loaded. The application then creates the controls using *CreateWindow* or *CreateWindowEx* calls with the appropriate control class name in the *lpClassName* parameter. Care must be taken to load COMMCTRL.DLL properly. See the section called “Why Are My HWNDs Always NULL?” in this chapter for details.

For any Windows CE common control, there are a number of messages that an application can send to the control to take advantage of various features and control functionality. In addition, there are many notifications that a common control can send to its parent window via the WM_NOTIFY message.

Table 5.1 Windows CE Common Controls Covered in This Chapter

CONTROL	USE
Month Calendar	A complete month view calendar control. User interface allows for easy selection of one or more dates.
Date Time Picker	Displays dates and times, and provides a convenient user interface for changing the date and time information displayed.
Rebars	Resizable child control container.
Command Bands	A special rebar containing close, help, and OK buttons.

The programming model of all common controls is basically the same. Applications create controls with various control styles to enable various control features. Then parent windows send the controls messages to program their behavior. Controls also send notifications to their parent window to alert the parent that some action has been performed or some other occurrence of interest has taken place. It is therefore more economical to present a sample application for each control that highlights some of the more interesting features of the particular control.

After understanding the sample application, you can delve into other messages, notifications, or styles that might be of interest to you for your specific application programming needs. Using the samples as a model, you should find that taking advantage of the other Windows CE common controls not covered in this chapter will not present any serious challenges.

At the end of each section covering a control, a brief description of all messages and notifications for the particular control is given.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

Program month calendar controls

Program date time picker controls

Program rebar controls

Program command band controls

Why Are My HWNDs Always NULL?

There is a serious discrepancy between the on-line documentation and the reality of creating any of the Windows CE common controls covered in this chapter.

The documentation states that applications can create these controls by loading **COMMCTRL.DLL** with *InitCommonControls*, and then calling *CreateWindow* or *CreateWindowEx* with the appropriate control window class name. Alternatively, the documentation states, an application can load just the control classes it needs with *InitCommonControlsEx*, and then proceed with *CreateWindow* or *CreateWindowEx*.

It turns out that you *must* use the latter method with either *CreateWindow* or *CreateWindowEx*.

For example, I tried the following:

```
#include <commctrl.h>
HWND hwndMonth;
InitCommonControls();
//Code to create main application window, etc, removed
hwndMonth = CreateWindowEx(0, MONTHCAL_CLASS,...);
```

To my surprise, *hwndMonth* was NULL after the *CreateWindowEx* call executed. To try and figure out what was going on, I put a call to *GetLastError* right after creating the control and got back error code 1407, which stands for **ERROR_CANNOT_FIND_WND_CLASS**.

This can only mean that *InitCommonControls* in fact does not register the window class for the month calendar control. This error also occurred for other common control classes covered in this chapter.

When doing the following, however, everything worked as expected:

```
#include <commctrl.h>
HWND hwndMonth;
INITCOMMONCONTROLSEX icex;
icex.dwSize = sizeof(icex);
icex.dwICC = ICC_DATE_CLASSES;
InitCommonControlsEx(&icex);
hwndMonth = CreateWindowEx(0, MONTHCAL_CLASS,...);
```

The Month Calendar Control

The month calendar control provides a quick way to include full-featured calendar functionality in your applications. It can display dates over any specified range of dates, and automatically accounts for the

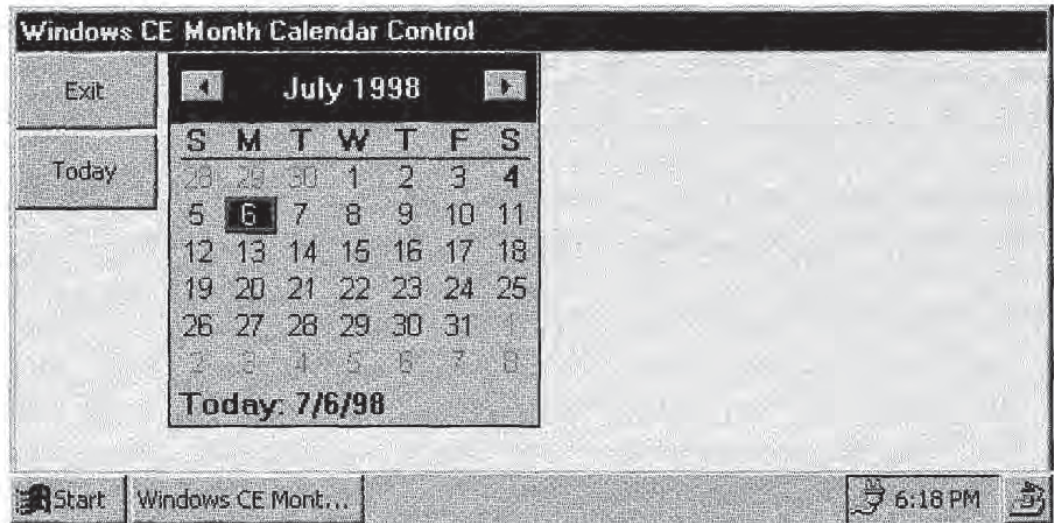


Figure 5.1 The month calendar control.

day of week variations for dates in different years, as well as for leap years. An example of the month calendar control is shown in Figure 5.1.

The control allows users to move backward or forward through the months of the year by clicking the left or right arrow button in the control title. If the device on which the application is running has a keyboard, users can also move through the months using the Page Up (to advance) and Page Down (to go back) keys.

As an alternative, if the user taps the name of the month in the control title, a pop-up menu appears listing all of the months in the year. Selecting a month from this menu tells the control to display the selected month.

The current year can also be changed. Tapping on the year in the title of the control forces an up-down control to appear that can be used to change the year. On devices with a keyboard, CTRL+Page Up and CTRL+Page Down also move the year forward or back.

A number of features of the month calendar control are programmable, such as whether the control indicates the current date, and various control color options.

Writing personal information management applications such as an appointment book or meeting scheduler is made much easier given the functionality of the month calendar control.

The control class for the month calendar control, which you need to pass to *CreateWindow* or *CreateWindowEx* when creating an instance of this control, is `MONTHCAL_CLASS`.

Month Calendar Control Styles

There are five control styles that can be used with the month calendar control:

MCS_DAYSTATE. Indicates that the control is capable of drawing specific dates in bold text.

MCS_MULTISELECT. Indicates that the control can select a range of dates. The default is that month calendar controls can only select one date at a time.

MCS_NOTODAY. Control does not display "Today is . . ." text at the bottom.

MCS_NOTODAYCIRCLE. Control does not box the current date.

MCS_WEEKNUMBERS. Control displays the number of each week (1–52) to the left of each week.

Day States

Month calendar controls can be made to display dates of interest in bold text. For example, you may want an appointment calendar application to display holidays in bold to make them easier to identify.

Highlighting dates in bold in a month calendar control is done using the *day state* mechanism. A day state is a data type called `MONTHDAYSTATE` which is simply a `DWORD`. Each of the 32 bits of a `MONTHDAYSTATE` is used to represent the state of the corresponding day in a particular month. If, for example, bit 5 is set to 1, day 5 in the corresponding month is displayed in bold on the month calendar control. Zero values in `MONTHDAYSTATEs` mean the corresponding dates are not bold.

Applications typically tell month calendar controls which dates to display in bold in response to the `MCN_GETDAYSTATE` notification. We will look at the specifics of responding to this and other month calendar control notifications a little later.

An Example

In this example, we create a month calendar control that highlights a limited set of holidays. For the sake of simplicity, the holidays it highlights are among those that always fall on the same date every year. This prevents me from having to implement an algorithm that can do things such as determine what date the third Sunday in June is in any given year (sorry, Dad).

Our application uses a month calendar control that only allows users to select a single date at a time. It displays the currently selected date in the main application window title bar. Also, the application has a button labeled Today which sets the current selection to today's date.

To pick the holidays, I sat down with my wife's "Cat Lover" calendar and chose seven holidays at random (I wouldn't have picked January 22, "Answer Your Cat's Question Day," any other way). Here's what I came up with:

- January 1: New Year's Day
- January 22: see above
- February 14: Valentine's Day
- March 17: St. Patrick's Day
- July 4: Independence Day
- October 31: Halloween
- December 25: Christmas

Our month calendar control will display each of these holidays in bold. Therefore, our application needs to use day states. Let's look first at how to do this.

Recall that for any given month, a month calendar control uses a `MONTHDAYSTATE` 32-bit integer to represent the dates to display in bold. The least significant bit represents the first of the month, the next bit represents the second, and so on. Since the control keeps track of the number of days in a given month for a particular year, the last day of the month may be bit 28, 29, 30, or 31. Defining a day state for Christmas, for example, could be done like this:

```
MONTHDAYSTATE mdsXMas;  
mdsXMas = (MONTHDAYSTATE)(0x01 << 24);
```

Since bit zero of a `MONTHDAYSTATE` represents the first of the month, bit 24 corresponds to the 25th. Shifting the number `0x01` 24 bits to the left sets that bit.

The application defines an array of `MONTHDAYSTATE` values to represent all of the holidays I picked. Notice that the first entry of this array, representing the month of January, has two holidays:

```
#define ONE 0x01
MONTHDAYSTATE mdsHoliday[12] = {(ONE | (ONE<<21)), //January
    (ONE<<13), //February
    (ONE<<16), //March
    0, //April
    0, //May
    0, //June
    (ONE<<3), //July
    0, //August
    0, //September
    (ONE<<30), //October
    0, //November
    (ONE<<24) //December};
```

The next interesting thing that the application does is to create the month calendar control and set some of its visual properties. `hwndMain` and `hInstance` are the main application window and the application `HINSTANCE`, respectively.

```
#define IDC_MONTH 1026
HWND hwndMonth;
hwndMonth = CreateWindowEx(
    MONTHCAL_CLASS, NULL,
    WS_VISIBLE|WS_BORDER|WS_CHILD|MCS_DAYSTATE,
    0,0,0,0,
    hwndMain,
    (HMENU) IDC_MONTH, hInstance, NULL);
OnInitMonthCalendar(hwndMonth, 70, 0);
```

The `MCS_DAYSTATE` creates a month calendar control that can use day states, which we need in order to highlight our holidays.

The only funny thing here is that all of the window position parameters passed to `CreateWindowEx` are zero. The reason for this is that a month calendar control can be made to display more than one month at a time. The control therefore leaves it to the application to set the size of the control to accommodate the number of months to display.

This is made easier by the `MCM_GETMINREQRECT` message. This message is sent to a month calendar control to determine the mini-

imum height and width required to display one calendar month. When an application sends this message to a month calendar control, the control returns a `RECT` through the `lParam` parameter of `SendMessage`. The *right* and *bottom* members of this `RECT` contain the minimum width and height required, respectively.

Let's look at the `OnInitMonthCalendar` function to see how to use the `MCM_GETMINREQRECT` message:

```
void OnInitMonthCalendar(HWND hwndCal, int nLeft, int nTop)
{
    RECT r;
    SendMessage(hwndCal, MCM_GETMINREQRECT,
        0, (LPARAM) (LPRECT)&r);
    /* Resize the month calendar control window
       to accommodate one full calendar month.
    */
    SetWindowPos(hwndCal, NULL,
        nLeft, nTop,
        r.right, /cx, new width
        r.bottom, /cy, new height
        SWP_NOZORDER);
}
```

If an application wanted to display more than one month at a time, it could pass integer multiples of *r.right* and *r.left* as the *cx* and *cy* parameters of `SetWindowPos`.

The next interesting part of the application is in the main window's window procedure, where we handle the control notifications we are interested in (Figure 5.2). In our example, we respond to the `MCN_SELCHANGE` and `MCN_GETDAYSTATE` notifications. The pertinent part of the window procedure is shown in Figure 5.2.

The `MCN_SELCHANGE` notification is sent by the month calendar control to its parent window whenever the date selection in the month calendar control is changed by some user interaction with the control. Furthermore, as the name implies, this notification is only sent when the selected date or dates change. For example, tapping a date in the control and then tapping the same date again results in an `MCN_SELCHANGE` notification only for the first tap.

There is a related notification called `MCN_SELECT`. This notification is sent by the month calendar control only when the user explicitly taps a date or selects a date range. It is not sent any other time, for example

```

case WM_NOTIFY:
    LPNMHDR lpmhdr;
    nID = (UINT)wParam;
    switch(nID)
    {
    case IDC_MONTH:
        lpmhdr = (LPNMHDR)lParam;
        switch(lpmhdr->code)
        {
        case MCN_SELCHANGE:
            LPNMSELCHANGE lpsel;
            lpsel = (LPNMSELCHANGE)lParam;
            OnSelect(lpmhdr->hwndFrom, hwndMain, lpsel);
            break;
        case MCN_GETDAYSTATE:
            LPNMDAYSTATE lpds;
            lpds = (LPNMDAYSTATE)lParam;
            OnGetDayState(lpds);
            break;
        default:
            break;
        } //End of switch(lpmhdr->code) block
        return (0);
    default:
        return (0);
    } //End of switch(nID) block

```

Figure 5.2 Handling month calendar control notifications.

when the date changes by selecting a month from the pop-up menu or tapping the month scroll buttons. In the example described above where a user taps the same date multiple times, an MCN_SELECT notification would be sent for each of the taps.

Since the MCN_SELCHANGE notification is sent upon any user interaction that changes the date selection in the control, we only need to respond to MCN_SELCHANGE.

When either the MCN_SELCHANGE or MCN_SELECT notification is sent, the *lParam* of the parent window's window procedure is an NMSELCHANGE structure. This structure is defined as:

```

typedef struct tagNMSELCHANGE
{
    NMHDR nmhdr;
    SYSTEMTIME stSelStart;
    SYSTEMTIME stSelEnd;
} NMSELCHANGE, FAR* LPNMSELCHANGE;

```

The two notification-specific members of this structure, *stSelStart* and *stSelEnd*, are `SYSTEMTIME` structures containing date information about the first and last dates in the new date selection range. If the control does not have the `MCS_MULTISELECT` style, *stSelStart* and *stSelEnd* will be the same.

In our sample application, we change title bar text to display the currently selected date in response to these notifications. The application code for handling this notification is the *OnSelect* function:

```
#define IsMultiSelect(hwnd) \
    (GetWindowLong(hwnd, GWL_STYLE) & MCS_MULTISELECT)

void OnSelect(HWND hwndCal, HWND hwndParent,
    LPNMSELCHANGE lpsel)
{
    TCHAR pszText[64];
    //Set caption text only if control is single select
    if (!IsMultiSelect(hwndCal))
    {
        wsprintf(pszText, TEXT("Selected Date: %d\\%d\\%d"),
            lpsel->stSelStart.wMonth,
            lpsel->stSelStart.wDay,
            lpsel->stSelStart.wYear);
        SetWindowText(hwndParent, pszText);
    }
}
```

The *IsMultiSelect* macro just tests if the month calendar control specified has the `MCS_MULTISELECT` style. *OnSelect* says if the control only allows single selection, set the current selected date in the application's main window caption. The selected date in this case is either of the `SYSTEMTIME` members of the `NMSELCHANGE` structure sent by the control with the `MCN_SELCHANGE` notification.

The second notification we respond to is `MCN_GETDAYSTATE`. This notification is sent by the month calendar control to request the day state information, which it uses to determine which dates to display in bold text.

Along with the `MCN_GETDAYSTATE` notification, the control sends an `NMDAYSTATE` structure in the *lParam* of the parent window's window procedure. This structure is defined as:

```
typedef struct tagNMDAYSTATE
{
```

```
NMHDR nmhdr;  
SYSTEMTIME stStart;  
int cDayState;  
LPMONTHDAYSTATE prgDayState;  
} NMDAYSTATE, FAR* LPNMDAYSTATE;
```

The month calendar control requires that applications supply day state information for more than just the current month. For example, if the current month (as determined by the date that the control is currently using as today's date) is June, the control will want day state information for May, June, and July. The *cDayState* member of this structure tells the application exactly how many months' worth of day state information is needed. The *stStart* member indicates the first month for which the control wants day state information. This month is found in the *wMonth* member of the *stStart* SYSTEMTIME structure. *prgDayState* is an array of MONTHDAYSTATE values that the application fills in with the application specific day state information.

Looking at our sample application's *OnGetDayState* function will make this much clearer. As the code in Figure 5.2 previously showed, this function is called in response to the MCN_GETDAYSTATE notification:

```
void OnGetDayState(LPNMDAYSTATE lpds)  
{  
    int i, nStart;  
  
    nStart = lpds->stStart.wMonth-1;  
    for (i=0; i<lpds->cDayState; i++)  
    {  
        //Account for month roll over, i.e., nStart > 11.  
        if (nStart>11)  
        {  
            nStart = 0;  
        }  
        lpds->prgDayState[i] = mdsHoliday[nStart++];  
    }  
}
```

nStart is the index into our *mdsHoliday* array. It is initialized to the starting month indicated by the NMDAYSTATE structure, minus 1. The minus 1 accounts for the fact that in *mdsHoliday*, January corresponds to index 0, but SYSTEMTIME month values are 1-based. Then for each of the months for which the month calendar control is requesting day state information, we assign the holiday day state infor-

mation for that month to the corresponding `MONTHDAYSTATE` value in the `prgDayState` member of the `NMDAYSTATE` structure.

When the window procedure returns after processing this `MCN_GETDAYSTATE` notification, the month calendar control uses the day state information in the `NMDAYSTATE` structure to display the appropriate dates in bold.

Finally, the sample application allows the user to return to today's date by tapping the Today button. This is done in the `OnGotoToday` function:

```
void OnGotoToday(HWND hwndCal)
{
    SYSTEMTIME stToday;
    SendMessage(hwndCal, MCM_GETTODAY, 0, (LPARAM)&stToday);
    SendMessage(hwndCal, MCM_SETCURSEL, 0, (LPARAM)&stToday);
}
```

This function simply determines the date that the control currently uses as today's date with the `MCM_GETTODAY` message, and then sets the current month calendar control selection by sending `MCM_SETCURSEL`. `MCM_GETTODAY` returns a `SYSTEMTIME` structure representing today's date. `MCM_SETCURSEL` takes a `SYSTEMTIME` telling the control what day to set the current selection to.

The Today Button Doesn't Work, Right?

Click on the Today button in the sample application. The month calendar control switches to today's date, but the application caption text doesn't change. This "bug" was left in the sample application to highlight a subtle undesirable feature of the month calendar control.

It turns out that programmatic changes to the current selection in a month calendar control do not trigger `MCN_SELECT` or `MCN_SELCHANGE` notifications. Therefore, the `MCM_SETCURSEL` message sent in the `OnGotoToday` function does not cause the `MCN_SELECT` or `MCN_SELCHANGE` notifications to be sent. Hence, the main application caption text does not change when the Today button is pressed.

This is a serious oversight in the design of the month calendar control. The application developer must manually trigger the notification handlers for each of these notifications.

One method for fixing this bug is presented in the next section.

Before We Move On, Let's Fix the Bug

In the previous section, we pointed out a small bug with the Today button in the month calendar control sample application. This section describes one way to fix this bug.

The caption text in the main window of the application is changed by the *OnSelect* function. Pressing the Today button results in a call to the *OnGotoToday* function. Fixing the bug is as simple as making *OnGotoToday* appropriately call *OnSelect*.

OnSelect requires handles to the month calendar control and the parent window. These are global variables available to any function in the application. Additionally, *OnSelect* needs a pointer to an NMSELCHANGE structure. Actually, it only needs the *wMonth*, *wDay*, and *wYear* components of the *stSelStart* member of such a structure. *OnGotoToday* already obtains this information by sending an MCM_GETTODAY message. The bug can thus be fixed by replacing the original *OnGotoToday* function with the following:

```
void OnGotoToday(HWND hwndCal)
{
    SYSTEMTIME stToday;
    NMSELCHANGE nmsel;
    memset(&nmsel, 0, sizeof(nmsel));
    SendMessage(hwndCal, MCM_GETTODAY, 0, (LPARAM)&stToday);
    SendMessage(hwndCal, MCM_SETCURSEL, 0, (LPARAM)&stToday);
    nmsel.stSelStart = stToday;
    OnSelect(hwndMonth, hwndMain, &nmsel);
}
```

There are very few changes to the *OnGotoToday* function here. We declare an NMSELCHANGE structure, *nmsel*, and initialize its contents to zero. After sending the MCM_SETCURSEL message to the month calendar control, we assign the *stSelStart* member of *nmsel* to the DATETIME structure retrieved by the MCM_GETTODAY message.

hwndMonth and *hwndMain* are the global variables containing the window handles of the month calendar control and the main application window, respectively. The *OnSelect* call therefore has all the information it needs to update the main window caption correctly.

Month Calendar Control Messages and Notifications

The complete list of the control messages and notifications associated with the month calendar control are described in Tables 5.2 and 5.3.

Table 5.2 Month Calendar Control Messages

MESSAGE	MEANING
MCM_GETCOLOR	Retrieves the color of the specified part of the control.
MCM_GETCURSEL	Gets the SYSTEMTIME structure corresponding to the currently selected date.
MCM_GETFIRSTDAYOFWEEK	Returns the first day of the week displayed for each week in the control.
MCM_GETMAXSELCOUNT	Returns the maximum number of days that can be selected at one time in the control.
MCM_GETMAXTODAYWIDTH	Returns the maximum width of the Today string displayed at the bottom of month calendar controls.
MCM_GETMINREQRECT	Returns the minimum width and height required to display one full calendar month.
MCM_GETMONTHDELTA	Returns the number of months that the control advances or retreats when the user taps the right or left month scroll button.
MCM_GETMONTHRANGE	Returns SYSTEMTIME structures representing the maximum and minimum dates that can be displayed by the control.
MCM_GETRANGE	Retrieves the maximum and minimum allowable dates set for the control.
MCM_GETSELRANGE	Gets the range of dates currently selected in a control with the MCS_MULTISELECT style.
MCM_GETTODAY	Retrieves the date currently set as today's date in the control.
MCM_HITTEST	Determines which part of the control contains the specified point.
MCM_SETCOLOR	Sets the color of the specified part of the control.
MCM_SETCURSEL	Sets the current date selection in the control. Cannot be used with controls with the MCS_MULTISELECT style.
MCM_SETDAYSTATE	Sets the day state information for days that are currently visible in the control.
MCM_SETFIRSTDAYOFWEEK	Sets the day (Monday, Tuesday, etc.) to use as the first day of each week displayed in the control.
MCM_SETMAXSELCOUNT	Sets the maximum number of days that can be selected in a control.
MCM_SETMONTHDELTA	Sets the number of months the control advances or retreats when a user taps the right or left month scroll button.

(Continues)

Table 5.2 Month Calendar Control Messages (Continued)

MESSAGE	MEANING
MCM_SETRANGE	Sets the maximum and minimum dates for a control.
MCM_SETSELRANGE	Sets the range of currently selected dates for a control. Message only applies to controls with the MCS_MULTISELECT style.
MCM_SETTODAY	Sets the date that the control specifies as today's date.

The Complete Sample Application

The complete source code for the month calendar control sample application is shown below.

month.h

```
#ifndef __MONTH_H_
#define __MONTH_H_
#define MAX_STRING_LENGTH 129
#define IsMultiSelect(hwnd) \
    (GetWindowLong(hwnd, GWL_STYLE) & MCS_MULTISELECT)
/Child control IDs
#define IDC_EXIT 1024
#define IDC_TODAY 1025
#define IDC_MONTH 1026
#define ONE 0x01
TCHAR pszAppName[] = TEXT("MONTHSAMPLE");
TCHAR pszTitle[] = TEXT("Windows CE Month Calendar Control");
HINSTANCE ghInst;
int nWidth;
int nHeight;
/* Define the various windows used in this application:
   hwndMain -> The main application window.
```

Table 5.3 Month Calendar Control Notifications

NOTIFICATION	MEANING
MCN_GETDAYSTATE	Sent by a control to request day state information used to determine which dates to display in bold.
MCN_SELCHANGE	Sent by a control anytime the currently selected date or range of dates changes.
MCN_SELECT	Sent by a control whenever the user explicitly selects a new current date or range of dates, i.e., the user taps a specific date in the calendar.

```

    hwndExit -> Exit button
    hwndToday -> Goto Today button
    hwndMonth -> Month calendar control
*/
HWND hwndMain;
HWND hwndExit;
HWND hwndToday;
HWND hwndMonth;
//MONTHDAYSTATES for holidays
MONTHDAYSTATE mdsHoliday[12] = {(ONE | (ONE<<21)), //January
    (ONE<<13), //February
    (ONE<<16), //March
    0, //April
    0, //May
    0, //June
    (ONE<<3), //July
    0, //August
    0, //September
    (ONE<<30), //October
    0, //November
    (ONE<<24) //December};
void OnInitMonthCalendar(HWND hwndCal, int nLeft, int nTop);
void OnSelect(HWND hwndCal,
    HWND hwndParent,
    LPNMSELCHANGE lpsel);
void OnGetDayState(LPNMDAYSTATE lpds);
void OnGotoToday(HWND hwndCal);
LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
    WPARAM wParam, LPARAM lParam);
#endif

```

main.cpp

```

#include <windows.h>
#include <commctrl.h>
#include "month.h"
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine,
    int nCmdShow)
{
    MSG msg;
    RECT rc;
    INITCOMMONCONTROLSEX icex;
    WNDCLASS wc;
    ghInst = hInstance;
    wc.style = 0;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;

```

```

wc.hInstance = hInstance;
wc.hIcon = NULL;
wc.hCursor = NULL;
wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wc.lpszMenuName = NULL;
wc.lpszClassName = szAppName;
RegisterClass(&wc);
icex.dwSize = sizeof(icex);
icex.dwICC = ICC_DATE_CLASSES;
InitCommonControlsEx(&icex);
SystemParametersInfo(SPI_GETWORKAREA, NULL,
    &rc, NULL);
nWidth = rc.right;
nHeight = rc.bottom;
hwndMain = CreateWindow(szAppName, szTitle,
    WS_VISIBLE|WS_BORDER|WS_CAPTION,
    0,0,nWidth, nHeight,
    NULL, NULL, hInstance, NULL);
hwndExit = CreateWindow(TEXT("BUTTON"),TEXT("Exit"),
    WS_VISIBLE|WS_CHILD|BS_PUSHBUTTON,
    0,0,65,35, hwndMain,
    (HMENU)IDC_EXIT, hInstance, NULL);
hwndToday = CreateWindow(TEXT("BUTTON"),
    TEXT("Today"),
    WS_VISIBLE|WS_CHILD|BS_PUSHBUTTON,
    0,37,65,35,
    hwndMain, (HMENU)IDC_TODAY,
    hInstance, NULL);
hwndMonth = CreateWindowEx(MONTHCAL_CLASS,
    NULL,
    WS_VISIBLE|WS_BORDER|WS_CHILD|MCS_DAYSTATE,
    0,0,0,0,
    hwndMain, (HMENU)IDC_MONTH,
    hInstance, NULL);
OnInitMonthCalendar(hwndMonth, 70, 0);
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage (&msg);
    DispatchMessage(&msg);
}
return(msg.wParam);
}
LRESULT CALLBACK WndProc(HWND hwnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    UINT nID;
    switch(message)
    {

```

```

    case WM_NOTIFY:
        LPNMHDR lpmhdr;
        nID = (UINT)wParam;
        switch(nID)
        {
        case IDC_MONTH:
            lpmhdr = (LPNMHDR)lParam;
            switch(lpmhdr->code)
            {
            case MCN_SELCHANGE:
                LPNMSELCHANGE lpsel;
                lpsel = (LPNMSELCHANGE)lParam;
                OnSelect(lpmhdr->hwndFrom, hwndMain, lpsel);
                break;
            case MCN_GETDAYSTATE:
                LPNMDAYSTATE lpds;
                lpds = (LPNMDAYSTATE)lParam;
                OnGetDayState(lpds);
                break;
            default:
                break;
            } //End of switch(lpmhdr->code) block
            return (0);
        default:
            return (0);
        } //End of switch(nID) block
    case WM_COMMAND:
        nID = LOWORD(wParam);
        switch(nID)
        {
        case IDC_TODAY:
            OnGotoToday(hwndMonth);
            break;
        case IDC_EXIT:
            DestroyWindow(hwnd);
            PostQuitMessage(0);
            break;
        default:
            break;
        } //End of switch(nID) statement
        return (0);
    default:
        return (DefWindowProc(hwnd,message,wParam,lParam));
    } //End of switch(message) statement
}

void OnInitMonthCalendar(HWND hwndCal, int nLeft, int nTop)
{
    RECT r;
    SendMessage(hwndCal, MCM_SETCOLOR, MCSC_MONTHBK,
        (LPARAM)RGB(192,192,192));
    SendMessage(hwndCal, MCM_SETCOLOR, MCSC_TITLEBK,

```

```

        (LPARAM)RGB(0,0,0));
SendMessage(hwndCal, MCM_SETCOLOR, MCSC_BACKGROUND,
        (LPARAM)RGB(128,128,128));
SendMessage(hwndCal, MCM_GETMINREQRECT,
        0, (LPARAM)(LPRECT)&r);
SetWindowPos(hwndCal, NULL, nLeft, nTop,
        r.right, r.bottom, SWP_NOZORDER);
}
void OnSelect(HWND hwndCal,
        HWND hwndParent,
        LPNMSELCHANGE lpsel)
{
    TCHAR pszText[64];
    //Set caption text only if control is single select
    if (!IsMultiSelect(hwndCal))
    {
        wsprintf(pszText, TEXT("Selected Date: %d\\%d\\%d"),
            lpsel->stSelStart.wMonth,
            lpsel->stSelStart.wDay,
            lpsel->stSelStart.wYear);
        SetWindowText(hwndParent, pszText);
    }
}
void OnGetDayState(LPNMDAYSTATE lpds)
{
    int i, nStart;
    nStart = lpds->stStart.wMonth-1;
    for (i=0; i<lpds->cDayState; i++)
    {
        //Account for month roll over, i.e., nStart > 11.
        if (nStart>11)
        {
            nStart = 0;
        }
        lpds->prgDayState[i] = mdsHoliday[nStart++];
    }
}
void OnGotoToday(HWND hwndCal)
{
    SYSTEMTIME stToday;
    SendMessage(hwndCal, MCM_GETTODAY, 0, (LPARAM)&stToday);
    SendMessage(hwndCal, MCM_SETCURSEL, 0, (LPARAM)&stToday);
}

```

The Date Time Picker Control

The date time picker control is closely related to the month calendar control. Since we spent so much time and effort describing the month

calendar control, we will not spend as much time on the date time picker control.

The window class for this control is `DATETIMEPICK_CLASS`.

A date time picker control is an editable text field that can display date and time information in a variety of formats (Figure 5.3). Predefined formats include the long and short formats—for example, “Thursday, July 04, 1776” and “7/4/76” respectively (beware Year 2000-aware folks!).

The control also supports time format. This means that the control just displays time in `hh:mm:ss` format.

Application-specific display formats can be defined in a number of ways. Applications set the format using the `DTM_SETFORMAT` message. With this message the application specifies a format string that the control uses to format its display. This format string can include *callback fields*. In this case, the control sends notifications to which the parent window responds by telling the control what text to display in a particular callback field.

By default, date time picker controls include an arrow button similar to that found in combo boxes. When this button is pressed, a month calendar control appears from which users can then select the current

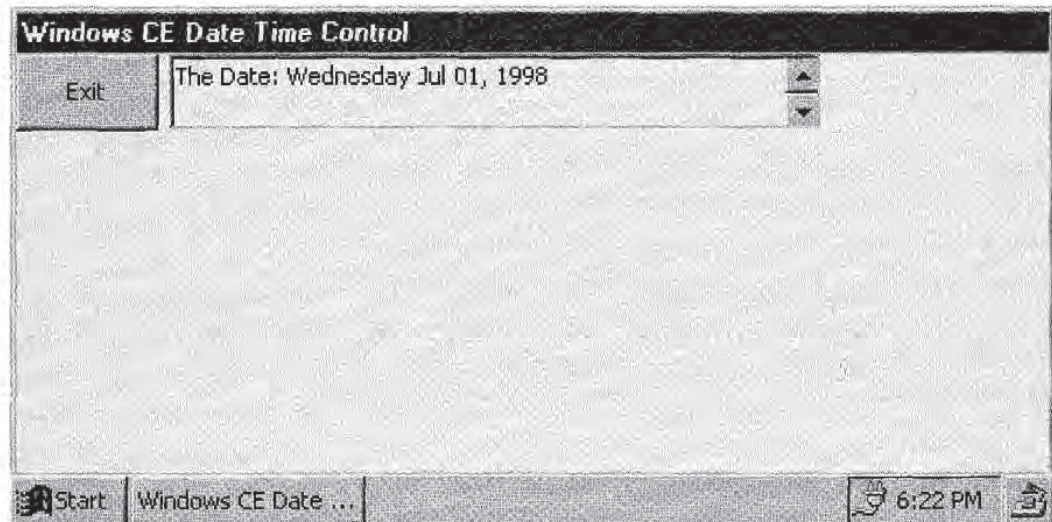


Figure 5.3 The date time picker control.

date. Alternatively, date time pickers can include an up-down control for picking the current date.

The date time picker control also allows the user to type into the edit field of the control.

TIP**DROP-DOWN MONTH CALENDAR CONTROL VERSION OF THE DATE TIME PICKER**

The drop-down month calendar control version of the date time picker control is a good choice for Windows CE applications running on devices with limited touch screen sizes. Applications that require the ability to display dates and times can do so with a minimum of screen real estate using this control style by creating small date time picker controls.

When the control is closed, the date and time information takes up very little space. The more detailed month calendar control only appears temporarily when dropped down by the user.

Date Time Picker Control Styles

The six styles that can be specified for a date time picker control are shown below. Only one of `DTS_LONGDATEFORMAT`, `DTS_SHORTDATEFORMAT`, and `DTS_TIMEFORMAT` can be used with a particular control.

DTS_APPCANPARSE. Indicates that the control can parse strings entered into the control by users. After the user edits the contents of the control, a `DTN_USERSTRING` notification is sent, to which the parent window can respond by interpreting the string in some way.

DTS_LONGDATEFORMAT. Specifies that the control is to display dates in the long date format.

DTS_SHOWNONE. Allows the control to display no date. Used with the `DTM_SETSYSTEMTIME` and `DTM_GETSYSTEMTIME` messages.

DTS_SHORTDATEFORMAT. Specifies that the control is to display dates in the short date format.

DTS_TIMEFORMAT. Specifies that the control displays the time, instead of dates. If this style is specified, the control does not include

A Note on Date Time Picker Controls That Include Month Calendar Controls

Date time picker controls do not keep a static month calendar control. So, for example, if you wrote the following code, you should not expect *hwndCal* to be a valid window:

```

HWND hwndCal, hwndDateTime;
//Init common controls, etc.
//Create the date time picker control...
hwndDateTime = CreateWindow(DATETIMEPICK_CLASS,
    NULL,
    WS_CHILD|WS_VISIBLE|
    DTS_SHORTDATEFORMAT,
    ...);
//...and extract the month calendar control associated with it
hwndCal = (HWND)SendMessage(hwndDateTime,
    DTM_GETMONTHCAL,
    0, 0L);

```

hwndCal will be **NULL**. The reason is that the month calendar control associated with the date time picker is only around between the times that the date time picker sends the **DTN_DROPDOWN** and **DTN_CLOSEUP** notifications.

Therefore, your applications must initialize the month calendar control in response to the **DTN_DROPDOWN** notification, the indication that the month calendar control is being displayed. And yes, this initialization must include proper positioning of the month calendar control window using the **MCM_GETMINREQRECT** technique we saw in the application in the previous section.

a month calendar control, but only an up-down control for time selection.

DTS_UPDOWN. Specifies that the control include an up-down control for date selection instead of a month calendar control. This style is always included for date time picker controls with the **DTS_TIMEFORMAT** style.

An Example

In this section we present a very simple example of how to use the date time picker control. All the example does is create a control that displays dates in the long date format and that responds to some basic text editing user input. Specifically, if the user types "Today" in the

display area of the control, the control sets its current selection to today's date.

The date time picker control in this example uses an up-down control for moving through dates. Since the month calendar control was described in detail in the previous section, including an example of a date time picker using a month calendar control would be redundant.

The control is created with the following *CreateWindow* call. *hwndMain* and *hInstance* are the main application window and the application *HINSTANCE*, respectively.

```
#define IDC_DATETIME 1025
HWND hwndDateTime;
hwndDateTime = CreateWindow(DATETIMEPICK_CLASS,
    TEXT("DateTime"),
    WS_VISIBLE|WS_BORDER|WS_CHILD|DTS_LONGDATEFORMAT|
    DTS_APPCANPARSE|DTS_UPDOWN,
    70,0,300,35,
    hwndMain,
    (HMENU) IDC_DATETIME,
    hInstance,
    NULL);
```

The *DTS_APPCANPARSE* style is set to allow the control's parent to respond to user text input.

The most interesting feature of our application is that if the user types "Today" into the contents of the date time picker, the date time picker sets its date to today's date. The ability of the application to respond to user text input was enabled by the *DTS_APPCANPARSE* style. The control informs its parent that the user has entered text by sending the *DTN_USERSTRING* notification. The application's main window procedure is responsible for responding to this notification. Here is the portion of the window procedure that handles *WM_NOTIFY* messages:

```
case WM_NOTIFY:
    nID = (UINT)wParam;
    switch(nID)
    {
    case IDC_DATETIME:
        lpmhdr = (LPMHDR)lParam;
        switch(lpmhdr->code)
        {
        case DTN_USERSTRING:
            LPNMDATETIMESTRING lpstr;
```

```

    lpstr = (LPNMDATETIMESTRING)lParam;
    if (!lstrcmp(lpstr->pszUserString,TEXT("Today")))
    {
        GetLocalTime(&lpstr->st);
        lpstr->dwFlags = GDT_VALID;
    }
    break;
default:
    break;
} //End of switch(lpnmhdr->code) block
return (0);
default:
    return (0);
} //End of switch(nID) block

```

The DTN_USERSTRING notification is accompanied by an NMDATETIMESTRING structure:

```

typedef struct tagNMDATETIMESTRING
{
    NMHDR nmhdr;
    LPCTSTR pszUserString;
    SYSTEMTIME st;
    DWORD dwFlags;
} NMDATETIMESTRING, FAR* LPNMDATETIMESTRING;

```

pszUserString is the string entered by the user. *st* is a SYSTEMTIME structure that is filled in by the parent of the date time picker control. The date specified will be the date displayed by the control after the main window procedure returns from processing the DTN_USERSTRING notification.

dwFlags can be set to GDT_VALID, indicating that the *st* member of the NMDATETIMESTRING structure is valid and that the control should display this date in the control's current date format. Alternatively, *dwFlags* can be GDT_NONE to tell the control to show no date, which is valid only if the DTS_SHOWNONE style is used.

Our sample application responds to DTN_USERSTRING by comparing the user input string to "Today". If the user typed "Today", the application calls *GetLocalTime* to determine today's date, and sets the *st* member of the NMDATETIMESTRING to this value. Thus the date time picker knows to display today's date.

Specifying Custom Date Time Formats

Date time picker controls are capable of displaying dates and times in formats other than the predefined short, long, and time formats. Speci-

fyng such formats, however, requires some extra work on the part of the application programmer.

The simplest way to specify a different format is to use the `DTM_SETFORMAT` message. This message allows the application to specify a format string to be used by a particular date time picker control. Format strings can include any of a set of predefined format codes. For example, "MMM" tells the control to display the three-character abbreviation for the month, and "dddd" tells it to display the full weekday name. (A full list of these codes is contained in the on-line documentation for the `DTM_SETFORMAT` message.) To embed literal strings inside a format string, enclose the desired text in single quotes.

For example, if the date was Thursday, July 2, 1998, and we wanted our date time picker to display this date as "The Date: Thursday July 02, 1998," our application could set the format string as follows:

```
SendMessage(hwndDateTime, DTM_SETFORMAT, 0,  
            (LPARAM)TEXT("The Date: 'ddddMMMdd', 'yy'));
```

Another way that applications can customize date time picker display formats is by means of callback fields. The application adds "X" characters to the format string specified with the `DTM_SETFORMAT` message. Then, whenever the control needs to display the date time information, it sends its parent `DTN_FORMAT` and `DTN_FORMAT-QUERY` notifications. The application responds to these notifications by specifying the text to use in place of the callback fields, and to indicate the physical size of the text to be displayed.

The application must allow users to enter text in the regions of the string displayed by the date time picker that corresponds to the callback fields. The application handles the `DTN_WMKEYDOWN` notification to respond to user input in callback fields.

Date Time Picker Control Messages and Notifications

Tables 5.4 and 5.5 give a complete list of date time picker control messages and notifications, along with their meanings.

The Complete Sample Application

I'll be the first to admit that this application won't be making any headlines, but it will help make you more familiar with how to use date time picker controls.

Table 5.4 Date Time Picker Control Messages

MESSAGE	MEANING
DTM_GETMCCOLOR	Retrieves the color of the specified part of the month calendar child control contained by the date time picker. Message is only supported for date time pickers that do not have the DTS_UPDOWN style bit set, i.e., that have month calendar controls. Compare to MCM_GETCOLOR.
DTM_GETMCFONT	Retrieves the font currently in use by the month calendar child control contained by the date time picker. Message is only supported for date time pickers that do not have the DTS_UPDOWN style bit set.
DTM_GETMONTHCAL	Retrieves the HWND of the month calendar child control contained by the date time picker. Only supported for date time pickers that do not have the DTS_UPDOWN style bit set.
DTM_GETRANGE	Message gets the range of date time values that the date time picker can display.
DTM_GETSYSTEMTIME	Retrieves the time currently displayed in the date time picker. Time is returned as a SYSTEMTIME.
DTM_SETFORMAT	Message sets the date time picker control's display format string.
DTM_SETMCCOLOR	Message sets the color of the specified part of the month calendar child control contained by the date time picker. Message is only supported for date time pickers that do not have the DTS_UPDOWN style bit set. Compare to MCM_SETCOLOR.
DTM_SETMCFONT	Message sets the font used by the month calendar child control contained by the date time picker. Message is only supported for date time pickers that do not have the DTS_UPDOWN style bit set.
DTM_SETRANGE	Message sets the range of date time values that the date time picker can display.
DTM_SETSYSTEMTIME	Message sets the date and time to be displayed by the date time picker.

datetime.h

```
#ifndef __DATETIME_H_
#define __DATETIME_H_
//Child control IDs
#define IDC_EXIT 1024
#define IDC_DATETIME 1025
```

Table 5.5 Date Time Picker Control Notifications

NOTIFICATION	MEANING
DTN_CLOSEUP	Sent by the control when the user closes the drop-down month calendar child control contained by the date time picker. Only applicable to date time pickers that do not have the DTS_UPDOWN style bit set.
DTN_DATETIMECHANGE	Sent by the control whenever the date time display changes.
DTN_DROPDOWN	Sent by the control when the user opens the drop-down month calendar child control contained by the date time picker. Only applicable to date time pickers that do not have the DTS_UPDOWN style bit set.
DTN_FORMAT	Sent by the control for each callback field in a format string. The application responds by providing the text to display in the callback fields.
DTN_FORMATQUERY	Sent by the control for each callback field in a format string. The application responds by specifying the maximum pixel width of the text that can be displayed in the corresponding callback field.
DTN_USERSTRING	Sent by the control after the user edits text in the date time picker's date time display.
DTN_WMKEYDOWN	Sent by the control whenever the user types in a callback field. Responding to this notification allows the control owner to implement custom behavior for key-strokes such as arrow keys.

```

TCHAR pszAppName[] = TEXT("DATETIMESAMPLE");
TCHAR pszTitle[] = TEXT("Windows CE Date Time Control");
HINSTANCE ghInst;
int nWidth; //Main window width
int nHeight; //Main window height
/* Define the various windows used in this application:
   hwndMain -> The main application window.
   hwndExit -> Exit button
   hwndDateTime -> Date time picker control
*/
HWND hwndMain;
HWND hwndExit;
HWND hwndDateTime;
LRESULT CALLBACK WndProc(HWND hwnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam);
#endif

```


main.cpp

```

#include <windows.h>
#include <commctrl.h>
#include "datetime.h"
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPTSTR lpCmdLine,
    int nCmdShow)
{
    MSG msg;
    RECT rc;
    INITCOMMONCONTROLSEX icex;
    WNDCLASS wc;
    ghInst = hInstance;
    wc.style = 0;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = NULL;
    wc.hCursor = NULL;
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = pszAppName;
    RegisterClass(&wc);
    icex.dwSize = sizeof(icex);
    icex.dwICC = ICC_DATE_CLASSES;
    InitCommonControlsEx(&icex);
    SystemParametersInfo(SPI_GETWORKAREA, NULL,
        &rc, NULL);
    nWidth = rc.right;
    nHeight = rc.bottom;
    hwndMain = CreateWindow(pszAppName,
        pszTitle,
        WS_VISIBLE|WS_BORDER|WS_CAPTION,
        0,0,nWidth,nHeight,
        NULL, NULL, hInstance, NULL);
    hwndExit = CreateWindow(TEXT("BUTTON"),
        TEXT("Exit"),
        WS_VISIBLE|WS_CHILD|BS_PUSHBUTTON,
        0,0,65,35,
        hwndMain, (HMENU)IDC_EXIT,
        hInstance, NULL);
    hwndDateTime = CreateWindow(DATETIMEPICK_CLASS,
        TEXT("DateTime"),
        WS_VISIBLE|WS_BORDER|WS_CHILD|
        DTS_LONGDATEFORMAT|DTS_APPCANPARSE|DTS_UPDOWN,
        70,0,300,35,
        hwndMain, (HMENU)IDC_DATETIME,

```

```

    hInstance, NULL);
SendMessage(hwndDateTime, DTM_SETFORMAT, 0,
    (LPARAM)TEXT("The Date: 'ddddMMdd', 'yy'"));
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage (&msg);
    DispatchMessage(&msg);
}
return(msg.wParam);
}
LRESULT CALLBACK WndProc(HWND hwnd,
    UINT message,
    WPARAM wParam,
    LPARAM lParam)
{
    UINT nID;
    LPNMHDR lpnmhdr;
    switch(message)
    {
    case WM_NOTIFY:
        nID = (UINT)wParam;
        switch(nID)
        {
        case IDC_DATETIME:
            lpnmhdr = (LPNMHDR)lParam;
            switch(lpnmhdr->code)
            {
            case DTN_USERSTRING:
                LPNMDATETIMESTRING lpstr;
                lpstr = (LPNMDATETIMESTRING)lParam;
                if (!lstrcmp(lpstr->pszUserString, TEXT("Today")))
                {
                    GetLocalTime(&lpstr->st);
                    lpstr->dwFlags = GDT_NONE;
                }
                break;
            default:
                break;
            } //End of switch(lpnmhdr->code) block
            return (0);
        default:
            return (0);
        } //End of switch(nID) block
    case WM_COMMAND:
        nID = LOWORD(wParam);
        switch(nID)
        {
        case IDC_EXIT:
            DestroyWindow(hwnd);
            PostQuitMessage(0);

```

```
        break;
    default:
        break;
    } //End of switch(nID) statement
    return (0);
default:
    return (DefWindowProc(hwnd, message, wParam, lParam));
} //End of switch(message) statement
}
```

Rebar Controls

Rebar controls are those nice little draggable strips with buttons or other controls that appear all over applications like Microsoft Developer Studio and Microsoft Word. Applications use rebar controls as an attractive and flexible way to group and arrange related sets of child controls. Figure 5.4 shows this section's sample application using a rebar control.

Rebar controls act as containers for other Windows CE child controls. A rebar control can contain one or more *bands*, each of which in turn can contain one child control. The control contained by a rebar band can be a toolbar, giving the impression of multiple controls in a single

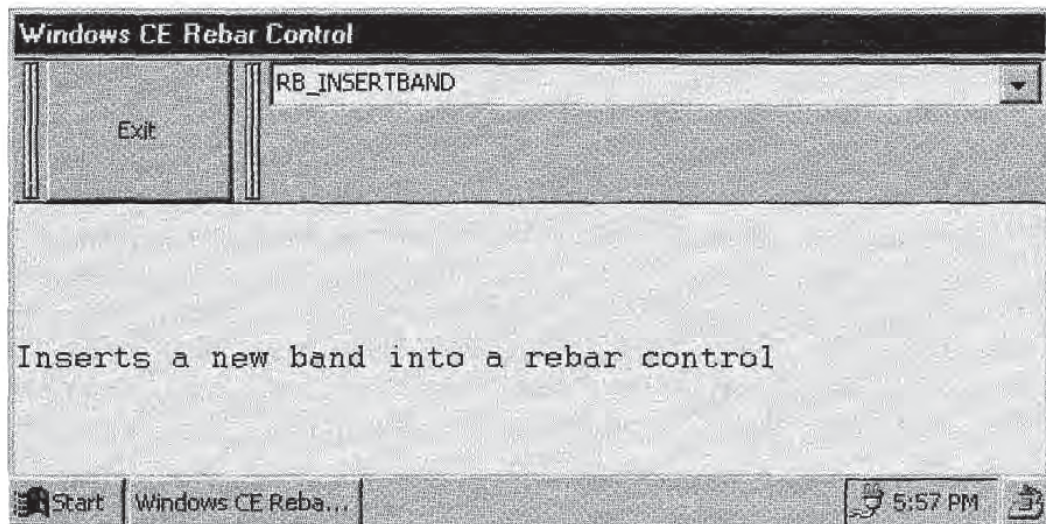


Figure 5.4 Rebar control with two bands.

band. Rebar controls can also include image lists. Bands in a rebar control can display a particular image list bitmap.

Each band in a rebar control can also include a *gripper bar*. A gripper bar appears as two vertical lines that can be used to drag the band.

Rebar Control Styles

There are seven styles that can be used to specify various rebar control characteristics:

CCS_VERT. Causes the control and the bands it contains to display vertically instead of horizontally.

RBS_AUTOSIZE. Rebar band layout automatically updates when child control size or position changes.

RBS_BANDBORDERS. Draws borders around rebar bands.

RBS_FIXEDORDER. Bands can be moved to different rows, but band order is fixed.

RBS_SMARTLABELS. If a band has an icon, the icon is only displayed when the band is minimized. If a band has a text label, the label is only displayed when the band is either in its restored or maximized state.

RBS_VARHEIGHT. Displays bands at the minimum required height if possible. If this style is not set, the height of all bands in the control is set to the height of the tallest band.

RBS_VERTICALGRIPPER. Displays the gripper bar vertically instead of horizontally. Style is ignored if the rebar does not also have the **CCS_VERT** style.

Applications interact with the controls contained by rebar bands in the same way as with any other child control. The application can send the same child control messages to rebar band child controls. The child controls in rebar bands send **WM_COMMAND** messages to the parent of the rebar control. As this is normally the main application window, applications can respond to user interaction with the child controls as they would if the controls were not contained by a rebar control band.

As with most Windows CE controls, there are a number of messages and notifications used by rebar controls. These include functionality for inserting and deleting bands, getting the number of bands in a

rebar control, resizing the rebar control, and the like. Text and background bitmaps can also be added to rebar control bands to further customize their appearance.

It's a safe bet that on the majority of occasions that you choose to use rebar controls, you will use them to group the child controls that drive the functionality of your Windows CE applications, relying on the default behavior of rebar controls to provide other functionality such as moving them with the gripper bar. It is therefore most useful to discuss the procedure for creating rebar controls and inserting bands with child controls into them. This will be the focus of this section.

The sample application for this section demonstrates a rebar control with two bands (see Figure 5.4). The first band contains the omnipotent "Exit" button. The second band contains a combo box. This combo box provides us some relief from the monotony of listing the messages and notifications supported by rebar controls at the end of chapter. Instead, the combo box lists all of the rebar control messages and notifications. Selecting an item in the combo box causes the application to display a description of the corresponding message or notification.

All of the new functionality presented in the sample application is related to creating rebar controls and inserting bands. Since this is described in detail in the pages that follow, listing the entire sample application at the end of the chapter is unnecessary.

Creating Rebar Controls

A rebar control is created using the REBARCLASSNAME control class.

The control is created with the following *CreateWindow* call. *hwndMain* and *hInstance* are the main application window and the application *HINSTANCE*, respectively.

```
#define IDC_REBAR 1024
HWND hwndRebar;
hwndRebar = CreateWindow(REBARCLASSNAME, NULL,
    WS_CHILD|WS_VISIBLE|WS_BORDER|
    RBS_VARHEIGHT|RBS_BANDBORDERS,
    0,0,0,0,
    hwndMain, (HMENU)IDC_REBAR,
    hInstance,NULL);
```

As was the case with the month calendar control, the *x*, *y*, *nWidth*, and *nHeight* parameters can be set to zero. The dimensions of the bands are what really matter, and these are inserted after the rebar control is created.

Rebar Control Bands

Bands are the real nucleus of a rebar control. The bands are what contain the child controls and define the appearance of the rebar control.

To fully understand bands, we must first look at how Windows CE represents bands. We can then explore how to add bands to rebar controls.

The REBARBANDINFO Structure

All information describing a band is specified in terms of a REBARBANDINFO structure. REBARBANDINFO information is supplied by applications when inserting bands into rebar controls. It can also be queried by an application to get information about existing rebar control bands. The structure is defined as:

```
typedef struct tagREBARBANDINFO
{
    UINT cbSize;
    UINT fMask;
    UINT fStyle;
    COLORREF clrFore;
    COLORREF clrBack;
    LPTSTR lpText;
    UINT cch;
    int iImage;
    HWND hwndChild;
    UINT cxMinChild;
    UINT cyMinChild;
    UINT cx;
    HBITMAP hbmBack;
    UINT wID;
    UINT cyChild;
    UINT cyMaxChild;
    UINT cyIntegral;
    UINT cxIdeal;
    LPARAM lParam;
} REBARBANDINFO, FAR* LPREBARBANDINFO;
```

cbSize just indicates the size of the REBARBANDINFO structure. Applications set this value using the *sizeof* function.

The rest of the members of this structure are used to describe various characteristics of a particular rebar band. Not all of the members are necessarily used. The *fMask* member defines which members are valid for a given instance of the structure. *fMask* can be one or more of the following values (the RBBIM prefix stands for “rebar band info mask”):

RBBIM_BACKGROUND. Indicates that the *hbmBack* member is valid.

RBBIM_CHILD. The *hwndChild* member is valid.

RBBIM_CHILD_SIZE. The *cxMinChild* and *cyMinChild* members are valid.

RBBIM_COLORS. The *clrFore* and *clrBack* members are valid.

RBBIM_IDEAL_SIZE. The *cxIdeal* member is valid.

RBBIM_ID. The *wID* member is valid.

RBBIM_IMAGE. The *iImage* member is valid.

RBBIM_LPARAM. The *lParam* member is valid.

RBBIM_SIZE. The *cx* member is valid.

RBBIM_STYLE. The *fStyle* member is valid.

RBBIM_TEXT. The *lpText* member is valid.

The *fStyle* member is used to specify various styles for the band in question. Just as the parent rebar control has a set of styles associated with it, each band in a rebar control can have its own style attributes. The *fStyle* member can be one or more of the following (the RBBS prefix stands for “rebar band style”):

RBBS_BREAK. Indicates that the band is on a new line, i.e., in a new row.

RBBS_CHILDEDGE. The band has an edge at the top and bottom.

RBBS_FIXEDBMP. If the band has a background bitmap, the bitmap does not move when the band is resized.

RBBS_FIXEDSIZE. The band cannot be moved/sized, and no gripper bar is displayed.

RBBS_GRIPPERALWAYS. The band always displays a gripper bar, even if the RBBS_FIXEDSIZE style is set.

RBBS_HIDDEN. Makes the band invisible.

RBBS_NOVERT. The band will not be displayed if the parent rebar control uses the CCS_VERT style.

RBBS_VARIABLEHEIGHT. The band can be resized by the rebar control. The *cyIntegral* and *cyMaxChild* members of the corresponding REBARBANDINFO structure control the resizing.

The *clrFore* and *clrBack* members specify the band's foreground and background colors, respectively. These colors are ignored if the *hbmBack* member is valid.

lpText contains the text label used with the band. *cch* specifies the size of *lpText* in bytes.

iImage is the zero-based index of the image to display with the band. The rebar control must be using an image list in this case.

The *hwndChild* member specifies the HWND of the child control contained by the band. *cxMinChild* and *cyMinChild* specify the minimum width and height of the control. The band cannot be smaller than these values. Similarly, *cyMaxChild* specifies the maximum child control height, and hence the maximum band height. This value is ignored if the band does not have the RBBS_VARIABLEHEIGHT style. *cx* specifies the width of the band. *cyChild* specifies the initial height of the band. It is also ignored if RBBS_VARIABLEHEIGHT is not set.

The *hbmBack* member specifies a bitmap to use as the band background. *wID* is used to identify the band in Custom Draw notifications.

The *cyIntegral* member defines the smallest number of pixels that the band grows or shrinks when resized. This member is ignored if the band does not have the RBBS_VARIABLEHEIGHT style.

cxIdeal specifies the ideal band width. If the band is maximized to its ideal width via the RB_MAXIMIZEBAND message, the rebar control tries to make the band this size.

Finally, *lParam* is a 32-bit value which the application can use to store any other application-defined information with the corresponding rebar control band.

Inserting Bands into Rebar Controls

Now that we understand how Windows CE represents bands, adding bands to a rebar control is straightforward.

Bands are inserted by sending the message RB_INSERTBAND to the rebar control. The *wParam* of this message is a UINT specifying the zero-based index of the band. The *lParam* is a pointer to a REBAR-

BANDINFO structure which contains all the characteristics of the band to be inserted.

For example, to create the band containing the Exit button shown in Figure 5.4, the application includes the following code. *hwndRebar* is the HWND of the rebar control created previously:

```

HWND hwndExit;
REBARBANDINFO rbbi;
hwndExit = CreateWindow(TEXT("BUTTON"),
    TEXT("Exit"),...);
memset(&rbbi, 0, sizeof(rbbi));
rbbi.cbSize = sizeof(rbbi);
rbbi.fMask = (RBBIM_CHILD|RBBIM_CHILDSIZE|
    RBBIM_STYLE|RBBIM_SIZE);
rbbi.fStyle = (RBBBS_GRIPPERALWAYS);
rbbi.hwndChild = hwndExit;
rbbi.cxMinChild = 30; //Band, button min width
rbbi.cyMinChild = 65; //Band, button min height
rbbi.cx = 100; //Band width
SendMessage(hwndRebar, RB_INSERTBAND, 0, (LPARAM)&rbbi);

```

The *fMask* member of *rbbi* indicates that the band will have a child control (RBBIM_CHILD) and that the *cxMinChild* and *cyMinChild* members of *rbbi* are valid (RBBIM_CHILDSIZE.) *rbbi.fStyle* is also valid, as indicated by the RBBIM_STYLE mask bit. RBBIM_SIZE indicates that the *cx* member of *rbbi* is used to specify the width of the band.

The *fStyle* member of *rbbi* specifies that the band to be inserted will always display a gripper bar.

The band will contain the “Exit” button because *rbbi.hwndChild* is set to the HWND of that button. The code goes on to specify that this button (and therefore the band) cannot be smaller than 30 pixels high and 65 pixels wide, and that the band will be 100 pixels wide.

That’s all there is to it. With this brief introduction, and a quick look at the sample application to familiarize yourself with rebar control messages and notifications, you are well on your way to enhancing your Windows CE applications with rebar controls.

Command Bands

A command band control is a fancy rebar control that can contain OK, Close, and Help buttons, called *adornments*, which rebar controls alone do not support (Figure 5.5). Like command bars, command bands can

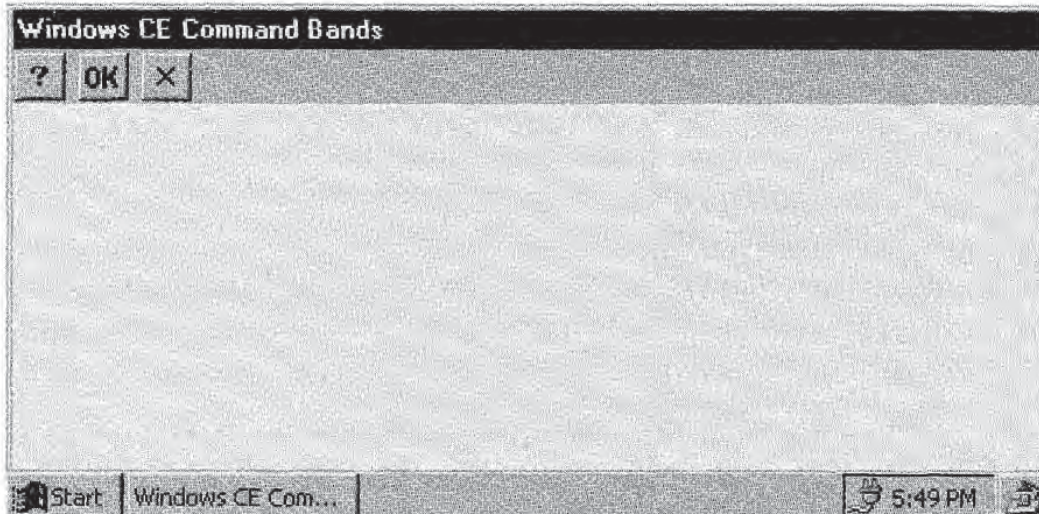


Figure 5.5 Command band with adornments.

contain child controls (Figure 5.6). In addition, each band in a command band control contains a command bar control by default. This means that a programmer can use command bands to construct windows containing multiple menus (Figure 5.6). In the Windows CE control hierarchy, we can think of command band controls as a superset of command bar controls.

All of the operations you might want to perform with command band controls have been wrapped into command band API functions.

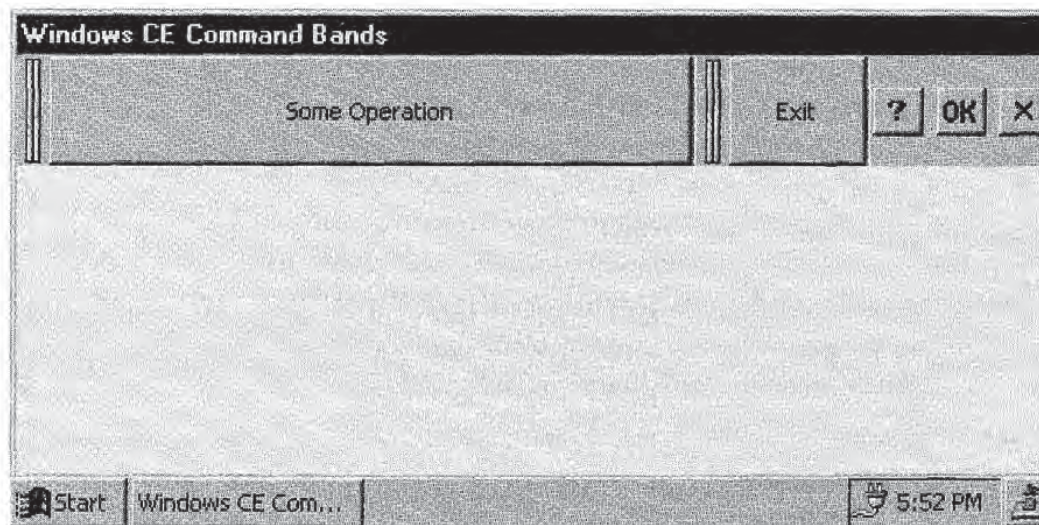


Figure 5.6 Command band with adornments and child control bands.

Instead of sending messages to the control explicitly, your application calls these functions in order to use command bands.

Many of these functions will remind you of the command band's sibling, the command bar. In fact, the application programming interface for creating and using command bands is almost identical to that for command bars.

Since we have already looked at how rebar controls work, and using the command band API will be nothing new because of our familiarity with command bars, this section will quickly introduce command band concepts and then move on.

Command Band Functions

The command band API is very similar to the command bar API in both form and usage. For example, usage of *CommandBands_AddAdornments* is the same as *CommandBar_AddAdornments*.

In all of the functions that follow, the *hinst* parameter (if present) is the application HINSTANCE.

```
CommandBands_AddAdornments(hwndCmdBands, hinst, dwFlags, prbbi);
```

CommandBands_AddAdornments inserts the Close button into the command band control specified by the *hwndCmdBands* parameter. Additionally, a Help or OK button (or both) can be inserted by specifying the appropriate values in *dwFlags*: *CMDBAR_HELP* adds the Help button, and *CMDBAR_OK* adds the OK button. *prbbi* points to a *REBARBANDINFO* structure. This structure can be used to customize the properties of the band that contains the adornment buttons. *prbbi* can also be NULL.

```
CommandBands_AddBands(hwndCmdBands, hinst, cBands, prbbi);
```

CommandBands_AddBands inserts bands into the command band control specified by *hwndCmdBands*. The number of bands to be inserted is in *cBands*. *prbbi* is an array of *REBARBANDINFO* structures defining the bands to be inserted. Both *CommandBands_AddAdornments* and *CommandBands_AddBands* return TRUE if successful and FALSE if they fail.

```
CommandBands_Create(hinst, hwndParent, wID, dwStyles, himl);
```

CommandBands_Create creates a new command band control. *hwndParent* is the parent of the control. *wID* is the command band control iden-

tifier. *dwStyles* contains the command band control styles. Command bands use the same style specifiers as rebar controls. *himgl* is the handle of an image list containing the images to be used with the bands. This parameter can be NULL.

If successful, *CommandBands_Create* returns the HWND of the newly created command band control. Otherwise it returns NULL.

```
CommandBands_GetCommandBar (hwndCmdBands, uBand);
```

CommandBands_GetCommandBar is the poorly named function that retrieves the HWND of the band specified by the zero-based index *uBand*. If unsuccessful, this function returns NULL.

```
CommandBands_Height (hwndCmdBands);
```

CommandBands_Height returns the height in pixels of the specified command band control.

```
CommandBands_IsVisible (hwndCmdBands);
```

CommandBands_IsVisible determines whether the specified control is visible. The function returns TRUE if the control is visible and FALSE if not.

```
CommandBands_GetRestoreInformation (hwndCmdBands, uBand, pcb);
```

CommandBands_GetRestoreInformation gets a *COMMANDBANDSRESTOREINFO* structure for the band specified by *uBand*. This structure, returned in the *pcb* parameter, contains size, maximized/minimized state information, and the like for the band. This information is usually obtained by an application before it closes. This information can then be held in persistent storage so that the next time the application starts, command bands can be restored to their previous states. The function returns TRUE if successful and FALSE if it fails.

```
CommandBands_Show (hwndCmdBands, fShow);
```

CommandBands_Show is used to show or hide the specified command band control. *fShow* is TRUE to make the control visible, FALSE to hide it. The previous state of the control is returned.

Concluding Remarks

The discussion of command band controls concludes our introduction of Windows CE application building blocks. At this point, you have

enough background to build the framework for a huge number of useful Windows CE programs.

Part II of this book shows you how to take advantage of the various persistent storage options available in Windows CE. We will look at the Windows CE file system and registry, as well as how to use Windows CE database technology. These components give you a wide variety of options for storing, retrieving, and organizing the information used by your Windows CE applications.

Windows CE Persistent Storage

Memory in Windows CE devices consists of some amount of read-only memory, or ROM, and some amount of random access memory, also known by its acronym RAM. ROM is where the Windows CE operating system and applications that ship with Windows CE devices are stored.

RAM on a Windows CE device is divided into two sections. The first section is used as *program memory*. This memory is used, for example, by heaps and stacks. For example, whenever your Windows CE applications call *LocalAlloc* to reserve memory on the application's default heap, the memory that is reserved is in program memory. The applications that you write and download to devices also reside in program memory.

The other section of Windows CE RAM is devoted to the *object store*. The object store is used for persistent storage. Persistent storage in Windows CE consists of files and directories, databases and database records, and the Windows CE registry. All of these types of storage objects are called persistent because powering off the Windows CE device on which they are stored does not cause the data they contain to be lost.

In the following chapters, we will discuss Windows CE persistent storage in detail. Each chapter is dedicated to one of the persistent storage classes, such as databases or the registry, and to programming techniques and the various APIs for using them in your applications. But before exploring each of these specific items in detail, we must first discuss the general features of the Windows CE object store.

Object Identifiers

Any object that resides in the Windows CE object store is assigned a unique *object identifier* by the operating system. This identifier is used, for example, to specify a database to be opened or a database record to delete. This object identifier is of type CEOID, one of the basic Windows CE data types.

Given the unique identifier of any object in the object store, an application can extract all of the other information about the object. For instance, given the object identifier of a particular file, the application can determine such information as the name of the file or the directory that contains the file. From a database identifier, an application can determine the number of records stored in the database or the total number of bytes of object store memory used by the database.

The *CeOidGetInfo* Function

Information about a particular object in the object store is retrieved using the function *CeOidGetInfo*. Because it can be used to get information about so many types of objects, this function is of paramount importance when working with the Windows CE object store.

It is important to keep in mind that this function does not retrieve the *contents* of an object store object. For example, it cannot be used to directly read the data contained in a particular Windows CE database record. A useful analogy is to think of the information obtained by *CeOidGetInfo* as similar to the kind of information you get from the Windows NT Explorer on a desktop PC. *CeOidGetInfo* can give you information about an object's relationship to other objects in the object store, as well as information such as file length, database size, or object names. Other Windows CE API functions must be used to access or modify the data represented by an object identifier.

The syntax of *CeOidGetInfo* is:

```
CeOidGetInfo(oid, poidInfo);
```

oid is the object identifier of the object of interest. *poidInfo* is a pointer to a CEOIDINFO structure through which the function returns all the information about the object identified by *oid*. The CEOIDINFO structure is defined as:

```
typedef struct _CEOIDINFO
{
    WORD wObjType;
    WORD wPad;
    union {
        CEFILEINFO infFile;
        CEDIRINFO infDirectory;
        CEDBASEINFO infDatabase;
        CERECORDINFO infRecord;
    };    //End of union
} CEOIDINFO;
```

wObjType identifies the type of object represented by the object identifier passed to *CeOidGetInfo*. *wObjType* can be one of the following values:

OBJTYPE_INVALID	Specified object identifier not found in the object store
OBJTYPE_FILE	Object specified by object identifier is a file
OBJTYPE_DIRECTORY	Object specified by object identifier is a directory
OBJTYPE_DATABASE	Object specified by object identifier is a database
OBJTYPE_RECORD	Object specified by object identifier is a database record

The *wPad* member is a WORD that is in the structure only to align the structure on a double-word boundary. It therefore contains no information about the object queried with *CeOidGetInfo*.

The last member of the CEOIDINFO structure is a union containing an object information structure with the attributes of the object being queried. Which member of this union is valid depends on the value of *wObjType*. For example, if the object queried is a database (as indicated by a *wObjType* value of OBJTYPE_DATABASE), the member you would use is *infDatabase*.

In all there are four such object information structures: CEFILEINFO, CEDIRINFO, CERECORDINFO, and CEDBASEINFO. In the next three chapters the use of these structures will be covered in greater detail. The definition of each is given in the following sections.

The CEFILEINFO Structure

CEFILEINFO structures are used to describe files in the object store:

```
typedef struct _CEFILEINFO
{
    DWORD dwAttributes;
    CEOID oidParent;
    WCHAR szFileName[MAX_PATH];
}
```



```

    FILETIME ftLastChanged;
    DWORD dwLength;
} CEFILEINFO;

```

dwAttributes contains the attributes of the file. *oidParent* is the object identifier of the file's parent directory. If NULL, the file is at the top level of the file system. *oidParent* can be passed to a subsequent call of *CeOidGetInfo* to get information about a file's parent directory. *szFileName* is a null-terminated Unicode string containing the full path and file name of the file. *ftLastChanged* indicates when the file was last modified, and *dwLength* gives the length of the file in bytes.

The CEDIRINFO Structure

Directories are described with CEDIRINFO:

```

typedef struct _CEDIRINFO
{
    DWORD dwAttributes;
    CEOID oidParent;
    WCHAR szDirName[MAX_PATH];
} CEDIRINFO;

```

dwAttributes contains the attributes of the directory. *oidParent* is the object identifier of this directory's parent. If NULL, the directory is in the root directory of the file system. *szDirName* contains the full path name of the directory.

The CERECORDINFO and CEDBASEINFO Structures

The CERECORDINFO structure contains information about a particular Windows CE database record:

```

typedef struct _CERECORDINFO
{
    CEOID oidParent;
} CERECORDINFO;

```

This structure contains only one member, *oidParent*. *oidParent* contains the object identifier of the Windows CE database to which this record belongs.

A CEDBASEINFO structure contains details of a Windows CE database object. This structure will be defined and discussed in detail in the Chapter 7.

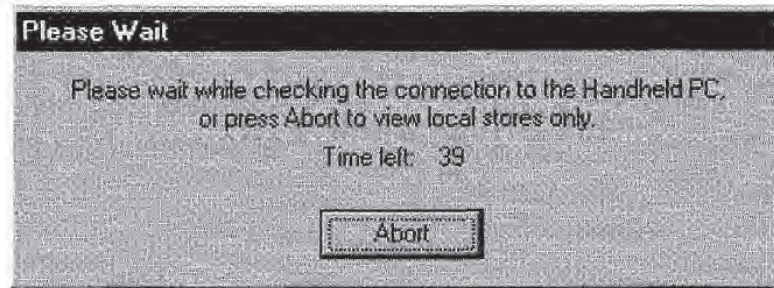


Figure II.1 Connecting to the Remove Object Viewer.

Viewing the Windows CE Object Store

The Windows CE Toolkit provides a tool for visually examining the object store on either a Windows CE device or the Windows CE emulator. To invoke it, select the Remote Object Viewer options from the Tools menu in Microsoft Developer Studio. The dialog box shown in Figure II.1 will appear.

If you have a Windows CE device connected to your PC, the Remote Object Viewer will connect to it and display the object store on the device. Otherwise, to view the emulator's object store, press the abort button.

The Remote Object Viewer works much like the Windows NT Explorer. It contains a tree view user interface that allows you to browse the hierarchy of files on the drives attached to the Windows NT machine on which you are running the Remote Object Viewer.

For example, in Figure II.2 you can see a root item in the tree labeled C:Drive. Expanding this node would display all of the directories on my C: drive.

More interesting, however, is that the Remote Object Viewer allows you to browse both the Windows CE file system and all of the databases contained in the object store. This tool is very useful for quickly creating folders, moving and deleting files, or viewing database records.

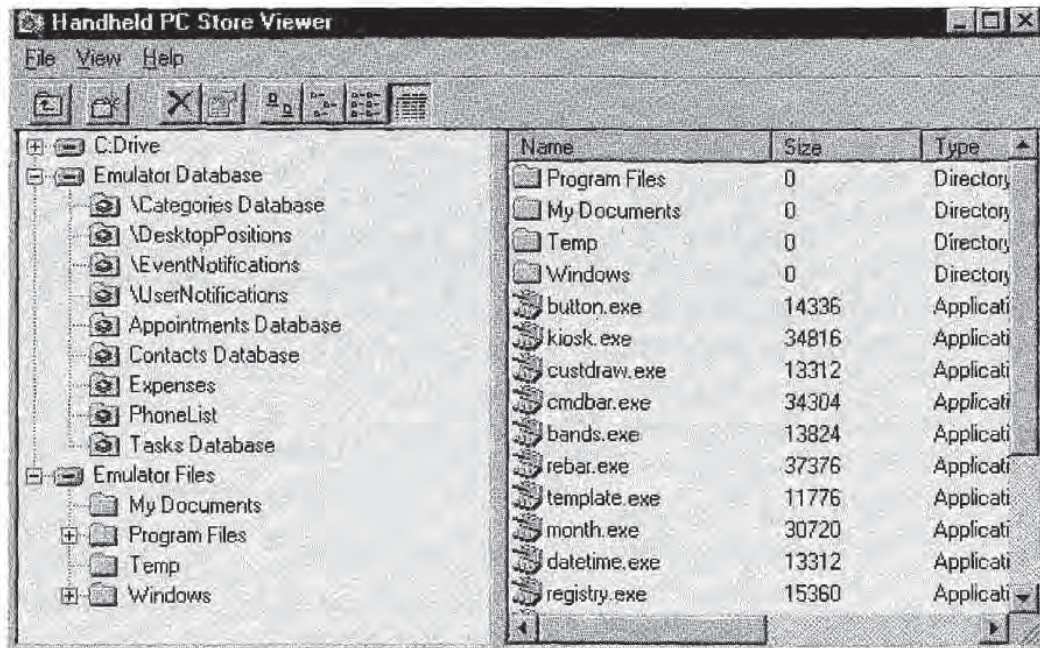


Figure II.2 The Remote Object Viewer.

In Figure II.2, the Emulator Database tree view item contains all of the databases currently stored by the Windows CE emulation object store. Similarly, the Emulator Files item contains the hierarchy of directories and files in the emulation file system. If the host Windows NT PC were connected to a Windows CE device such as a handheld PC or a palm-size PC, the Remote Object Viewer would instead show the databases and file system on that device.

Using the Remote Object Viewer is very much like using the Windows NT Explorer. You can view the contents of directories by expanding the corresponding folder icon. Menu options allow you to rename and delete files or folders. You can also create new folders. You can change the way the contents are displayed with the various View menu options.

The Remote Object Viewer also allows you to transfer files between your desktop computer and the Windows CE emulation environment. If your computer is connected to an actual Windows CE device, you

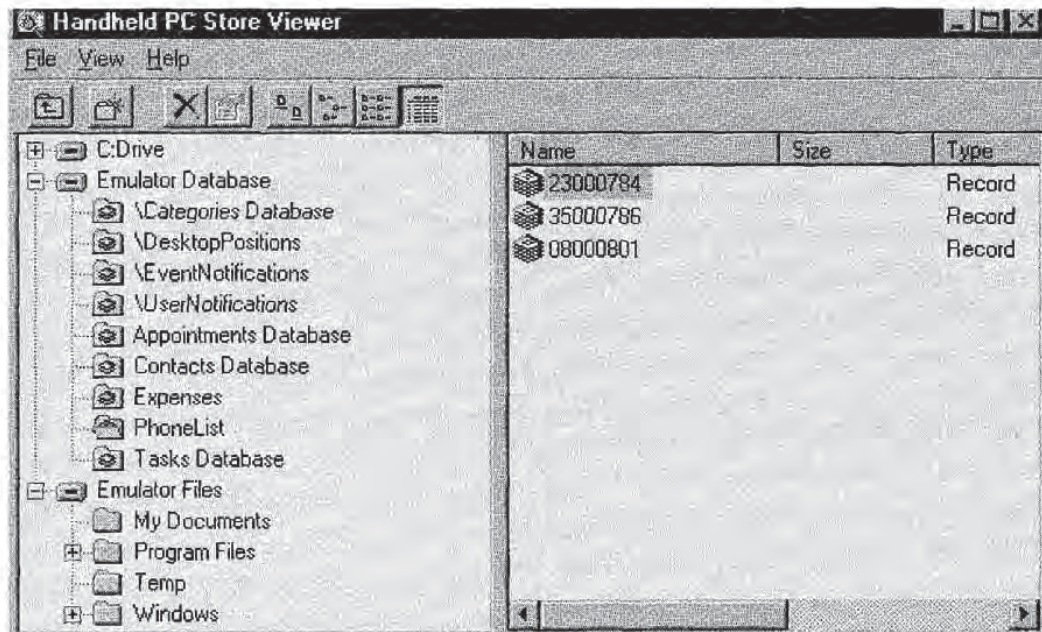
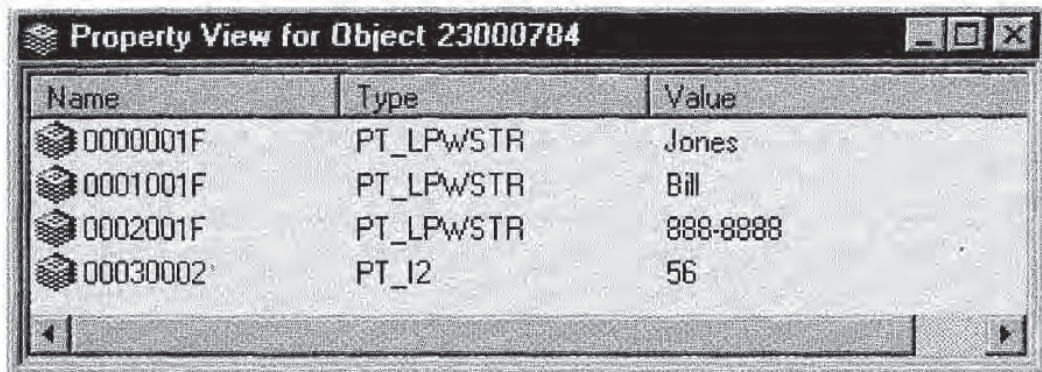


Figure II.3 Browsing the contents of the phone list database with the Remote Object Viewer.

can also transfer files between the computer and the device. As with the Windows NT Explorer, simply drag the icon representing the file to be transferred to the folder where you wish the file to reside.

For databases, the Remote Object Viewer allows you to look at all of the records in a particular database. For example, Figure II.3 shows the contents of the phone list database associated with an application we will see in Chapter 7. Each item in the right-hand pane of the Remote Object Viewer is a database record labeled with its unique CEOID object identifier.

If you select an individual database record in the right-hand pane and then select the Properties option from the File menu, the window in Figure II.4 is displayed. This window details the contents of the selected database record. It displays the record property data types and values. The subjects of Windows CE database records, record properties, and the like are covered in detail in Chapter 7.



The screenshot shows a window titled "Property View for Object 23000784". Inside the window is a table with three columns: "Name", "Type", and "Value". The table contains four rows of data, each with a small cube icon to the left of the "Name" column. The data is as follows:

Name	Type	Value
0000001F	PT_LPWSTR	Jones
0001001F	PT_LPWSTR	Bill
0002001F	PT_LPWSTR	888-8888
00030002	PT_I2	56

Figure II.4 A look at the properties of a phone list database record.

Working with the Windows CE File System

The concepts of files and directories under Windows CE are the same as on other Windows platforms. A *file* is defined as a named collection of information. Files can contain data, as do the document files created by a word processor or notepad application. Files can also be executable programs or dynamic link libraries. The essential point is that the file is the most basic unit of storage that allows Windows CE to distinguish one set of data or information from another. A *directory* is a named group of files or other directories.

Files and directories on a traditional Windows NT or Windows 98 desktop computer have always been closely linked to the presence of permanent storage in the form of floppy disks or hard disks. Computer users are used to using utilities such as Windows Explorer to browse the contents of a physical disk on their own PC or on a PC to which they have access via a computer network.

Devices running under Windows CE do not use floppy or hard disks as storage media. Under Windows CE, files and directories are one of the object types supported by the object store. Files and directories are therefore stored persistently in device RAM, along with databases and database records and the Windows CE registry.

Despite this difference, working with files under Windows CE is very much like working with files under Windows NT or Windows 98 on a desktop PC. Although some file system features are not supported, the file system application programming interface and its semantics are much the same. Hence your understanding of working with the file system API on Windows NT or Windows 98 will go a long way in helping you learn how to work with the Windows CE file system.

AFTER COMPLETING THIS CHAPTER YOU WILL KNOW HOW TO . . .

- Create and delete files and directories**
- Open and close files and directories**
- Read and write files**
- Copy files and directories**
- Rename files and directories**
- Search for files and directories**
- Access persistent storage on flash cards**

The File System Explorer Application

The file system programming concepts covered in this chapter will be illustrated with the example of a simple File System Explorer application. The application files are found in the `\Samples\filesys` directory of the companion CD. The application that is generated by the project is called `FILESYS.EXE`.

You can certainly already use the Remote Object Viewer to browse the file system on your Windows CE device or on the Windows CE emulator. But developing an explorer application from scratch is a good way to learn how to use the file system API.

The application interface is a tree view control that displays the contents of the Windows CE file system hierarchically as shown in Figure 6.1. The folder at the top of the tree view display represents the *root directory* of the file system. All other directories and files reside somewhere under the root directory.