Powered by Microsoft® Windows® CE

CD-ROM Included

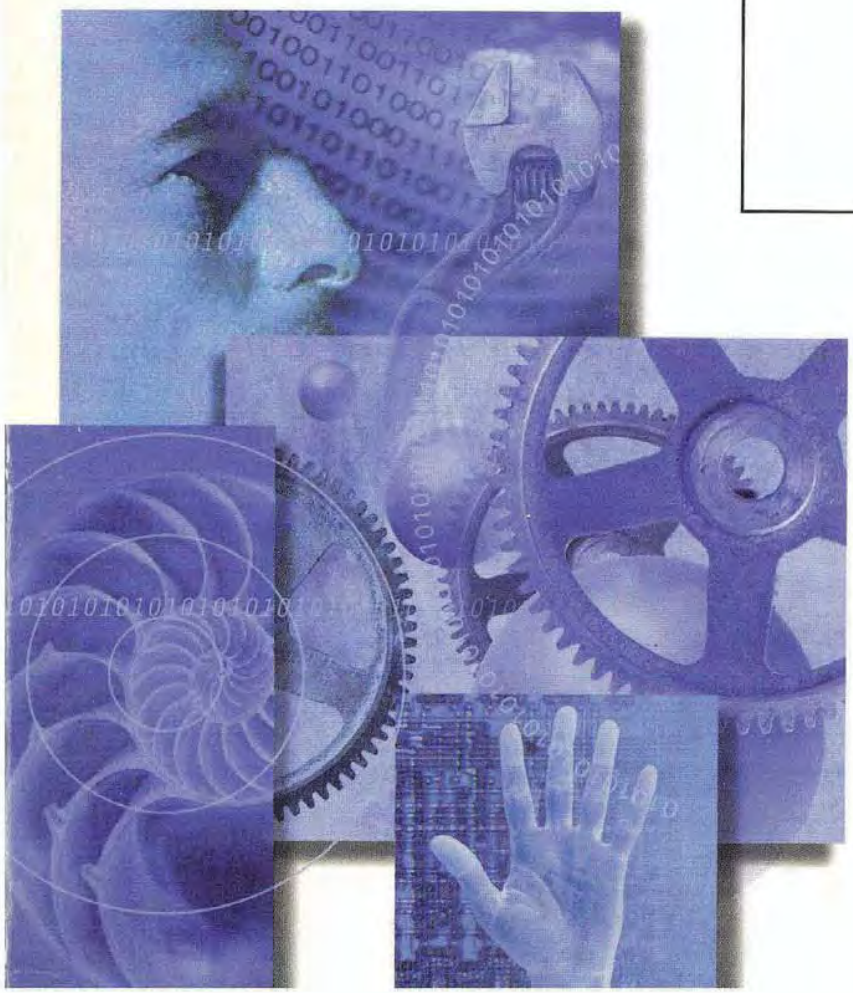
Microsoft® Programming Series

Includes Windows CE Platform SDKs

Programming Microsoft® Windows® CE

"DOUG'S CODE DEMONSTRATES A PERFECT GRASP OF WINDOWS CE—*CRAFTY* AND *ELEGANT*."

—Charles Petzold, author, *Programming Windows*



The definitive guide to programming the Windows CE API

Douglas Boling

Microsoft Press


Powered by
Microsoft®
Windows®CE
Programming Windows CE
QA76.76
063
B623
1998
Boling
Microsoft
PRESS

PROGRAMMING MICROSOFT® WINDOV

Douglas i

Microsoft

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Boling, Douglas McConnaughey, 1960–
Programming Microsoft Windows CE / Douglas McConnaughey Boling.
p. cm.
Includes index.
ISBN 1-57231-856-2
1. Microsoft Windows (Computer file) 2. Operating Systems (Computers) I. Title.
QA76.76.O63B623 1998
005.4'469--dc21 98-39279
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QMQM 3 2 1 0 9 8

Distributed in Canada by ITP Nelson, a division of Thomson Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com.

Acquisitions Editor: Eric Stroo
Project Editor: Kathleen Atkins
Technical Editor: Jim Fuchs

To Nancy Jane

Contents at a Glance

Contents

Acknowledgments

I'd heard stories from authors about the travails of writing a book. Still I was unprepared for the task. While I wrote, I learned just how much of a team effort is necessary to make a book. My name appears on the cover, but countless others were involved in its creation.

First, there is the talented team at Microsoft Press. Kathleen Atkins, the project leader and editor of this book, took my gnarled syntax and confused text and made it readable. Kathleen, thanks for your words of encouragement, your guidance, and for making this book as good as it is. The book's technical editor, Jim Fuchs, was my voice in the initial editing process. His judgement was so good that I rarely had to correct an edit for technical reasons. Many thanks also go to Cheryl Penner, the copy editor and proofreader; Elizabeth Hansford, the principal compositor; and Michael Victor, who translated my stick drawings into professional illustrations. Finally, thanks to Eric Stroo, who took a chance and signed me to write this book. Eric, the sun seems to be out now.

For technical help, I was privileged to be able to mine the golden knowledge of the Microsoft Windows CE development team. Special thanks go to Mike Thomson, who put up with endless inquiries about the technical details of Windows CE. On the rare occasions that Mike didn't have the answer, he guided me to the folks who did. Among those folks who helped were Dave Campbell, Carlos Alayo, Scott Holden, Omar Maabreh, Jeff Kelley, and Jeff Blum. While these guys did the best they could, I am, of course, responsible for any mistakes introduced into the text as I interpreted their answers.

You can't write a book of this type without hardware. My thanks go to Cheryl Balbach, Scott Nelson, and the Casio Corporation for their assistance. When other companies turned me down, Casio stepped up to the plate and provided prerelease and hard-to-find hardware necessary to test my code. Thanks, Cheryl. Call me if you need any more drop testing performed.

I also owe a debt of gratitude to the folks at Vadem Ltd. It was while working at Vadem that I was initially introduced to Windows CE and, amazingly enough, allowed to contribute to the creation of one of the machines you'll see in the introduction. Thanks to Craig Colvin, who talked me into working at Vadem and is now busy designing new and innovative Windows CE products; John Zhao, the president; and Henry Fung, CTO; as well as the managers down the line, Jim Stair and Norm Farquhar.

To all of you, thanks for allowing me to disappear as the book ran behind schedule. I'd also like to thank Edmond Ku, Scott Chastain, Ron Butterworth, Anthony Armenta, and the rest of the Clio team.

One good friend deserves special mention. Jeff Prosise started me down this path when he talked me into writing my first article in 1985. When you get past his honesty, good nature, and modesty, you're left with one incredibly smart guy, devoted to his family and friends. Thanks, Jeff, for everything.

My career as a writer started at the top, *PC Magazine*. There, I'd like to thank Michael Miller, Jake Kirchner, Bill Howard, and Gail Shaffer. Other folks no longer directly tied to the magazine but whom I still regard as part of the *PC Magazine* family are Bill Machrone, Trudy Neuhaus, and Dale Lewallen.

In addition, I thank two of the masters—Charles Petzold and Ray Duncan. These guys, along with Jeff Prosise, write the best technical books on the planet.

Thanks also to the folks at *Microsoft Systems Journal* and *Microsoft Interactive Developer,* Eric Maffei, Josh Trupin, and Gretchen Bilson. A special thanks goes to Joe Flanigan, who introduced me to some of the folks on the Windows CE team at Microsoft.

I'd also like to thank a number of musical groups that helped me through long hours in front of the PC. These include but aren't limited to the Beach Boys, the Cranberries, Alan Parson's Project, Toad the Wet Sprocket, the Eagles, and Dire Straits. Thanks also to the Southland Corporation, owners of the 7-Eleven franchise, for inventing the Big Gulp and its more potent cousins, the Super Big Gulp and the Double Gulp. Thanks also to the Coca-Cola Corporation for providing the caffeine.

On a more serious note, if there's any one person whose name also deserves to be on the cover of this book, it's Nancy Jane Hendricks Boling, my wife. Nancy endured a year of being a single parent because I spent every spare moment in front of my PC and an array of Windows CE devices writing this book. Thank you, Nancy. I'm sure I didn't say it enough over the past year. I love you. Your name isn't on the cover, but the book is dedicated to you. I must also mention two other family members—our sons Andy, 2 ½ years old, and Sam, born during the writing of Chapter 9. Andy is well on his way to becoming the best big brother a boy can be. Sam, well, he has the cutest giggle. Thanks also to Amy Sekeras for taking such good care of Andy and Sam.

Finally, I lack the words to adequately say thanks to my parents, Ronald and Jane Boling. Mom and Dad, you are simply the best parents I know, have met, or ever read about. It is my goal in life to attempt to be as good a parent to my children as you are to Rob, Chris, Jay, and me. I am truly blessed to have you as parents.

Introduction

I was introduced to Microsoft Windows CE right before it was released in the fall of 1996. A Windows programmer for many years, I was intrigued by an operating system that applied the well-known Windows API to a smaller, more power-conserving operating system. The distillation of the API for smaller machines enables tens of thousands of Windows programmers to write applications for an entirely new class of systems. The subtle differences, however, make writing Windows CE code somewhat different from writing for Windows 98 or Windows NT. It's those differences that I'll address in this book.

Just What Is Windows CE?

Windows CE is the newest, smallest, and arguably the most interesting of the Microsoft Windows operating systems. Windows CE was designed from the ground up to be a small, ROM-based operating system with a Win32 subset API. Windows CE extends the Windows API into the markets and machines that can't support the larger footprints of Windows 98 and Windows NT.

Windows 98 is a great operating system for users who need backward compatibility with DOS and Windows *2.x* and *3.x* programs. While it has shortcomings, Windows 98 succeeds amazingly well at this difficult task. Windows NT, on the other hand, is written for the enterprise. It sacrifices compatibility and size to achieve its high level of reliability and robustness.

Windows CE isn't backward compatible with MS-DOS or Windows. Nor is it an all-powerful operating system designed for enterprise computing. Instead, Windows CE is a lightweight, multithreaded operating system with an optional graphical user interface. Its strength lies in its small size, its Win32 subset API, and its multiplatform support.

Products Based on Windows CE

The first products designed for Windows CE were handheld "organizer" type devices with 480-by-240 or 640-by-240 screens and chiclets keyboards. These devices, dubbed Handheld PCs, were first introduced at Fall Comdex 96. Fall Comdex 97 saw the release of a dramatically upgraded version of the operating system, Windows CE 2.0,

with newer hardware in a familiar form—this time the box came with a 640-by-240 landscape screen and a somewhat larger keyboard.

In January 1998 at the Consumer Electronics Show, Microsoft announced two new platforms, the Palm-size PC and the Auto PC. The Palm-size PC was aimed directly at the pen-based organizer market currently dominated by the Palm Pilot. The Palm-size PC sports a portrait mode, 240-by-320 screen and uses stylus-based input. A number of Palm-size PCs are on the market today.

Figure I-1 shows both a Palm-size PC, in this case a Casio E-10, and a Handheld PC, in this case a Casio A-20.

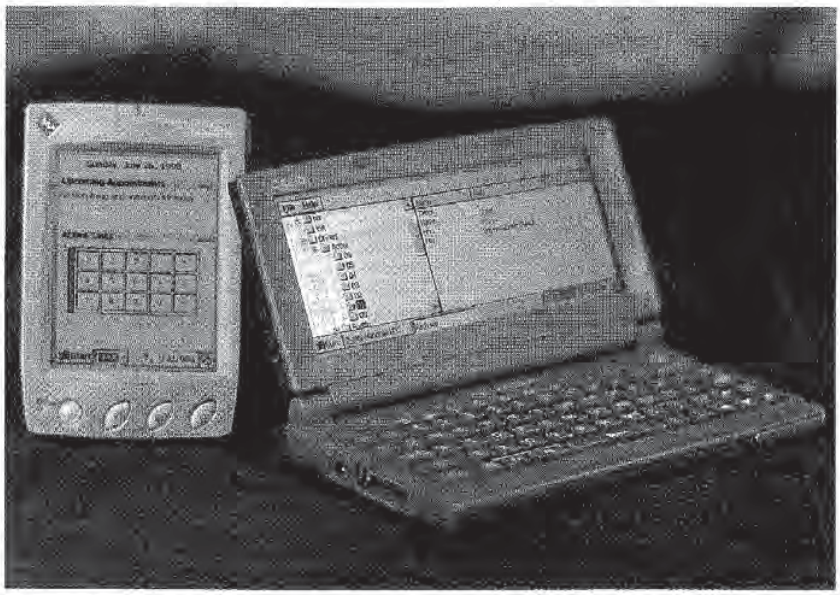


Figure I-1. *The Casio E-10 Palm-size PC and the Casio A-20 Handheld PC.*

Just as this book is being released, Microsoft has introduced the Handheld PC Professional, which is a greatly enhanced H/PC with new applications and which uses the latest version of the operating system, Windows CE 2.11.[1] This device brings the compact nature of Windows CE to devices of laptop size. The advantages of applying Windows CE to a laptop device are many. First, the battery life of a Handheld PC Pro is at least 10 hours, far better than the 2-to 3-hour average of a PC-compatible laptop. Second, the size and weight of the Windows CE devices are far more user friendly, with systems as thin as 1 inch weighing less than 3 pounds. Even with the diminutive size, a Handheld PC Pro still sports a large VGA screen and a keyboard that a normal human can use. The Vadem Clio Handheld PC Pro, shown in Figure I-2, is an example of how Windows CE is being used in newer platforms. The system

1. Windows CE 2.11 is Windows CE 2.10 with a few minor changes.

can be used as a standard laptop or "flipped" into a tablet-mode device. This device is just one example of how Windows CE is expanding into new system types.

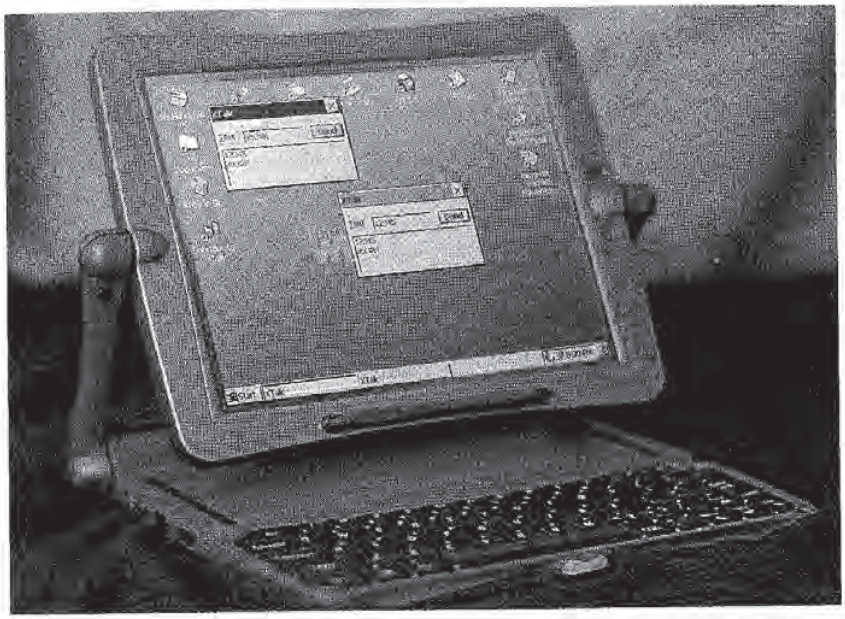


Figure I-2. *The Vadem Clio Handheld PC Pro.*

I refer to the Handheld PC Pro throughout this book under its operating system version, Windows CE 2.1, because the platform name, Handheld PC Pro, was determined very late in the process. I knew of, and in fact, had a hand in the development of a Handheld PC Pro under its code name Jupiter. However, you can't use code names in a book, so its operating system version had to suffice.

Other platforms—Auto PC, Web TV set-top boxes, and embedded platforms designed for specific tasks—are also appearing or will appear in the coming months. What's amazing about Windows CE is that the flexibility of the operating system allows it to be used in all these diverse designs while all the time retaining the same basic, well-known Win32 API.

WHY YOU SHOULD READ THIS BOOK

Programming Microsoft Windows CE is written for anyone who will be writing applications for Windows CE. Both the embedded systems programmer using Windows CE for a specific application and the Windows programmer interested in porting an existing Windows application or writing an entirely new one can use the information in this book to make their tasks easier.

The embedded systems programmer, who might not be as familiar with the Win32 API as the Windows programmer, can read the first section of the book to

become familiar with Windows programming. While this section isn't the comprehensive tutorial that can be found in books such as *Programming Windows* by Charles Petzold, it does provide a base that will carry the reader through the other chapters in the book. It also can help the embedded systems programmer develop fairly complex and quite useful Windows CE programs.

The experienced Windows programmer can use the book to learn about the differences among the Win32 APIs used by Windows CE, Windows NT, and Windows 98. Programmers who are familiar with Win32 programming recognize subtle differences between the Windows 98 and Windows NT APIs. The differences between Windows CE and its two cousins are even greater. The small footprint of Windows CE means that many of the overlapping APIs in the Win32 model aren't supported. Some sections of the Win32 API aren't supported at all. On the other hand, because of its unique setting, Windows CE extends the Win32 API in a number of areas that are covered in this text.

The method used by *Programming Windows CE* is to teach by example. I wrote numerous Windows CE example programs specifically for this book. The source for each of these examples is printed in the text. Both the source and the final compiled programs for a number of the processors supported by Windows CE are also provided on the accompanying CD.

The examples in this book are all written directly to the API, the so-called "Petzold" method of programming. Since the goal of this book is to teach you how to write programs for Windows CE, the examples avoid using a class library such as MFC, which obfuscates the unique nature of writing applications for Windows CE. Some people would say that the availability of MFC on Windows CE eliminates the need for direct knowledge of the Windows CE API. I believe the opposite is true. Knowledge of the Windows CE API enables more efficient use of MFC. I also believe that truly knowing the operating system also dramatically simplifies the debugging of applications.

WHAT ABOUT MFC?

The simple fact is that Windows CE systems aren't the best platform for a general-purpose class library like MFC. The slower processors and the significantly lower memory capacity of Windows CE devices make using MFC problematic. Most Windows CE systems don't include the MFC library in their ROM. This means that the MFC and OLE32 DLLs required by MFC applications must be downloaded into the systems. The first versions of the Palm-size PCs don't even support MFC.

That said, there's a place for MFC on Windows CE devices. One such place might be if you're designing a custom application for a system you know will have the MFC and OLE32 DLLs in ROM. For those specific applications, you might want to use MFC, but only if you know the target environment and have configured the system with the proper amount of RAM to do the job.

WINDOWS CE DEVELOPMENT TOOLS

This book is written with the assumption that the reader knows C and is at least familiar with Microsoft Windows. All code development was done with Microsoft Visual C++ 5.0 and Windows CE Visual C++ for Windows CE under Windows NT 4.0.

To compile the example programs in this book, you need Microsoft Visual C++ 5.0, which is part of the integrated development environment (IDE), DevStudio, running on a standard IBM-compatible PC. You also need Microsoft Visual C++ for Windows CE, which isn't a stand-alone product. It's an add-in to Visual C++ 5.0 that incorporates components to the compiler that produce code for the different CPUs supported by Windows CE. Visual C++ for Windows CE isn't currently available through standard retail channels, but information on ordering it directly from Microsoft can be found on the Microsoft Web site. Finally, you need one of the platform SDKs for Windows CE. These SDKs provide the custom include files for each of the Windows CE platforms. These platform SDKs are available for free on the Microsoft Web site. As a convenience, I've also included the platform SDKs available at the time of the writing of this book on the accompanying CD.

While not absolutely required for developing applications for Windows CE, Windows NT 4.0 is strongly recommended for the development environment. It's possible to compile and download Windows CE programs under Windows 98, but many of the features of the integrated development environment (IDE), such as Windows CE emulation and remote debugging, aren't supported.

Visual C++ for Windows CE won't change the outward appearance of Visual C++, with the exception of a few new tools listed under the tools menu. Nor will the installation of Visual C++ for Windows CE prevent you from developing applications for other Windows operating systems. The installation of Visual C++ for Windows CE will result in new Windows CE targets such as WCE MIPS and WCE SH and WCE x86Em being added to the platforms listing when you're creating a new Win32 application. Also, a Windows CE MFC AppWizard will be added to the new projects listing to assist in creating MFC programs for Windows CE.

TARGET SYSTEMS

You don't need to have a Windows CE target device to experience the sample programs provided by this book. The various platform SDKs come with a Windows CE emulator that lets you perform basic testing of a Windows CE program under Windows NT. This emulator comes in handy when you want to perform initial debugging to ensure that the program starts, creates the proper windows, reacts to menu selections, and so on. However, the emulator has some limitations and there simply is no replacement for having a target Windows CE system to perform final debugging and testing for applications.

You should consider a number of factors when deciding what Windows CE hardware to use for testing. First, if the application is to be a commercial product, you should buy at least one system for each type of target CPU. You need to test against all of the target CPUs because, while the source code will probably be identical, the resulting executable will be different in size and so will the memory allocation footprint for each target CPU.

Most applications will also be written specifically for the Handheld PC or Palm-size PC, not both. Although the base operating system for both the Handheld PC and Palm-size PC is Windows CE, the hardware underneath is vastly different. The strict memory constraints of the Palm-size PC, as well as its much smaller screen, its different orientation, and its lack of a keyboard, force compromises that aren't acceptable on a Handheld PC or its larger relative, the Handheld PC Pro. Other constraints on Palm-size PC systems, such as the lack of printing and TrueType support, differentiate its environment from the Handheld PC's.

In this book, I demonstrate programs that can run on the Handheld PC, Handheld PC Pro, or Palm-size PC. The goal is to allow the lessons to be applied to all platforms. For some examples, however, the different screen dimensions mean that the example will run better on one particular system. I point out the differences and the reasons they exist. For example, some controls might exist on only one platform or the other. The shells for the two platforms—Handheld or Palm-size—are also different and need separate coverage. Finally, a small set of features in Windows CE are simply not supported on the smaller Palm-size PC platform.

WHAT'S ON THE CD

The accompanying CD contains the source code for all the examples in the book. I've also provided project files for Microsoft DevStudio so that you can open preconfigured projects. Unless otherwise noted, the examples are Windows CE 2.0 compatible so that they can run on most Windows CE systems available today. Chapter 13, "Shell Programming—Part 2" contains examples that are compiled for Windows CE 2.01, so they won't run on current Handheld PCs. There are some examples, such as the console applications in Chapter 12, that are specific to the Handheld PC Pro and other devices running Windows CE 2.10.

When you build for a specific platform, remember that it might not be backward compatible with earlier versions of Windows CE. For example, Microsoft moved some of the C library support from statically linked libraries in Windows CE 2.0 into the operating system for Windows CE 2.01, the Palm-size PC release. This reduces the size of an executable, but prevents code built for the Palm-size PC from running on a Handheld PC running Windows CE 2.0. You can, however, compile code for a Handheld PC running Windows CE 2.0 and have it run on a Palm-size PC.

In addition to the examples, the CD contains a number of folders of interest to the Windows CE programmer. I've included the platform SDKs for the Handheld PC as well as for the Palm-size PC. Unfortunately, the Handheld PC Pro SDK wasn't available in time for this release. Like the other platform SDKs, that one is available for free on the Microsoft Web site. Check out the readme file on the CD for late-breaking information about what else is included on the CD.

OTHER SOURCES

While I have attempted to make *Programming Microsoft Windows CE* a one-stop shop for Windows CE programming, no one book can cover everything. A nice complement to this book is *Inside Windows CE* by John Murray. It documents the "oral history" of Windows CE. Knowing this kind of information is crucial to understanding just why Windows CE is designed the way it is. Once you know the why, it's easy to extrapolate the what, when trying to solve problems. Murray's book is great, not just because of the information you'll learn about Windows CE but also because it's an entertaining read.

For learning more about Windows programming in general, I suggest the classic text *Programming Windows* by Charles Petzold. This is, by far, the best book for learning Windows programming. Charles presents examples that show how to tackle difficult but common Windows problems. For learning more about the Win32 kernel API, I suggest Jeff Richter's *Advanced Windows*. Jeff covers the techniques of process, thread, and memory management down to the most minute detail. For learning more about MFC programming, there's no better text than Jeff Prosise's *Programming Windows 95 with MFC*. This book is the "Petzold" of MFC programming and simply a required read for MFC programmers.

FEEDBACK

While I have striven to make the information in this book as accurate as possible, you'll undoubtedly find errors. If you find a problem with the text or just have ideas about how to make the next version of the book better, please drop me a note at *CEBook@DelValle.com*. I can't promise you that I'll answer all your notes, but I will read every one.

Doug Boling
Tahoe City, California
August 1998

Part I

WINDOWS PROGRAMMING BASICS

Chapter 1

Hello Windows CE

From Kernighan and Ritchie to Petzold and on to Prosise, programming books traditionally start with a "hello, world" program. It's a logical place to begin. Every program has a basic underlying structure that, when not obscured by some complex task it was designed to perform, can be analyzed to reveal the foundation shared by all programs running on its operating system.

In this programming book, the "hello, world" chapter covers the details of setting up and using the programming environment. The environment for developing Microsoft Windows CE applications is somewhat different from that for developing standard Microsoft Windows applications because Windows CE programs are written on PCs running Microsoft Windows NT and debugged mainly on separate, Windows CE–based target devices.

While experienced Windows programmers might be tempted to skip this chapter and move on to meatier subjects, I suggest that they—you—at least skim the chapter to note the differences between a standard Windows program and a Windows CE program. A number of subtle and significant differences in both the development process and the basic program skeleton for Windows CE applications are covered in this first chapter.

WHAT IS DIFFERENT ABOUT WINDOWS CE?

Windows CE has a number of unique characteristics that make it different from other Windows platforms. First of all, the systems running Windows CE are most likely not using an Intel x86 compatible microprocessor. Instead, a short list of supported CPUs run Windows CE. Fortunately, the development environment isolates the programmer from almost all of the differences among the various CPUs.

Nor can a Windows CE program be assured of a screen or a keyboard. Some Windows CE devices have a 240-by-320-pixel portrait-style screen while others might have screens with more traditional landscape orientations in 480-by-240, 640-by-240, or 640-by-480-pixel resolution. An embedded device might not have a display at all. The target devices might not support color. And, instead of a mouse, most Windows CE devices have a touch screen. On a touch-screen device, left mouse button clicks are achieved by means of a tap on the screen, but no obvious method exists for delivering right mouse button clicks. To give you some method of delivering a right click, the Windows CE convention is to hold down the Alt key while tapping. It's up to the Windows CE application to interpret this sequence as a right mouse click.

Fewer Resources in Windows CE Devices

The resources of the target devices vary radically across systems that run Windows CE. When writing a standard Windows program, the programmer can make a number of assumptions about the target device, almost always an IBM-compatible PC. The target device will have a hard disk for mass storage and a virtual memory system that uses the hard disk as a swap device to emulate an almost unlimited amount of (virtual) RAM. The programmer knows that the user has a keyboard, a two-button mouse, and a monitor that these days almost assuredly supports 256 colors and a screen resolution of at least 640 by 480 pixels.

Windows CE programs run on devices that almost never have hard disks for mass storage. The absence of a hard disk means more than just not having a place to store large files. Without a hard disk, virtual RAM can't be created by swapping data to the disk. So Windows CE programs are almost always run in a low-memory environment. Memory allocations can, and often do, fail because of the lack of resources. Windows CE might terminate a program automatically when free memory reaches a critically low level. This RAM limitation has a surprisingly large impact on Windows CE programs and is one of the main difficulties involved in porting existing Windows applications to Windows CE.

Unicode

One characteristic that a programmer can count on when writing Windows CE applications is Unicode. Unicode is a standard for representing a character as a 16-bit value as opposed to the ASCII standard of encoding a character into a single 8-bit value. Unicode allows for fairly simple porting of programs to different international markets because all the world's known characters can be represented in one of the 65,536 available Unicode values. Dealing with Unicode is relatively painless as long as you avoid the dual assumptions made by most programmers that strings are represented in ASCII and that characters are stored in single bytes.

A consequence of a program using Unicode is that with each character taking up two bytes instead of one, strings are now twice as long. A programmer must be careful making assumptions about buffer length and string length. No longer should you assume that a 260-byte buffer can hold 259 characters and a terminating zero. Instead of the standard char data type, you should use the TCHAR data type. TCHAR is defined to be char for Microsoft Windows 95 and Microsoft Windows 98 development and unsigned short for Unicode-enabled applications for Microsoft Windows NT and Windows CE development. These types of definitions allow source-level compatibility across ASCII- and Unicode-based operating systems.

New Controls

Windows CE includes a number of new Windows controls designed for specific environments. New controls include the command bar that provides menu- and toolbar-like functions all on one space-saving line, critical on the smaller screens of Windows CE devices. The date and time picker control and calendar control assist calendar and organizer applications suitable for handheld devices, such as the Handheld PC (H/PC) and the Palm-size PC. Other standard Windows controls have reduced function, reflecting the compact nature of Windows CE hardware-specific OS configurations.

Another aspect of Windows CE programming to be aware of is that Windows CE can be broken up and reconfigured by Microsoft or by OEMs so that it can be better adapted to a target market or device. Windows programmers usually just check the version of Windows to see whether it is from the Microsoft Windows 3.1, 95, or 98 line or Windows NT line; by knowing the version they can determine what API functions are available to them. Windows CE, however, has had four variations already in its first two years of existence: the Handheld PC, the Palm-size PC, the Handheld PC Pro, and the Auto PC. A number of new platforms are on their way, with much in common but also with many differences among them. Programmers need to understand the target platform and to have their programs check what functions are available on that particular platform before trying to use a set of functions that might not be supported on that device.

Finally, because Windows CE is so much smaller than Windows 98 or Windows NT, it simply can't support all the function calls that its larger cousins do. While you'd expect an operating system that didn't support printing, such as Windows CE on the original Palm-size PC, not to have any calls to printing functions, Windows CE also removes some redundant functions supported by its larger cousins. If Windows CE doesn't support your favorite function, a different function or set of functions will probably work just as well. Sometimes Windows CE programming seems to consist mainly of figuring out ways to implement a feature using the sparse API of Windows CE. If 2000 functions can be called sparse.

IT'S STILL WINDOWS PROGRAMMING

While differences between Windows CE and the other versions of Windows do exist, they shouldn't be overstated. Programming a Windows CE application is programming a Windows application. It has the same message loop, the same windows, and for the most part, the same resources and the same controls. The differences don't hide the similarities. For those who aren't familiar with Windows programming, here's a short introduction.

Windows programming is far different from MS-DOS–based or Unix-based programming. An MS-DOS or Unix program uses *getc*- and *putc*-style functions to read characters from the keyboard and write them to the screen whenever the program needs to do so. This is the classic "pull" style used by MS-DOS and Unix programs, which are procedural. A Windows program, on the other hand, uses a "push" model, in which the program must be written to react to notifications from the operating system that a key has been pressed or a command has been received to repaint the screen.

Windows applications don't ask for input from the operating system; the operating system notifies the application that input has occurred. The operating system achieves these notifications by sending *messages* to an application window. All windows are specific instances of a *window class*. Before we go any further, let's be sure we understand these terms.

The Window Class

A window is a region on the screen, rectangular in all but the most contrived of cases, that has a few basic parameters, such as position—x, y, and z (a window is over or under other windows on the screen)—visibility, and hierarchy—the window fits into a parent/child window relationship on the system *desktop*, which also happens to be a window.

Every window created is a specific instance of a window class. A window class is a template that defines a number of attributes common to all the windows of that class. In other words, windows of the same class have the same attributes. The most important of the shared attributes is the *window procedure*.

The window procedure

The behavior of all windows belonging to a class is defined by the code in its window procedure for that class. The window procedure handles all notifications and requests sent to the window. These notifications are sent either by the operating system, indicating that an event has occurred to which the window must respond, or by other windows querying the window for information.

These notifications are sent in the form of messages. A message is nothing more than a call being made to a window procedure, with a parameter indicating the nature of the notification or request. Messages are sent for events such as a window being moved

or resized or to indicate a key press. The values used to indicate messages are defined by Windows. Applications use predefined constants, such as WM_CREATE or WM_MOVE, when referring to messages. Since hundreds of messages can be sent, Windows conveniently provides a default processing function to which a message can be passed when no special processing is necessary by the window class for that message.

The life of a message

Stepping back for a moment, let's look at how Windows coordinates all of the messages going to all of the windows in a system. Windows monitors all the sources of input to the system, such as the keyboard, mouse, touch screen, and any other hardware that could produce an event that might interest a window. As an event occurs, a message is composed and directed to a specific window. Instead of Windows directly calling the window procedure, the system imposes an intermediate step. The message is placed in a message queue for the application that owns the window. When the application is prepared to receive the message, it pulls it out of the queue and tells Windows to dispatch that message to the proper window in the application.

If it seems to you that a number of indirections are involved in that process, you're right. Let's break it down.

1. An event occurs, so a message is composed by Windows and placed in a message queue for the application that owns the destination window. In Windows CE, as in Windows 95 and Windows NT, each application has its own unique message queue[1]. (This is a break from Windows 3.1 and earlier versions of Windows, where there was only one, systemwide message queue.) Events can occur, and therefore messages can be composed, faster than an application can process them. The queue allows an application to process messages at its own rate, although the application had better be responsive or the user will see a jerkiness in the application. The message queue also allows Windows to set a notification in motion and continue with other tasks without having to be limited by the responsiveness of the application to which the message is being sent.

2. The application removes the message from its message queue and calls Windows back to dispatch the message. While it may seem strange that the application gets a message from the queue and then simply calls Windows back to process the message, there's a method to this madness. Having the application pull the message from the queue allows it to preprocess the message before it asks Windows to dispatch the message to

1. Technically, each thread in a Windows CE application can have a message queue. I'll talk about threads later in the book.

the appropriate window. In a number of cases, the application might call different functions in Windows to process specific kinds of messages.

3. Windows dispatches the message; that is, it calls the appropriate window procedure. Instead of having the application directly call the window procedure, another level of indirection occurs, allowing Windows to coordinate the call to the window procedure with other events in the system. The message doesn't stand in another queue at this point, but Windows might need to make some preparations before calling the window procedure. In any case, the scheme relieves the application of the obligation to determine the proper destination window—Windows does this instead.

4. The window procedure processes the message. All window procedures have the same calling parameters: the handle of the specific window instance being called, the message, and two generic parameters that contain data specific to each message type. The window handle differentiates each instance of a window for the window procedure. The message parameter, of course, indicates the event that the window must react to. The two generic parameters contain data specific to the message being sent. For example, in a WM_MOVE message indicating that the window is about to be moved, one of the generic parameters points to a structure containing the new coordinates of the window.

Your First Program

Enough small talk. It's time to jump into the first example, Hello Windows CE. While the entire program files for this and all examples in the book are available on the companion CD-ROM, I suggest that, at least in this one case, you avoid simply loading the project file from the CD and instead type in the entire example by hand. By performing this somewhat tedious task, you'll see the differences in the development process as well as the subtle program differences between standard Win32 programs and Windows CE programs. Figure 1-1 contains the complete source for HelloCE, my version of a hello, world program.

HelloCE.h

```
//------------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
```

Figure 1-1. *The HelloCE program.*

```
//=====================================================================
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

//---------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                             // Structure associates
    UINT Code;                                  // messages
                                                // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                              // Structure associates
    UINT Code;                                  // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);     // function
};

//---------------------------------------------------------------------
// Generic defines used by application
#define  IDC_CMDBAR 1                           // Command bar ID

//---------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoHibernateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoActivateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);
```

HelloCE.c

```
//=====================================================================
// HelloCE - A simple application for Windows CE
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
```

(continued)

Figure 1-1. *continued*

```
//
//=============================================================================
#include <windows.h>                    // For all that Windows stuff
#include <commctrl.h>                   // command bar includes
#include "helloce.h"                    // Program-specific stuff

//-----------------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT("HelloCE");
HINSTANCE hInst;                        // Program instance handle

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_PAINT, DoPaintMain,
    WM_HIBERNATE, DoHibernateMain,
    WM_ACTIVATE, DoActivateMain,
    WM_DESTROY, DoDestroyMain,
};

//=============================================================================
//
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
    HWND hwndMain;

    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
```

```
}
//--------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application main window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = MainWndProc;                   // Callback function
    wc.cbClsExtra = 0;                              // Extra class data
    wc.cbWndExtra = 0;                              // Extra window data
    wc.hInstance = hInstance;                       // Owner handle
    wc.hIcon = NULL;                                // Application icon
    wc.hCursor = NULL;                              // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                        // Menu name
    wc.lpszClassName = szAppName;                   // Window class name

    if (RegisterClass (&wc) == 0) return 1;

    return 0;
}
//--------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow) {
    HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    hWnd = CreateWindow (szAppName,                 // Window class
                         TEXT("Hello"),             // Window title
                         WS_VISIBLE,                // Style flags
                         CW_USEDEFAULT,             // x position
                         CW_USEDEFAULT,             // y position
                         CW_USEDEFAULT,             // Initial width
                         CW_USEDEFAULT,             // Initial height
                         NULL,                      // Parent
                         NULL,                      // Menu, must be null
                         hInstance,                 // Application instance
                         NULL);                     // Pointer to create
                                                    // parameters
```

(continued)

Figure 1-1. *continued*

```
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//-----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {

    return nDefRC;
}
//=======================================================================
// Message handling procedures for main window
//

//-----------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//-----------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB;

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);
```

```
    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    return 0;
}
//------------------------------------------------------------------------
// DoPaintMain - Process WM_PAINT message for window.
//
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect;
    HDC hdc;

    // Adjust the size of the client rectangle to take into account
    // the command bar height.
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    hdc = BeginPaint (hWnd, &ps);
    DrawText (hdc, TEXT ("Hello Windows CE!"), -1, &rect,
              DT_CENTER | DT_VCENTER | DT_SINGLELINE);

    EndPaint (hWnd, &ps);
    return 0;
}
//------------------------------------------------------------------------
// DoHibernateMain - Process WM_HIBERNATE message for window.
//
LRESULT DoHibernateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {

    // If not the active window, nuke the command bar to save memory.
    if (GetActiveWindow() != hWnd)
        CommandBar_Destroy (GetDlgItem (hWnd, IDC_CMDBAR));

    return 0;
}
//------------------------------------------------------------------------
// DoActivateMain - Process WM_ACTIVATE message for window.
//
LRESULT DoActivateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    HWND hwndCB;
```

(continued)

Figure 1-1. *continued*

```
    // If activating and no command bar, create it.
    if ((LOWORD (wParam) != WA_INACTIVE) &&
        (GetDlgItem (hWnd, IDC_CMDBAR) == 0)) {

        // Create a command bar.
        hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

        // Add exit button to command bar.
        CommandBar_AddAdornments (hwndCB, 0, 0);
    }
    return 0;
}
//------------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
```

If you look over the source code for HelloCE, you'll see the standard boilerplate for all programs in this book. I'll talk at greater length about a few of the characteristics, such as Hungarian notation and the somewhat different method I use to construct my window procedures later, in their own sections, but at this point I'll make just a few observations about them.

Just after the comments, you see the include of windows.h. You can find this file in all Windows programs; it lists the definitions for the special variable types and function defines needed for a typical program. Windows.h and the include files it contains make an interesting read because the basics for all windows programs come from the functions, typedefs, and structures defined there. The include of commctrl.h provides, among other things, the definitions for the command bar functions that are part of almost all Windows CE programs. Finally, the include of HelloCE.h gives you the boilerplate definitions and function prototypes for this specific program.

A few variables defined globally follow the defines and includes. I know plenty of good arguments why no global variables should appear in a program, but I use them as a convenience that shortens and clarifies the example programs in the book. Each program defines an *szAppName* Unicode string to be used in various places in that program. I also use the *hInst* variable a number of places and I'll mention it when I cover the *InitApp* procedure. The final global structure is a list of messages along

with associated procedures to process the messages. This structure is used by the window procedure to associate messages with the procedure that handles them. Now, on to a few other characteristics common to all the programs in this book.

Hungarian Notation

A tradition, and a good one, of almost all Windows programs since Charles Petzold wrote *Programming Windows* is Hungarian notation. This programming style, developed years ago by Charles Simonyi at Microsoft, prefixes all variables in the program usually with one or two letters indicating the variable type. For example, a string array called *Name* would instead be called *szName*, with the *sz* prefix indicating that the variable type is a zero-terminated string. The value of Hungarian notation is the dramatic improvement in readability of the source code. Another programmer, or you after not looking at a piece of code for a while, won't have to look repeatedly at a variable's declaration to determine its type. The following are typical Hungarian prefixes for variables:

Variable Type	***Hungarian Prefix***
Integer	*i* or *n*
Word (16-bit)	*w* or *s*
Double word (32-bit unsigned)	*dw*
Long (32-bit signed)	*l*
Char	*c*
String	*sz*
Pointer	*p*
Long pointer	*lp*
Handle	*h*
Window handle	*hwnd*
Struct size	*cb*

You can see a few vestiges of the early days of Windows. The *lp*, or long pointer, designation refers to the days when, in the Intel 16-bit programming model, pointers were either short (a 16-bit offset) or long (a segment plus an offset). Other prefixes are formed from the abbreviation of the type. For example, a handle to a brush is typically specified as *hbr*. Prefixes can be combined, as in *lpsz*, which designates a long pointer to a zero-terminated string. Most of the structures defined in the Windows API use Hungarian notation in their field names. I use this notation as well throughout the book, and I encourage you to use this notation in your programs.

My Programming Style

One criticism of the typical SDK style of Windows programming has always been the huge *switch* statement in the window procedure. The *switch* statement parses the message to the window procedure so that each message can be handled independently. This standard structure has the one great advantage of enforcing a similar structure across almost all Windows applications, making it much easier for one programmer to understand the workings of another programmer's code. The disadvantage is that all the variables for the entire window procedure typically appear jumbled at the top of the procedure.

Over the years, I've developed a different style for my Windows programs. The idea is to break up the *WinMain* and *WinProc* procedures into manageable units that can be easily understood and easily transferred to other Windows programs. *WinMain* is broken up into procedures that perform application initialization, instance initialization, and instance termination. Also in *WinMain* is the ubiquitous message loop that's the core of all Windows programs.

I break the window procedure into individual procedures, with each handling a specific message. What remains of the window procedure itself is a fragment of code that simply looks up the message that's being passed to see whether a procedure has been written to handle that message. If so, that procedure is called. If not, the message is passed to the default window procedure.

This structure divides the handling of messages into individual blocks that can be more easily understood. Also, with greater isolation of one message-handling code fragment from another, you can more easily transfer the code that handles a specific message from one program to the next. I first saw this structure described a number of years ago by Ray Duncan in one of his old "Power Programming" columns in *PC Magazine*. Ray is one of the legends in the field of MS-DOS and OS/2 programming. I've since modified the design a bit to fit my needs, but Ray should get the credit for this program structure.

Building HelloCE

To create HelloCE from scratch on your system, start Microsoft Visual C++ and create a new Win32 application. The first change from standard Win32 programming becomes evident when you create the new project. You'll have the opportunity to select a new platform specific to Windows CE, as shown in Figure 1-2. These platforms have a WCE prefix followed by the target CPU. For example, selecting Win32 (WCE MIPS) enables compiling to a Windows CE platform with a MIPS CPU. No matter what target device you have, be sure to check the WCE x86em target. This allows you to run the sample program in the emulator under Windows NT.

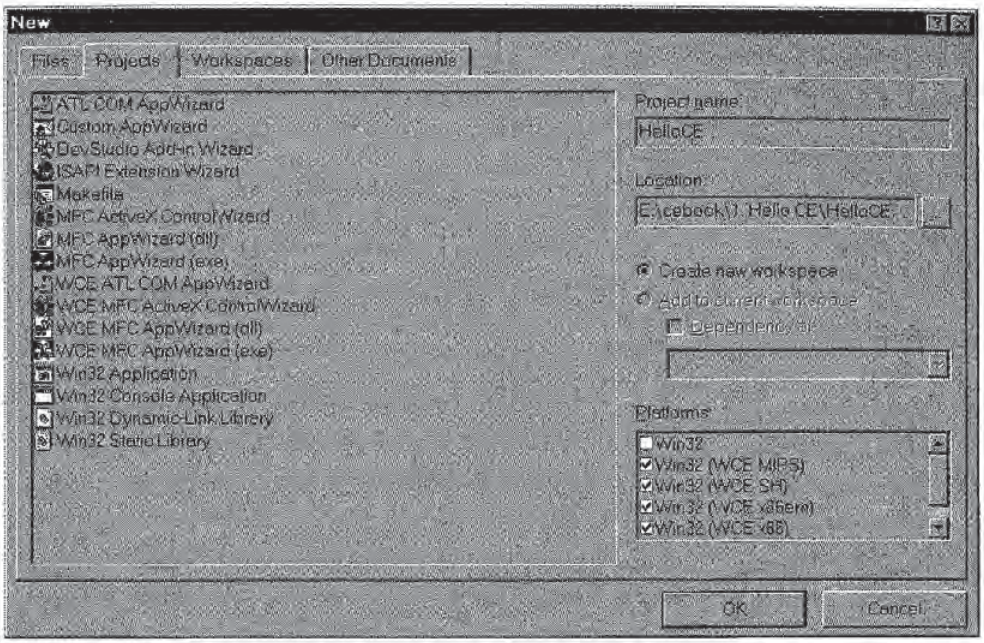


Figure 1-2. *The Platforms list box allows Visual C++ 5.0 to target Windows CE platforms.*

After you have created the proper source files for HelloCE or copied them from the CD, select the target Win32 (WCE x86em) Debug and then build the program. This step compiles the source and, assuming you have no compile errors, automatically launches the emulator and inserts the EXE into the emulator file system; you can then launch HelloCE. If you're running Windows 95 or Windows 98, the system displays an error message because the emulator runs only under Windows NT.

If you have a Windows CE system available, such as an H/PC, attach the H/PC to the PC the same way you would to sync the contents of the H/PC with the PC. Open the Mobile Devices folder and establish a connection between the H/PC and the PC. While it's not strictly necessary to have the Mobile Devices connection to your Windows CE device running because the SDK tools inside Visual C++ are supposed to make this connection automatically, I've found that having it running makes for a more stable connection between the development environment and the Windows CE system.

Once the link between the PC and the Windows CE device is up and running, switch back to Visual C++, select the compile target appropriate for the target device (for example, Win32 [WCE SH] Debug for an HP 360 HPC), and rebuild. As in the

case of building for the emulator, if there are no errors Visual C++ automatically downloads the compiled program to the remote device. The program is placed in the root directory of the object store.

Running the program

To run HelloCE on an H/PC, simply click on the My Handheld PC icon to bring up the files in the root directory. At that point, a double-tap on the application's icon launches the program.

Running the program on a Palm-size PC is somewhat more complex. Because the Palm-size PC doesn't come with an Explorer program that allows users to browse through the files on the system, you can't launch HelloCE without a bit of preparatory work. You can launch the program from Visual C++ by selecting Execute from the Build menu. Or you can have Visual C++ automatically copy the executable file into the \windows\start menu\programs directory of the Palm-size PC. This automatically places the program in the Programs submenu under the Start menu. You can tell Visual C++ to automatically copy the file by setting the remote target path in the Debug tab of the Project Settings dialog box. Figure 1-3 shows this dialog box. When you've set this path, you can easily start the program by selecting it in the Start menu.

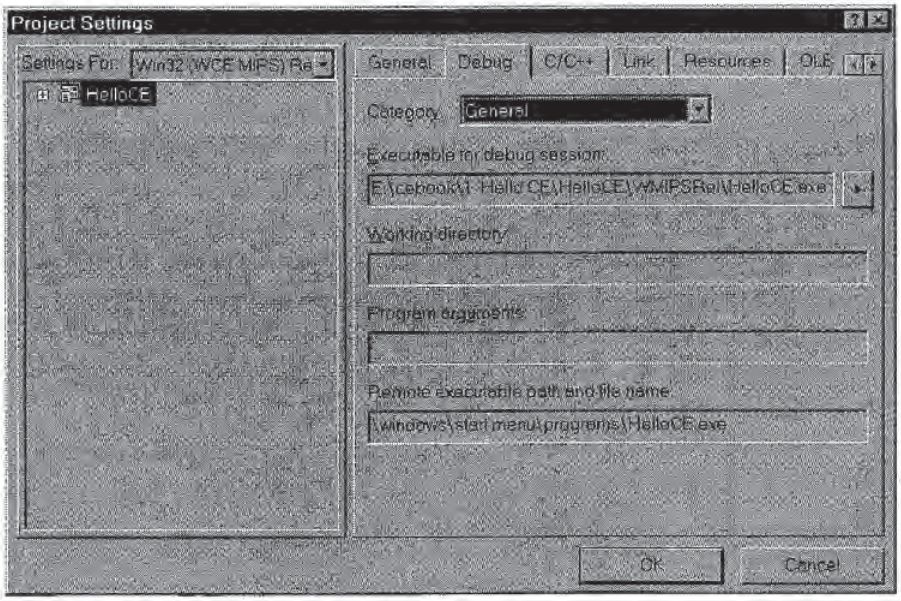


Figure 1-3. *The Project Settings dialog box in Visual C++ with the Debug tab selected.*

One "gotcha" to look out for here. If you're debugging and recompiling the program, it can't be downloaded again if an earlier version of the program is still running on the target system. That is, make sure HelloCE isn't running on the remote system when you start a new build in Visual C++ or the auto download part of the compile process will fail. If this happens, close the application and choose

the Update Remote File menu command in Visual C++ to download the newly compiled file.

Palm-size PC users will notice that unlike almost all Palm-size PC programs, HelloCE has a Close button in the upper right corner of the window. By convention, the user doesn't close Palm-size PC applications; they're closed only when the system needs more memory space. The lack of a Close button in Palm-size PC applications is only a user interface guideline, not a lack of function of the version of Windows CE in the Palm-size PC. For development, you might want to keep a Close button in your application because you'll need to close the program to download a new version. You can then remove the Close button before you ship your application.

If you don't have access to an H/PC or if you want to check out Windows CE programming without the hassle of connecting to a remote device, the emulation environment is a great place to start. It's the perfect place for stepping though the code just as you would were you debugging a standard PC-based Windows program. You can set breakpoints and step though code running on a remote system, but the slow nature of the serial link as well as the difficulty in single-stepping a program on the remote system make debugging on the emulator much less painful. On the other hand, debugging on the remote system is the only way to truly test your program. While the emulator is a good first step in the debug process, nothing replaces testing on the target system.

The code

Now that you have the program up and running either in the emulator or on a Windows CE device, it's time to look at the code itself. The program entry point, *WinMain*, is the same place any Windows program begins. Under Windows CE, however, some of the parameters for *WinMain* have limits to the allowable values. *WinMain* is defined as the following:

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

The first of the four parameters passed, *hInstance*, identifies the specific instance of the program to other applications and to Windows API functions that need to identify the EXE. The *hPrevInstance* parameter is left over from the old Win16 API (Windows 3.1 and earlier). In those versions of Windows, the *hPrevInstance* parameter was nonzero if there were any other instances of the program currently running. In all Win32 operating systems, including Windows CE, the *hPrevInstance* is always 0 and can be ignored.

The *lCmdLine* parameter points to a Unicode string that contains the text of the command line. Applications launched from Microsoft Windows Explorer usually have no command line parameters. But in some instances, such as when the system automatically launches a program, the system includes a command line

parameter to indicate why the program was started. The *lCmdLine* parameter provides us with one of the first instances in which Windows CE differs from Windows NT or Windows 98. Under Windows CE, the command line string is a Unicode string. In Windows NT and Windows 98, the string is always ASCII.

The final parameter, *nShowCmd*, specifies the initial state of the program's main window. In a standard Win32 program, this parameter might specify that the window be initially displayed as an icon (SW_SHOWMINIMIZE), maximized (SW_SHOWMAXIMIZED) to cover the entire desktop, or normal (SW_RESTORE), indicating that the window is placed on the screen in the standard resizable state. Other values specify that the initial state of the window should be invisible to the user or that the window be visible but incapable of becoming the active window. Under Windows CE, the values for this parameter are limited to only three allowable states: normal (SW_SHOW), hidden (SW_HIDE), or show without activate (SW_SHOWNOACTIVATE). Unless an application needs to force its window to a predefined state, this parameter is simply passed without modification to the *ShowWindow* function after the program's main window has been created.

On entry into *WinMain*, a call is made to *InitApp*, where the window class for the main window is registered. After that, a call to *InitInstance* is made; the main window is created in this function. I'll talk about how these two routines operate shortly, but for now I'll continue with *WinMain*, proceeding on the assumption that at the return from *InitInstance* the program's main window has been created.

The message loop

After the main window has been created, *WinMain* enters the message loop, which is the heart of every Windows application. HelloCE's message loop is shown here:

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

The loop is simple: *GetMessage* is called to get the next message in the application's message queue. If no message is available, the call waits, blocking that application's thread until one is available. When a message is available, the call returns with the message data contained in a MSG structure. The MSG structure itself contains fields that identify the message, provide any message-specific parameters, and identify the last point on the screen touched by the pen before the message was sent. This location information is different from the standard Win32 message point data in that in Windows 9*x* or Windows NT the point returned is the current mouse position instead of the last point clicked (or tapped, as in Windows CE).

The *TranslateMessage* function translates appropriate keyboard messages into a character message. (I'll talk about others of these filter type messages, such as

IsDialogMsg, later.) The *DispatchMessage* function then tells Windows to forward the message to the appropriate window in the application.

This *GetMessage*, *TranslateMessage*, *DispatchMessage* loop continues until *GetMessage* receives a WM_QUIT message which, unlike all other messages causes *GetMessage* to return 0. As can be seen from the *while* clause, a return value of 0 by *GetMessage* causes the loop to terminate.

After the message loop terminates, the program can do little else but clean up and exit. In the case of HelloCE, the program calls *TermInstance* to perform any necessary cleanup. HelloCE is a simple program and no cleanup is required. In more complex programs, *TermInstance* would free any system resources that aren't automatically freed when the program terminates.

The value returned by *WinMain* becomes the return code of the program. Traditionally, the return value is the value in the *wParam* parameter of the last message (WM_QUIT). The *wParam* value of WM_QUIT is set when that message is sent in response to a *PostQuitMessage* call made by the application.

InitApp

The goal of *InitApp* is to perform global initialization for all instances of the application that might run. In practice, *InitApp* is a holdover from Win16 days when window classes were registered on an applicationwide basis instead of for every instance, as is done under Win32. Still, having a place for global initialization can have its uses in some applications. For a program as simple as HelloCE, the entire task of *InitApp* can be reduced to registering the application's main window class. The entire procedure is listed below:

```
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register App Main Window class.
    wc.style = 0;                       // Class style flags
    wc.lpfnWndProc = MainWndProc;       // Callback function
    wc.cbClsExtra = 0;                  // Extra class data
    wc.cbWndExtra = 0;                  // Extra window data
    wc.hInstance = hInstance;           // Owner handle
    wc.hIcon = NULL;                    // Application icon
    wc.hCursor = NULL;                  // Default cursor
    wc.hbrBackground = (HBRUSH) = GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;            // Must be NULL
    wc.lpszClassName = szAppName;       // Class name

    if (RegisterClass (&wc) == 0) return 1;

    return 0;
}
```

Registering a window class is simply a matter of filling out a rather extensive structure describing the class and calling the *RegisterClass* function. The parameters assigned to the fields of the WNDCLASS structure define how all instances of the main window for HelloCE will behave. The initial field, *style*, sets the class style for the window. In Windows CE the class styles are limited to the following:

- CS_GLOBALCLASS indicates that the class is global. This flag is provided only for compatibility because all window classes in Windows CE are process global.
- CS_HREDRAW tells the system to force a repaint of the window if the window is sized horizontally.
- CS_VREDRAW tells the system to force a repaint of the window if the window is sized vertically.
- CS_NOCLOSE disables the Close button if one is present on the title bar.
- CS_PARENTDC causes a window to use its parent's device context.
- CS_DBLCLKS enables notification of double-clicks (double-taps under Windows CE) to be passed to the parent window.

The *lpfnWndProc* field should be loaded with the address of the window's window procedure. Because this field is typed as a pointer to a window procedure, the declaration to the procedure must be defined in the source code before the field is set. Otherwise, the compiler's type-checker will flag this line with a warning.

The *cbClsExtra* field allows the programmer to add extra space in the class structure to store class-specific data known only to the application. The *cbWndExtra* field is much handier. This field adds space to the Windows internal structure responsible for maintaining the state of each instance of a window. Instead of storing large amounts of data in the window structure itself, an application should store a pointer to an application-specific structure that contains the data unique to each instance of the window. Under Windows CE, both the *cbClsExtra* and *cbWndExtra* fields must be multiples of 4 bytes.

The *hInstance* field must be filled with the program's instance handle, which specifies the owning process of the window. The *hIcon* field is set to the handle of the window's default icon. The *hIcon* field isn't supported under Windows CE and should be set to NULL. (In Windows CE, the icon for the class is set after the first window of this class is created. For HelloCE, however, no icon is supplied and unlike other versions of Windows, Windows CE doesn't have any predefined icons that can be loaded.)

Unless the application being developed is designed for a Windows CE system with a mouse, the next field, *hCursor*, must be set to NULL. Almost all Windows CE systems use a touch panel instead of a mouse, so you find no cursor support in those systems. For those special systems that do have cursor support, the Windows CE doesn't support animated cursors or colored cursors.

The *hbrBackground* field specifies how Windows CE draws the background of the window. Windows uses the *brush*, a small predefined array of pixels, specified in this field to draw the background of the window. Windows CE provides a number of predefined brushes that you can load using the *GetStockObject* function. If the *hbrBackground* field is NULL, the window must handle the WM_ERASEBKGND message sent to the window telling it to redraw the background of the window.

The *lpszMenuName* field must be set to NULL because Windows CE doesn't support windows directly having a menu. In Windows CE, menus are provided by command bar or command band controls that can be created by the main window.

Finally the *lpszClassName* parameter is set to a programmer-defined string that identifies the class name to Windows. HelloCE uses the *szAppName* string, which is defined globally.

After the entire WNDCLASS structure has been filled out, the *RegisterClass* function is called with a pointer to the WNDCLASS structure as its only parameter. If the function is successful, a value identifying the window class is returned. If the function fails, the function returns 0.

InitInstance

The main task of *InitInstance* is to create the application's main window and display it in the form specified in the *nShowCmd* parameter passed to *WinMain*. The code for *InitInstance* is shown below:

```
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow) {
    HWND hWnd;
    HICON hIcon;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    hWnd = CreateWindow (szAppName,          // Window class
                         TEXT("Hello"),      // Window title
                         WS_VISIBLE,         // Style flags
                         0, 0,               // x, y position
                         CW_USEDEFAULT,      // Initial width
```

(continued)

```
                        CW_USEDEFAULT,      // Initial height
                        NULL,               // Parent
                        NULL,               // Menu, must be null
                        hInstance,          // App instance
                        NULL);              // Ptr to create params
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);

    return hWnd;
}
```

The first task performed by *InitInstance* is to save the program's instance handle *hInstance* in a global variable named *hInst*. The instance handle for a program is useful at a number of points in a Windows application. I save the value here because the instance handle is known, and this is a convenient place in the program to store it.

All Windows programmers learn early in their Windows programming lives the *CreateWindow* function call. Although the number of parameters looks daunting, the parameters are fairly logical once you learn them. The first parameter is the name of the window class of which our window will be an instance. In the case of HelloCE, the class name is a string constant, *szAppName*, which was also used in the WNDCLASS structure.

The next field is referred to as the *window text*. In other versions of Windows, this is the text that would appear on the title bar of a standard window. However, since Windows CE main windows rarely have title bars, this text is used only on the taskbar button for the window. The text is couched in a TEXT macro, which insures that the string will be converted to Unicode under Windows CE.

The style flags specify the initial styles for the window. The style flags are used both for general styles that are relevant to all windows in the system and for class-specific styles, such as those that specify the style of a button or a list box. In this case, all we need to specify is that the window be created initially visible with the WS_VISIBLE flag. Experienced Win32 programmers should refer to the documentation for *CreateWindow* because there are a number of window style flags that aren't supported under Windows CE.

The next four fields specify the initial position and size of the window. Since most applications under Windows CE are maximized (that is, they take up the entire screen above the taskbar), the size and position fields are set to default values, which are indicated by the CW_USEDEFAULT flag in each of the fields. The default value settings create a window that's maximized under the current versions of Windows CE but also compatible with future versions of the operating system, which might not

maximize every window. Be careful not to assume any particular screen size for a Windows CE device because different implementations have different screen sizes.

The next field is set to the handle of the parent window. Because this is the top-level window, the parent window field is set to NULL. The menu field is also set to NULL because Windows CE supports menus through the command bar and command bands controls.

The *hInstance* parameter is the same instance handle that was passed to the program. Creating windows is one place where that instance handle, saved at the start of the routine, comes in handy. The final parameter is a pointer that can be used to pass data from the *CreateWindow* call to the window procedure during the WM_CREATE message. In this example, no additional data needs to be passed, so the parameter is set to NULL.

If successful, the *CreateWindow* call returns the handle to the window just created, or it returns 0 if an error occurred during the function. That window handle is then used in the two statements (*ShowWindow* and *UpdateWindow*) just after the error-checking *if* statement. The *ShowWindow* function modifies the state of the window to conform with the state given in the *nCmdShow* parameter passed to *WinMain*. The *UpdateWindow* function forces Windows to send a WM_PAINT message to the window that has just been created.

That completes the *InitApp* function. At this point, the application's main window has been created and updated. So even before we have entered the message loop, messages have been sent to the main window's window procedure. It's about time to look at this part of the program.

MainWndProc

You spend most of your programming time with the window procedure when you're writing a Windows program. *WinMain* contains mainly initialization and cleanup code that, for the most part, is boilerplate. The window procedure, on the other hand, is the core of the program, the place where the actions of the program's windows create the personality of the program.

```
LRESULT CALLBACK MainWndProc(HWND hWnd, UINT wMsg, WPARAM wParam,
                             LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc(hWnd, wMsg, wParam, lParam);
}
```

All window procedures, regardless of their window class, are declared with the same parameters. The LRESULT return type is actually just a long (a *long* is a 32-bit value under Windows) but is typed this way to provide a level of indirection between the source code and the machine. While you can easily look into the include files to determine the real type of variables that are used in Windows programming, this can cause problems when you're attempting to move your code across platforms. Though it can be useful to know the size of a variable type for memory-use calculations, there is no good reason, and there are plenty of bad ones, not to use the type definitions provided by windows.h.

The CALLBACK type definition specifies that this function is an external entry point into the EXE, necessary because Windows calls this procedure directly, and that the parameters will be put in a Pascal-like right-to-left push onto the program stack, which is the reverse of the standard C-language method. The reason for using the Pascal language stack frame for external entry points goes back to the very earliest days of Windows development. The use of a fixed-size, Pascal stack frame meant that the called procedure cleaned up the stack instead of leaving it for the caller to do. This reduced the code size of Windows and its bundled accessory programs sufficiently so that the early Microsoft developers thought it was a good move.

The first of the parameters passed to the window procedure is the window handle, which is useful when you need to define the specific instance of the window. The *wMsg* parameter indicates the message being sent to the window. This isn't the MSG structure used in the message loop in *WinMain*, but a simple, unsigned integer containing the message value. The remaining two parameters, *wParam* and *lParam*, are used to pass message-specific data to the window procedure. The names *wParam* and *lParam* come to us from the Win16 days, when the *wParam* was a 16-bit value and *lParam* was a 32-bit value. In Windows CE, as in other Win32 operating systems, both the *wParam* and *lParam* parameters are 32 bits wide.

It's in the window procedure that my programming style differs significantly from most Windows programs written without the help of a class library such as MFC. For almost all of my programs, the window procedure is identical to the one shown above. Before continuing, I repeat: this program structure isn't specific to Windows CE. I use this style for all my Windows applications, whether they are for Windows 3.1, Windows 95, Windows NT, or Windows CE.

This style reduces the window procedure to a simple table look-up function. The idea is to scan the *MainMessages* table defined early in the C file for the message value in one of the entries. If the message is found, the associated procedure is then called, passing the original parameters to the procedure processing the message. If no match is found for the message, the *DefWindowProc* function is called. *DefWindowProc* is a Windows function that provides a default action for all messages in the system, which frees a Windows program from having to process every message being passed to a window.

The message table associates message values with a procedure to process it. The table is listed below:

```
// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_PAINT, DoPaintMain,
    WM_HIBERNATE, DoHibernateMain,
    WM_DESTROY, DoDestroyMain,
};
```

The table is defined as a constant, not just as good programming practice but also because it's helpful for memory conservation. Since Windows CE programs can be executed in place in ROM, data that doesn't change should be marked constant. This allows the Windows CE program loader to leave such constant data in ROM instead of loading a copy into RAM so that it can be modified later by the program.

The table itself is an array of a simple two-element structure. The first entry is the message value, followed by a pointer to the function that processes the message. While the functions could be named anything, I'm using a consistent structure throughout the book to help you keep track of them. The names are composed of a *Do* prefix (as a bow to object-oriented practice), followed by the message name and a suffix indicating the window class associated with the table. So, *DoCreateMain* is the name of the function that processes WM_CREATE messages for the main window of the program.

DoCreateMain

The WM_CREATE message is the first message sent to a window. WM_CREATE is unique among messages in that Windows sends it while processing the *CreateWindow* function, and therefore the window has yet to be completely created. This is a good place in the code to perform any data initialization for the window. But since the window is still being created, some Windows functions, such as *GetWindowRect*, used to query the size and position of the window, return inaccurate values. For our purposes, the procedure shown in the following code performs only one function: it creates a command bar for the window.

```
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB;

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    return 0;
}
```

Because Windows CE windows don't support standard menus attached to windows, a command bar is necessary for menus. While HelloCE doesn't have a menu, it does require a Close button, also provided by the command bar, so the program can be terminated by the user. For this reason, the simplest form of command bar, one with only a Close button, is created. You create the command bar by calling *CommandBar_Create* and passing the program's instance handle, the handle to the window, and a constant that will be used to identify this specific command bar. (This constant can be any integer value as long as it is unique among the other child windows in the window.) Once you've created the command bar, you add a Close button by calling *CommandBar_AddAdornments*. Since all we want to do is perform the default action for this function, the parameters passed are basic: the command bar handle and two zeros. That completes the processing of the WM_CREATE message. I'll examine the command bar in depth in Chapter 5.

DoPaintMain

Painting the window, and therefore processing the WM_PAINT message, is one of the critical functions of any Windows program. As a program processes the WM_PAINT message, the look of the window is achieved. Aside from painting the default background with the brush you specified when you registered the window class, Windows provides no help for processing this message. In HelloCE, the task of the *DoPaintMain* procedure is to display one line of text in the center of the window.

```
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect;
    HDC hdc;

    // Adjust the size of the client rect to take into account
    // the command bar height.
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    hdc = BeginPaint (hWnd, &ps);
    DrawText (hdc, TEXT ("Hello Windows CE!"), -1, &rect,
              DT_CENTER | DT_VCENTER | DT_SINGLELINE);

    EndPaint (hWnd, &ps);
    return 0;
}
```

Before the drawing can be performed, the routine must determine the size of the window. In a Windows program, a standard window is divided into two areas, the nonclient area and the client area. A window's title bar and its sizing border commonly comprise the nonclient area of a window, and Windows is responsible for drawing it. The client area is the interior part of the window, and the application is responsible for drawing that. An application determines the size and location of the client area by calling the *GetClientRect* function. The function returns a RECT structure that contains left, top, right, and bottom elements that delineate the boundaries of the client rectangle. The advantage of the client vs. nonclient area concept is that an application doesn't have to account for drawing such standard elements of a window as the title bar.

When you're computing the size of the client area, you must remember that the command bar resides in the client area of the window. So, even though the *GetClientRect* function works identically in Windows CE as in other versions of Windows, the application needs to compensate for the height of the command bar, which is always placed across the top of the window. Windows CE gives you a convenient function, *CommandBar_Height*, which returns the height of the command bar and can be used in conjunction with the *GetClientRect* call to get the true client area of the window that needs to be drawn by the application.

Other versions of Windows supply a series of WM_NC*xxx* messages that enable your applications to take over the drawing of the nonclient area. In Windows CE, windows seldom have title bars and at the present time, none of them have a sizing border. Because there's so little nonclient area, the Windows CE developers decided not to expose the nonclient messages.

All drawing performed in a WM_PAINT message must be enclosed by two functions, *BeginPaint* and *EndPaint*. The *BeginPaint* function returns an *HDC*, or handle to a device context. A *device context* is a logical representation of a physical display device such as a video screen or a printer. Windows programs never modify the display hardware directly. Instead, Windows isolates the program from the specifics of the hardware with, among other tools, device contexts.

BeginPaint also fills in a PAINTSTRUCT structure that contains a number of useful parameters.

```
typedef struct tagPAINTSTRUCT {
    HDC  hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
} PAINTSTRUCT;
```

The *hdc* field is the same handle that's returned by the *BeginPaint* function. The *fErase* field indicates whether the background of the window needs to be redrawn by the window procedure. The *rcPaint* field is a RECT structure that defines the client area that needs repainting. HelloCE ignores this field and assumes that the entire client window needs repainting for every WM_PAINT message, but this field is quite handy when performance is an issue because only a part of the window might need repainting. Windows actually prevents repainting outside of the *rcPaint* rectangle even when a program attempts to do so. The other fields in the structure, *fRestore*, *fIncUpdate*, and *rgbReserved*, are used internally by Windows and can be ignored by the application.

The only painting that takes place in HelloCE occurs in one line of text in the window. To do the painting, HelloCE calls the *DrawText* function. I cover the details of *DrawText* in the next chapter, but if you look at the function it's probably obvious to you that this call draws the string "Hello Windows CE" on the window. After *DrawText* returns, *EndPaint* is called to inform Windows that the program has completed its update of the window.

Calling *EndPaint* also validates any area of the window you didn't paint. Windows keeps a list of areas of a window that are *invalid* (areas that need to be redrawn) and *valid* (areas that are up to date). By calling the *BeginPaint* and *EndPaint* pair, you tell Windows that you've taken care of any invalid areas in your window, whether or not you've actually drawn anything in the window. In fact, you must call *BeginPaint* and *EndPaint*, or validate the invalid areas of the window by other means, or Windows will simply continue to send WM_PAINT messages to the window until those invalid areas are validated.

DoHibernateMain

You need *DoHibernateMain* because the WM_HIBERNATE message, unique to Windows CE, should be handled by every Windows CE program. A WM_HIBERNATE message is sent to a window to instruct it to reduce its memory use to the absolute minimum.

```
LRESULT DoHibernateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {

    // If not the active window, destroy the cmd bar to save memory.
    if (GetActiveWindow () != hWnd)
        CommandBar_Destroy (GetDlgItem (hWnd, IDC_CMDBAR));

    return 0;
}
```

In the case of HelloCE, the only real way to reduce memory use is to destroy the command bar control. This is done by means of a call to *CommandBar_Destroy*.

The only case in which one should not destroy the command bar is when the window is the active window, the window through which the user is interacting with the program at the current time.

More complex Windows CE applications have a much more elaborate procedure for handling the WM_HIBERNATE messages. Applications should free up as much memory and system resources as possible without losing currently unsaved data. In a choice between performance and lower memory use, an application is better reactivating slowly after a WM_HIBERNATE message than it is consuming more memory.

DoActivateMain

While the WM_ACTIVATE message is common to all Windows platforms, it takes on new significance for Windows CE applications because among its duties is to indicate that the window should restore any data structures or window controls that were freed by a WM_HIBERNATE message.

```
LRESULT DoActivateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                        LPARAM lParam) {
    HWND hwndCB;

    // If activating and no command bar, create it.
    if ((LOWORD (wParam) != WA_INACTIVE) &&
        (GetDlgItem (hWnd, IDC_CMDBAR) == 0)) {

        // Create a command bar.
        hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

        // Add exit button to command bar.
        CommandBar_AddAdornments (hwndCB, 0, 0);
    }
    return 0;
}
```

The lower word of the *wParam* parameter is a flag that tells why the WM_ACTIVATE message was sent to the window. The flag can be one of three values: WA_INACTIVE, indicating that the window is being deactivated after being the active window; WA_ACTIVE, indicating that the window is about to become the active window; and WA_CLICKACTIVE, indicating that the window is about to become the active window after having been clicked on by the user.

HelloCE processes this message by checking to see whether the window remains active and whether the command bar no longer exists. If both conditions are true, the command bar is re-created using the same calls used for the WM_CREATE message. The *GetDlgItem* function is convenient because it returns the handle of a child window of another window using its window ID. Remember that when the command bar, a

child of HelloCE's main window, was created, I used an ID of IDC_CMDBAR (defined in HelloCE.h). That ID value is passed to *GetDlgItem* to get the command bar window handle. However, if the command bar window doesn't exist, the value returned is 0, indicating that HelloCE needs to re-create the command bar.

DoDestroyMain

The final message that HelloCE must process is the WM_DESTROY message sent when a window is about to be destroyed. Because this window is the main window of the application, the application should terminate when the window is destroyed. To make this happen, the *DoDestroyMain* function calls *PostQuitMessage*. This function places a WM_QUIT message in the message queue. The one parameter of this function is the return code value that will be passed back to the application in the *wParam* parameter of the WM_QUIT message.

```
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
```

Notice that the *DoDestroyMain* function doesn't destroy the command bar control created in *DoCreateMain*. Since the command bar is a child window of the main window, it's automatically destroyed when its parent window is destroyed.

As I've mentioned, when the message loop sees a WM_QUIT message, it exits the loop. The *WinMain* function then calls *TermInstance*, which in the case of HelloCE, does nothing but return. *WinMain* then returns, terminating the program.

Running HelloCE

After you've entered the program into Visual C++ and built it, it can be executed by a double-tap on the HelloCE icon. The program displays the Hello Windows CE text in the middle of an empty window, as shown in Figure 1-4. Figure 1-5 shows HelloCE running on a Palm-size PC. The command bar is placed by Windows CE across the top of the window. Tapping on the Close button on the command bar causes Windows CE to send a WM_CLOSE message to the window. Although HelloCE doesn't explicitly process the WM_CLOSE message, the *DefWindowProc* procedure enables default processing by destroying the main window. As the window is being destroyed, a WM_DESTROY message is sent, which causes *PostQuitMessage* to be called.

Figure 1-4. *The HelloCE window on an H/PC.*

Figure 1-5. *The HelloCE window on a Palm-size PC.*

As I said, HelloCE is a very basic Windows CE program but it does gives you a skeleton of a Windows CE application upon which you can build. If you look at HelloCE.EXE using Explorer, the program is represented by a generic icon. When HelloCE is running, the button on the task bar representing HelloCE has no icon displayed next to the text. How to add a program's icon as well as how the *DrawText* function works are a couple of the topics I'll address in the next few chapters.

Chapter 2

Drawing on the Screen

In Chapter 1, the example program HelloCE had one task: to display a line of text on the screen. Displaying that line took only one call to *DrawText* with Windows CE taking care of such details as the font and its color, the positioning of the line of text inside the window, and so forth. Given the power of a graphical user interface (GUI), however, an application can do much more than simply print a line of text on the screen. It can craft the look of the display down to the most minute of details.

Over the life of the Microsoft Windows operating system, the number of functions available for crafting these displays has expanded dramatically. With each successive version of Windows, functions have been added that extend the tools available to the programmer. As functions were added, the old ones remained so that even if a function had been superseded by a new function old programs would continue to run on the newer versions of Windows. The approach in which function after function is piled on while the old functions are retained for backward compatibility was discontinued with the initial version of Windows CE. Because of the requirement to produce a smaller version of Windows, the CE team took a hard look at the Win32 API and replicated only the functions absolutely required by applications written for the Windows CE target market.

One of the areas of the Win32 API hardest hit by this reduction was graphical functions. Not that you now lack the functions to do the job—it's just that the high degree of redundancy led to some major pruning of the Win32 graphical functions.

An added challenge for the programmer is that different Windows CE platforms have subtly different sets of supported APIs. One of the ways in which Windows CE graphics support differs from that of its desktop cousins is that Windows CE doesn't support the different mapping modes available under other implementations of Windows. Instead, the Windows CE device contexts are always set to the MM_TEXT mapping mode. Coordinate transformations are also not supported under Windows CE. While these features can be quite useful for some types of applications, such as desktop publishing, their necessity in the Windows CE environment of small portable devices isn't as clear. Fortunately, as Windows CE matures we can expect more and more of the basic Win32 API to be supported.

So when you're reading about the functions and techniques used in this chapter, remember that some might not be supported on all platforms. So that a program can determine what functions are supported, Windows has always had the *GetDeviceCaps* function, which returns the capabilities of the current graphic device. Throughout this chapter, I'll refer to *GetDeviceCaps* when determining what functions are supported on a given device.

This chapter, like the other chapters in Part I of this book, reviews the drawing features supported by Windows CE. One of the most important facts to remember is that while Windows CE doesn't support the full Win32 graphics API, its rapid evolution has resulted in it supporting some of the newest functions in Win32—some so new that you might not be familiar with them. This chapter shows you the functions you can use and how to work around the areas where certain functions aren't supported under Windows CE.

PAINTING BASICS

Historically, Windows has been subdivided into three main components: the kernel, which handles the process and memory management; User, which handles the windowing interface and controls; and the Graphics Device Interface, or GDI, which performs the low-level drawing. In Windows CE, User and GDI are combined into the Graphics Windowing and Event handler, or GWE. At times, you might hear a Windows CE programmer talk about the GWE. The GWE is nothing really new—just a different packaging of standard Windows parts. In this book, I usually refer to the graphics portion of the GWE under its old name, GDI, to be consistent with standard Windows programming terminology.

But whether you're programming for Windows CE or Windows 98 or Windows NT, there is more to drawing than simply handling the WM_PAINT message. It's helpful to understand just when and why a WM_PAINT message is sent to a window.

Valid and Invalid Regions

When for some reason an area of a window is exposed to the user, that area, or *region*, as it's referred to in Windows, is marked invalid. When no other messages are waiting in an application's message queue and the application's window contains an invalid region, Windows sends a WM_PAINT message to the window. As mentioned in Chapter 1, any drawing performed in response to a WM_PAINT message is couched in calls to *BeginPaint* and *EndPaint*. *BeginPaint* actually performs a number of actions. It marks the invalid region as valid, and it computes the *clipping* region. The clipping region is the area to which the painting action will be limited. *BeginPaint* then sends a WM_ERASEBACKGROUND message, if needed, to redraw the background, and it hides the caret—the text entry cursor—if it's displayed. Finally *BeginPaint* retrieves the handle to the display device context so that it can be used by the application. The *EndPaint* function releases the device context and redisplays the caret if necessary. If no other action is performed by a WM_PAINT procedure, you must at least call *BeginPaint* and *EndPaint* if only to mark the invalid region as valid.

Alternatively, you can call to *ValidateRect* to blindly validate the region. But no drawing can take place in that case because an application must have a handle to the device context before it can draw anything in the window.

Often an application needs to force a repaint of its window. An application should never post or send a WM_PAINT message to itself or to another window. Instead, you do the following:

```
BOOL InvalidateRect (HWND hWnd, const RECT *lpRect, BOOL bErase);
```

Notice that *InvalidateRect* doesn't require a handle to the window's device context, only to the window handle itself. The *lpRect* parameter is the area of the window to be invalidated. This value can be NULL if the entire window is to be invalidated. The *bErase* parameter indicates whether the background of the window should be redrawn during the *BeginPaint* call as mentioned above. Note that unlike other versions of Windows, Windows CE requires that the *hWnd* parameter be a valid window handle.

Device Contexts

A *device context,* often referred to simply as a DC, is a tool that Windows uses to manage access to the display and printer, although for the purposes of this chapter I'll be talking only about the display. Also, unless otherwise mentioned, the explanation that follows applies to Windows in general and isn't specific to Windows CE.

Windows applications never write directly to the screen. Instead, they request a handle to a display device context for the appropriate window, and then using the handle, draw to the device context. Windows then arbitrates and manages getting the pixels from the DC to the screen.

BeginPaint, which should only be called in a WM_PAINT message, returns a handle to the display DC for the window. An application usually performs its drawing to the screen during the WM_PAINT messages. Windows treats painting as a low-priority task, which is appropriate since having painting at a higher priority would result in a flood of paint messages for every little change to the display. Allowing an application to complete all its pending business by processing all waiting messages results in all the invalid regions being painted efficiently at once. Users don't notice the minor delays caused by the low priority of the WM_PAINT messages.

Of course, there are times when painting must be immediate. An example of such a time might be when a word processor needs to display a character immediately after its key is pressed. To draw outside a WM_PAINT message, the handle to the DC can be obtained using this:

```
HDC GetDC (HWND hWnd);
```

GetDC returns a handle to the DC for the client portion of the window. Drawing can then be performed anywhere within the client area of the window because this process isn't like processing inside a WM_PAINT message; there's no clipping to restrict you from drawing in an invalid region.

Windows CE 2.1 supports another function that can be used to receive the DC. It is

```
HDC GetDCEx (HWND hWnd, HRGN hrgnClip, DWORD flags);
```

GetDCEx allows you to have more control over the device context returned. The new parameter, *hrgnClip* lets you define the clipping region, which limits drawing to that region of the DC. The *flags* parameter lets you specify how the DC acts as you draw on it. Windows CE doesn't support the following flags: DCX_PARENTCLIP, DCX_NORESETATTRS, DCX_LOCKWINDOWUPDATE, and DCX_VALIDATE.

After the drawing has been completed, a call must be made to release the device context:

```
int ReleaseDC (HWND hWnd, HDC hDC);
```

Device contexts are a shared resource, and therefore an application must not hold the DC for any longer than necessary.

While *GetDC* is used to draw inside the client area, sometimes an application needs access to the nonclient areas of a window, such as the title bar. To retrieve a DC for the entire window, make the following call:

```
HDC GetWindowDC (HWND hWnd);
```

As before, the matching call after drawing has been completed for *GetWindowDC* is *ReleaseDC*.

The DC functions under Windows CE are identical to the device context functions under Windows 98 and Windows NT. This should be expected because DCs are the core of the Windows drawing philosophy. Changes to this area of the API would result in major incompatibilities between Windows CE applications and their desktop counterparts.

WRITING TEXT

In Chapter 1, the HelloCE example displayed a line of text using a call to *DrawText*. That line from the example is shown here:

```
DrawText (hdc, TEXT ("Hello Windows CE!"), -1, &rect,
          DT_CENTER | DT_VCENTER | DT_SINGLELINE);
```

DrawText is a fairly high-level function that allows a program to display text while having Windows deal with most of the details. The first few parameters of *DrawText* are almost self-explanatory. The handle of the device context being used is passed, along with the text to display couched in a TEXT macro, which declares the string as a Unicode string necessary for Windows CE. The third parameter is the number of characters to print, or as is the case here, a –1 indicating that the string being passed is null terminated and Windows should compute the length.

The fourth parameter is a pointer to a rect structure that specifies the formatting rectangle for the text. *DrawText* uses this rectangle as a basis for formatting the text to be printed. How the text is formatted depends on the function's last parameter, the formatting flags. These flags specify how the text is to be placed within the formatting rectangle, or in the case of the DT_CALCRECT flag, the flags have *DrawText* compute the dimensions of the text that is to be printed. *DrawText* even formats multiple lines with line breaks automatically computed. In the case of HelloCE, the flags specify that the text should be centered horizontally (DT_CENTER), and centered vertically (DT_VCENTER). The DT_VCENTER flag works only on single lines of text, so the final parameter, DT_SINGLELINE, specifies that the text shouldn't be flowed across multiple lines if the rectangle isn't wide enough to display the entire string.

Device Context Attributes

What I haven't mentioned yet about HelloCE's use of *DrawText* is the large number of assumptions the program makes about the DC configuration when displaying the text. Drawing in a Windows device context takes a large number of parameters, such as foreground and background color and how the text should be drawn over the background as well as the font of the text. Instead of specifying all these parameters for each drawing call, the device context keeps track of the current settings, referred to as *attributes*, and uses them as appropriate for each call to draw to the device context.

Foreground and background colors

The most obvious of the text attributes are the foreground and background color. Two functions, *SetTextColor* and *GetTextColor*, allow a program to set and retrieve the current color. These functions work well with both four-color gray-scale screens as well as the color screens supported by Windows CE devices.

To determine how many colors a device supports, use *GetDeviceCaps* as mentioned previously. The prototype for this function is the following:

```
int GetDeviceCaps (HDC hdc, int nIndex);
```

You need the handle to the DC being queried because different DCs have different capabilities. For example, a printer DC differs from a display DC. The second parameter indicates the capability being queried. In the case of returning the colors available on the device, the NUMCOLORS value returns the number of colors as long as the device supports 256 colors or fewer. Beyond that, the returned value for NUMCOLORS is –1 and the colors can be returned using the BITSPIXEL value, which returns the number of bits used to represent each pixel. This value can be converted to the number of colors by raising 2 to the power of the BITSPIXEL returned value, as in the following code sample:

```
nNumColors = GetDeviceCaps (hdc, NUMCOLORS);
if (nNumColors == -1)
    nNumColors = 1 << GetDeviceCaps (hdc, BITSPIXEL);
```

Drawing mode

Another attribute that affects text output is the background mode. When letters are drawn on the device context, the system draws the letters themselves in the foreground color. The space between the letters is another matter. If the background mode is set to opaque, the space is drawn with the current background color. But if the background mode is set to transparent, the space between the letters is left in whatever state it was in before the text was drawn. While this might not seem like a big difference, imagine a window background filled with a drawing or graph. If text is written over the top of the graph and the background mode is set to opaque, the area around the text will be filled, and the background color will overwrite the graph. If the background mode is transparent, the text will appear as if it had been placed on the graph, and the graph will show through between the letters of the text.

The TextDemo Example Program

The TextDemo program, shown in Figure 2-1, demonstrates the relationships among the text color, the background color, and the background mode.

TextDemo.h

```
//--------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//--------------------------------------------------------------------
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

//--------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                             // Structure associates
    UINT Code;                                  // messages
                                                // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                              // Structure associates
    UINT Code;                                  // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);     // function.
};

//--------------------------------------------------------------------
// Generic defines used by application
#define IDC_CMDBAR      1                       // Command bar ID

//--------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
int InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);
```

Figure 2-1. *The TextDemo program.* *(continued)*

Figure 2-1. *continued*

TextDemo.c

```
//------------------------------------------------------------------
// TextDemo - Text output demo
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//------------------------------------------------------------------
#include <windows.h>                    // For all that Windows stuff
#include <commctrl.h>                   // Command bar includes
#include "TextDemo.h"                   // Program-specific stuff

//------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("TextDemo");
HINSTANCE hInst;                        // Program instance handle

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_PAINT, DoPaintMain,
    WM_DESTROY, DoDestroyMain,
};

//==================================================================
//
// Program Entry Point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;

    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    if ((rc = InitInstance (hInstance, lpCmdLine, nCmdShow)) != 0)
        return rc;
```

```
    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//----------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application main window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = MainWndProc;                   // Callback function
    wc.cbClsExtra = 0;                              // Extra class data
    wc.cbWndExtra = 0;                              // Extra window data
    wc.hInstance = hInstance;                       // Owner handle
    wc.hIcon = NULL;                                // Application icon
    wc.hCursor = NULL;                              // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                        // Menu name
    wc.lpszClassName = szAppName;                   // Window class name

    if (RegisterClass (&wc) == 0) return 1;

    return 0;
}
//----------------------------------------------------------------------
// InitInstance - Instance initialization
//
int InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow){
    HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    hWnd = CreateWindow (szAppName,                 // Window class
                         TEXT("TextDemo"),          // Window title
                         WS_VISIBLE,                // Style flags
                         CW_USEDEFAULT,             // x position
                         CW_USEDEFAULT,             // y position
```

(continued)

Figure 2-1. *continued*

```
                        CW_USEDEFAULT,      // Initial width
                        CW_USEDEFAULT,      // Initial height
                        NULL,               // Parent
                        NULL,               // Menu, must be null
                        hInstance,          // Application instance
                        NULL);              // Pointer to create
                                            // Parameters
    // Return fail code if window not created.
    if ((!hWnd) || (!IsWindow (hWnd))) return 0x10;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return 0;
}
//----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {

    return nDefRC;
}
//======================================================================
// Message handling procedures for MainWindow
//
//----------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
```

```
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB;

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    return 0;
}
//------------------------------------------------------------------------
// DoPaintMain - Process WM_PAINT message for window.
//
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect, rectCli;
    HBRUSH hbrOld;
    HDC hdc;
    INT i, cy;
    DWORD dwColorTable[] = {0x00000000, 0x00808080,
                            0x00cccccc, 0x00ffffff};

    // Adjust the size of the client rect to take into account
    // the command bar height.
    GetClientRect (hWnd, &rectCli);
    rectCli.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    hdc = BeginPaint (hWnd, &ps);

    // Get the height and length of the string.
    DrawText (hdc, TEXT ("Hello Windows CE"), -1, &rect,
              DT_CALCRECT | DT_CENTER | DT_SINGLELINE);

    cy = rect.bottom - rect.top + 5;

    // Draw black rectangle on right half of window.
    hbrOld = SelectObject (hdc, GetStockObject (BLACK_BRUSH));
    Rectangle (hdc, rectCli.left + (rectCli.right - rectCli.left) / 2,
               rectCli.top, rectCli.right, rectCli.bottom);
    SelectObject (hdc, hbrOld);

    rectCli.bottom = rectCli.top + cy;
```

(continued)

Figure 2-1. *continued*

```
    SetBkMode (hdc, TRANSPARENT);
    for (i = 0; i < 4; i++) {
        SetTextColor (hdc, dwColorTable[i]);
        SetBkColor (hdc, dwColorTable[3-i]);

        DrawText (hdc, TEXT ("Hello Windows CE"), -1, &rectCli,
                  DT_CENTER | DT_SINGLELINE);
        rectCli.top += cy;
        rectCli.bottom += cy;
    }

    SetBkMode (hdc, OPAQUE);
    for (i = 0; i < 4; i++) {
        SetTextColor (hdc, dwColorTable[i]);
        SetBkColor (hdc, dwColorTable[3-i]);

        DrawText (hdc, TEXT ("Hello Windows CE"), -1, &rectCli,
                  DT_CENTER | DT_SINGLELINE);
        rectCli.top += cy;
        rectCli.bottom += cy;
    }
    EndPaint (hWnd, &ps);
    return 0;
}
//------------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
```

The meat of TextDemo is in the *OnPaintMain* function. The first call to *DrawText* doesn't draw anything in the device context. Instead, the DT_CALCRECT flag instructs Windows to store the dimensions of the rectangle for the text string in *rect*. This information is used to compute the height of the string, which is stored in *cy*. Next, a black rectangle is drawn on the right side of the window. I'll talk about how a rectangle is drawn later in the chapter; it's used in this program to produce two different backgrounds before the text is written. The function then prints out the same string using different foreground and background colors and both the transparent and opaque drawing modes. The result of this combination is shown in Figure 2-2.

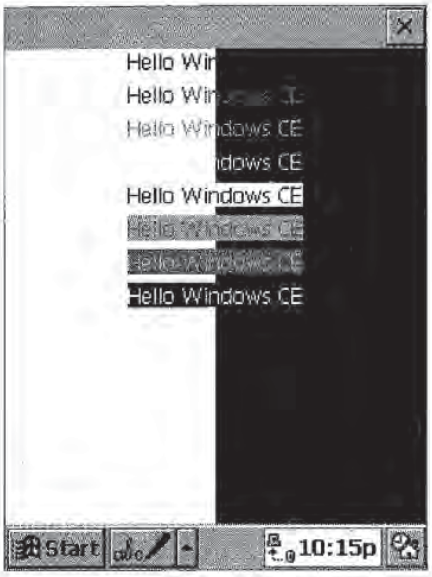


Figure 2-2. *TextDemo shows how the text color, background color, and background mode relate.*

The first four lines are drawn using the transparent mode. The second four are drawn using the opaque mode. The text color is set from black to white, so that each line drawn uses a different color, while at the same time the background color is set from white to black. In transparent mode, the background color is irrelevant because it isn't used; but in opaque mode, the background color is readily apparent on each line.

Fonts

If the ability to set the foreground and background colors were all the flexibility that Windows provided, we might as well be back in the days of MS-DOS and character attributes. Arguably, the most dramatic change from MS-DOS is Windows' ability to change the font used to display text. All Windows operating systems are built around the concept of WYSIWYG—what you see is what you get—and changeable fonts are a major tool used to achieve that goal.

Two types of fonts appear in all Windows operating systems—*raster* and *TrueType*. Raster fonts are stored as bitmaps, small pixel by pixel images, one for each character in the font. Raster fonts are easy to store and use but have one major problem: they don't scale well. Just as a small picture looks grainy when blown up to a much larger size, raster fonts begin to look blocky as they are scaled to larger and larger font sizes.

TrueType fonts solve the scaling problem. Instead of being stored as images, each TrueType character is stored as a description of how to draw the character. The font engine, which is the part of Windows that draws characters on the screen, then takes the description and draws it on the screen in any size needed. TrueType font support was introduced with Windows 3.1 but was only added to the Windows CE line in Windows CE 2.0. Even under Windows CE 2.0, though, some devices such as the original Palm-size PC, don't support TrueType fonts. A Windows CE system can support either TrueType or raster fonts, but not both. Fortunately, the programming interface is the same for both raster and TrueType fonts, relieving Windows developers from worrying about the font technology in all but the most exacting of applications.

The font functions under Windows CE closely track the same functions under other versions of Windows. Let's look at the functions used in the life of a font, from creation through selection in a DC and finally to deletion of the font. How to query the current font as well as enumerate the available fonts is also covered in the following sections.

Creating a font

Before an application is able to use a font other than the default font, the font must be created and then selected into the device context. Any text drawn in a DC after the new font has been selected into the DC will then use the new font.

Creating a font in Windows CE can be accomplished this way:

```
HFONT CreateFontIndirect (const LOGFONT *lplf);
```

This function is passed a pointer to a LOGFONT structure that must be filled with the description of the font you want.

```
typedef struct tagLOGFONT {
    LONG lfHeight;
    LONG lfWidth;
    LONG lfEscapement;
    LONG lfOrientation;
    LONG lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;
    BYTE lfOutPrecision;
    BYTE lfClipPrecision;
    BYTE lfQuality;
    BYTE lfPitchAndFamily;
    TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

The *lfHeight* field specifies the height of the font in device units. If this field is 0, the font manager returns the default font size for the font family requested. For most applications, however, you want to create a font of a particular point size. The following equation can be used to convert point size to the *lfHeight* field:

*lfHeight = -1 * (PointSize * GetDeviceCaps (hdc, LOGPIXELSY) / 72);*

Here, *GetDeviceCaps* is passed a LOGPIXELSY field instructing it to return the number of logical pixels per inch in the vertical direction. The 72 is the number of *points* (a typesetting unit of measure) per inch.

The *lfWidth* field specifies the average character width. Since the height of a font is more important than its width, most programs set this value to 0. This tells the font manager to compute the proper width based on the height of the font. The *lfEscapement* and *lfOrientation* fields specify the angle in tenths of degrees of the base line of the text and the *x*-axis. The *lfWeight* field specifies the boldness of the font from 0 through 1000, with 400 being a normal font and 700 being bold. The next three fields specify whether the font is to be italic, underline, or strikeout.

The *lpCharSet* field specifies the character set you have chosen. This field is more important in international releases of software, where it can be used to request a specific language's character set. The *lfOutPrecision* field can be used to specify how closely Windows matches your requested font. Among a number of flags available, a OUT_TT_ONLY_PRECIS flag specifies that the font created must be a TrueType font. The *lfClipPrecision* field specifies how Windows should clip characters that are partially outside the region being displayed. The *lfQuality* field is set to either DEFAULT_QUALITY or DRAFT_QUALITY, which gives Windows permission to synthesize a font that, while more closely matching the other requested fields, might look less polished.

The *lfPitchAndFamily* field specifies the family of the font you want. This field is handy when you're requesting a family such as Swiss, that features proportional fonts without serifs, or a family such as Roman, that features proportional fonts with serifs, but you don't have a specific font in mind. You can also use this field to specify simply a proportional or a monospaced font and allow Windows to determine which font matches the other specified characteristics passed into the LOGFONT structure. Finally, the *lfFaceName* field can be used to specify the typeface name of a specific font.

When *CreateFontIndirect* is called with a filled LOGFONT structure, Windows creates a logical font that best matches the characteristics provided. To use the font however, the final step of selecting the font into a device context must be made.

Selecting a font into a device context

You select a font into a DC by using the following function:

```
HGDIOBJ SelectObject (HDC hdc, HGDIOBJ hgdiobj);
```

This function is used for more than just setting the default font; you use this function to select other GDI objects, as we shall soon see. The function returns the previously selected object (in our case the previously selected font), which should be saved so that it can be selected back into the DC when we're finished with the new font. The line of code looks like the following:

```
hOldFont = SelectObject (hdc, hFont);
```

When the logical font is selected, the system determines the closest match to the logical font from the fonts available in the system. For devices without TrueType fonts, this match could be a fair amount off from the specified parameters. Because of this, never assume that just because you've requested a particular font, the font returned exactly matches the one you requested. For example, the height of the font you asked for might not be the height of the font that's selected into the device context.

Querying a font's characteristics

To determine the characteristics of the font that is selected into a device context, a call to

```
BOOL GetTextMetrics (HDC hdc, LPTEXTMETRIC lptm);
```

returns the characteristics of that font. A TEXTMETRIC structure is returned with the information and is defined as

```
typedef struct tagTEXTMETRIC {
    LONG tmHeight;
    LONG tmAscent;
    LONG tmDescent;
    LONG tmInternalLeading;
    LONG tmExternalLeading;
    LONG tmAveCharWidth;
    LONG tmMaxCharWidth;
    LONG tmWeight;
    LONG tmOverhang;
    LONG tmDigitizedAspectX;
    LONG tmDigitizedAspectY;
    char tmFirstChar;
    char tmLastChar;
    char tmDefaultChar;
```

```
    char tmBreakChar;
    BYTE tmItalic;
    BYTE tmUnderlined;
    BYTE tmStruckOut;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;
} TEXTMETRIC;
```

The TEXTMETRIC structure contains a number of the fields we saw in the LOGFONT structure but this time the values listed in TEXTMETRIC are the values of the font that's selected into the device context. Figure 2-3 shows the relationship of some of the fields to actual characters.

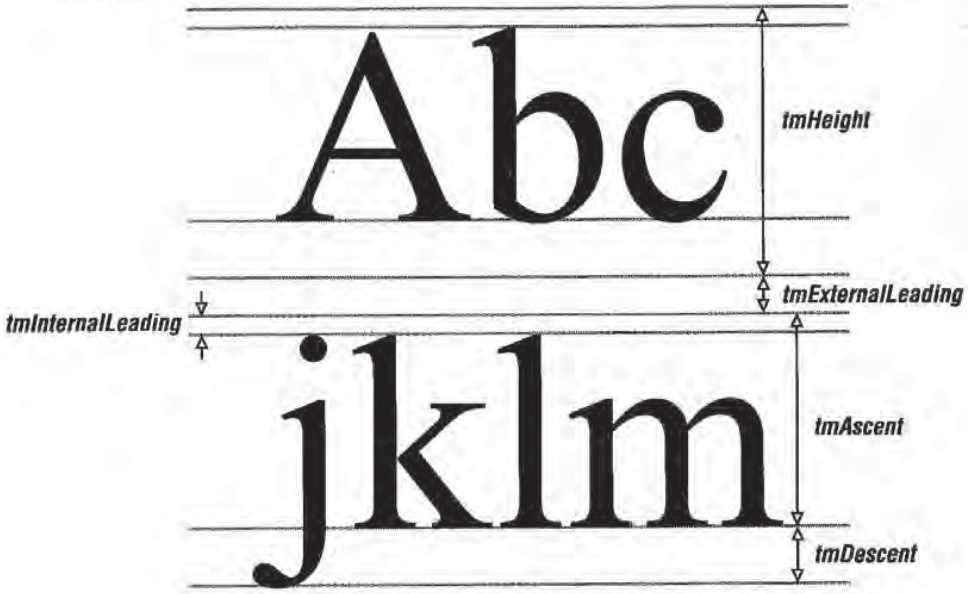


Figure 2-3. *Fields from the TEXTMETRIC structure and how they relate to a font.*

Aside from determining whether you really got the font you wanted, the *GetTextmetrics* call has another valuable purpose—determining the height of the font. Recall that in TextDemo, the height of the line was computed using a call to *DrawText*. While that method is convenient, it tends to be slow. You can use the TEXTMETRIC data to compute this height in a much more straightforward manner. By adding the *tmHeight* field, which is the height of the characters, to the *tmExternalLeading* field, which is the distance between the bottom pixel of one row and the top pixel of the next row of characters, you can determine the vertical distance between the baselines of two lines of text.

Destroying a font

Like other GDI resources, fonts must be destroyed after the program has finished using them. Failure to delete fonts before terminating a program causes what's known as a *resource leak*—an orphaned graphic resource that's taking up valuable memory but that's no longer owned by an application.

To destroy a font, first deselect it from any device contexts it has been selected into. You do this by calling *SelectObject*; the font passed is the font that was returned by the original *SelectObject* call made to select the font. After the font has been deselected, a call to

```
BOOL DeleteObject (HGDIOBJ hObject);
```

(with *hObject* containing the font handle) deletes the font from the system.

As you can see from this process, font management is no small matter in Windows. The many parameters of the LOGFONT structure might look daunting, but they give an application tremendous power to specify a font exactly.

One problem when dealing with fonts is determining just what types of fonts are available on a specific device. Windows CE devices come with a set of standard fonts, but a specific system might have been loaded with additional fonts by either the manufacturer or the user. Fortunately, Windows provides a method for enumerating all the available fonts in a system.

Enumerating fonts

To determine what fonts are available on a system, Windows provides this function:

```
int EnumFontFamilies (HDC hdc, LPCTSTR lpszFamily,
                      FONTENUMPROC lpEnumFontFamProc, LPARAM lParam);
```

This function lets you list all the font families as well as each font within a family. The first parameter is the obligatory handle to the device context. The second parameter is a string to the name of the family to enumerate. If this parameter is null, the function enumerates each of the available families.

The third parameter is something different—a pointer to a function provided by the application. The function is a callback function that Windows calls once for each font being enumerated. The final parameter, *lParam,* is a generic parameter that can be used by the application. This value is passed unmodified to the application's callback procedure.

While the name of the callback function can be anything, the prototype of the callback must match the declaration:

```
int CALLBACK EnumFontFamProc (LOGFONT *lpelf, TEXTMETRIC *lpntm,
                              DWORD FontType, LPARAM lParam);
```

The first parameter passed back to the callback function is a pointer to a LOGFONT structure describing the font being enumerated. The second parameter, a pointer to a textmetric structure, further describes the font. The font type parameter indicates whether the font is a raster or TrueType font.

The FontList Example Program

The FontList program, shown in Figure 2-4, uses the *EnumFontFamilies* function in two ways to enumerate all fonts in the system.

FontList.h

```
//------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//------------------------------------------------------------------
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

//------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                             // Structure associates
    UINT Code;                                  // messages
                                                // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                              // Structure associates
    UINT Code;                                  // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);     // function.
};

//------------------------------------------------------------------
// Generic defines used by application
#define  IDC_CMDBAR 1                           // Command bar ID

//------------------------------------------------------------------
// Program specific structures
//
#define FAMILYMAX   24
```

Figure 2-4. *The FontList program enumerates all fonts in the system.* *(continued)*

Figure 2-4. *continued*

```
typedef struct {
    int nNumFonts;
    TCHAR szFontFamily[LF_FACESIZE];
} FONTFAMSTRUCT;
typedef FONTFAMSTRUCT *PFONTFAMSTRUCT;

typedef struct {
    INT yCurrent;
    HDC hdc;
} PAINTFONTINFO;
typedef PAINTFONTINFO *PPAINTFONTINFO;

//--------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);
```

FontList.c

```
//====================================================================
// FontList - Lists the available fonts in the system
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//====================================================================
#include <windows.h>                    // For all that Windows stuff
#include <commctrl.h>                   // Command bar includes
#include "FontList.h"                   // Program-specific stuff

//--------------------------------------------------------------------
// Global data
//
```

```
const TCHAR szAppName[] = TEXT ("FontList");
HINSTANCE hInst;                        // Program instance handle

FONTFAMSTRUCT ffs[FAMILYMAX];
INT sFamilyCnt = 0;

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_PAINT, DoPaintMain,
    WM_DESTROY, DoDestroyMain,
};

//=====================================================================
//
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
    HWND hwndMain;

    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//---------------------------------------------------------------------
// InitApp - Application initialization
//
```

(continued)

Figure 2-4. *continued*

```
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application main window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = MainWndProc;                   // Callback function
    wc.cbClsExtra = 0;                              // Extra class data
    wc.cbWndExtra = 0;                              // Extra window data
    wc.hInstance = hInstance;                       // Owner handle
    wc.hIcon = NULL;                                // Application icon
    wc.hCursor = NULL;                              // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                        // Menu name
    wc.lpszClassName = szAppName;                   // Window class name

    if (RegisterClass (&wc) == 0) return 1;

    return 0;
}
//----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow) {
    HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    hWnd = CreateWindow (szAppName,                 // Window class
                         TEXT("Font Listing"),// Window title
                         WS_VISIBLE,                // Style flags
                         CW_USEDEFAULT,             // x position
                         CW_USEDEFAULT,             // y position
                         CW_USEDEFAULT,             // Initial width
                         CW_USEDEFAULT,             // Initial height
                         NULL,                      // Parent
                         NULL,                      // Menu, must be null
                         hInstance,                 // Application instance
                         NULL);                     // Pointer to create
                                                    // parameters
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;
```

```
    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//--------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {

    return nDefRC;
}
//====================================================================
// Font callback functions
//
//--------------------------------------------------------------------
// FontFamilyCallback - Callback function that enumerates the font
// families
//
int CALLBACK FontFamilyCallback (CONST LOGFONT *lplf,
                                 CONST TEXTMETRIC *lpntm,
                                 DWORD nFontType, LPARAM lParam) {
    int rc = 1;

    // Stop enumeration if array filled.
    if (sFamilyCnt >= FAMILYMAX)
        return 0;
    // Copy face name of font.
    lstrcpy (ffs[sFamilyCnt++].szFontFamily, lplf->lfFaceName);

    return rc;
}
//--------------------------------------------------------------------
// EnumSingleFontFamily - Callback function that enumerates fonts
//
int CALLBACK EnumSingleFontFamily (CONST LOGFONT *lplf,
                                   CONST TEXTMETRIC *lpntm,
                                   DWORD nFontType, LPARAM lParam) {
    PFONTFAMSTRUCT pffs;

    pffs = (PFONTFAMSTRUCT) lParam;
    pffs->nNumFonts++;    // Increment count of fonts in family
    return 1;
}
```

(continued)

Figure 2-4. *continued*

```
//-----------------------------------------------------------------
// PaintSingleFontFamily - Callback function that draws a font
//
int CALLBACK PaintSingleFontFamily (CONST LOGFONT *lplf,
                                    CONST TEXTMETRIC *lpntm,
                                    DWORD nFontType, LPARAM lParam) {
    PPAINTFONTINFO ppfi;
    TCHAR szOut[256];
    INT nFontHeight, nPointSize;
    HFONT hFont, hOldFont;

    ppfi = (PPAINTFONTINFO) lParam;  // Translate lParam into struct
                                     // pointer.

    // Create the font from the LOGFONT structure passed.
    hFont = CreateFontIndirect (lplf);

    // Select the font into the device context.
    hOldFont = SelectObject (ppfi->hdc, hFont);

    // Compute font size.
    nPointSize = (lplf->lfHeight * 72) /
                 GetDeviceCaps(ppfi->hdc,LOGPIXELSY);

    // Format string and paint on display.
    wsprintf (szOut, TEXT ("%s   Point:%d"), lplf->lfFaceName,
              nPointSize);
    ExtTextOut (ppfi->hdc, 25, ppfi->yCurrent, 0, NULL,
                szOut, lstrlen (szOut), NULL);

    // Compute the height of the default font.
    nFontHeight = lpntm->tmHeight + lpntm->tmExternalLeading;
    // Update new draw point.
    ppfi->yCurrent += nFontHeight;

    // Deselect font and delete.
    SelectObject (ppfi->hdc, hOldFont);
    DeleteObject (hFont);
    return 1;
}
//-----------------------------------------------------------------
// Message handling procedures for MainWindow
//
```

```
//-----------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//-----------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB;
    HDC hdc;
    INT i, rc;

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);

    //Enumerate the available fonts.
    hdc = GetDC (hWnd);
    rc = EnumFontFamilies ((HDC)hdc, (LPTSTR)NULL,
        FontFamilyCallback, 0);

    for (i = 0; i < sFamilyCnt; i++) {
        ffs[i].nNumFonts = 0;
        rc = EnumFontFamilies ((HDC)hdc, ffs[i].szFontFamily,
                               EnumSingleFontFamily,
                               (LPARAM)(PFONTFAMSTRUCT)&ffs[i]);
    }
    ReleaseDC (hWnd, hdc);
    return 0;
}
```

(continued)

Figure 2-4. *continued*

```
//-----------------------------------------------------------------
// DoPaintMain - Process WM_PAINT message for window.
//
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect;
    HDC hdc;
    TEXTMETRIC tm;
    INT nFontHeight, i;
    TCHAR szOut[256];
    PAINTFONTINFO pfi;

    // Adjust the size of the client rect to take into account
    // the command bar height.
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    hdc = BeginPaint (hWnd, &ps);

    // Get the height of the default font.
    GetTextMetrics (hdc, &tm);
    nFontHeight = tm.tmHeight + tm.tmExternalLeading;

    // Initialize struct that is passed to enumerate function.
    pfi.yCurrent = rect.top;
    pfi.hdc = hdc;
    for (i = 0; i < sFamilyCnt; i++) {

        // Format output string and paint font family name.
        wsprintf (szOut, TEXT("Family: %s   "),
                  ffs[i].szFontFamily);
        ExtTextOut (hdc, 5, pfi.yCurrent, 0, NULL,
                    szOut, lstrlen (szOut), NULL);
        pfi.yCurrent += nFontHeight;

        // Enumerate each family to draw a sample of that font.
        EnumFontFamilies ((HDC)hdc, ffs[i].szFontFamily,
                          PaintSingleFontFamily,
                          (LPARAM)&pfi);
    }
    EndPaint (hWnd, &ps);
    return 0;
}
```

```
//-----------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
   PostQuitMessage (0);
   return 0;
}
```

Enumerating the different fonts begins when the application is processing the WM_CREATE message in *OnCreateMain*. Here, *EnumFontFamilies* is called with the *FontFamily* field set to NULL so that each family will be enumerated. The callback function is *FontFamilyCallback*, where the name of the font family is copied into an array of strings.

The remainder of the work is performed during the processing of the WM_PAINT message. The *OnPaintMain* function begins with the standard litany of getting the size of the area below the command bar and calling *BeginPaint*, which returns the handle to the device context of the window. *GetTextMetrics* is then called to compute the row height of the default font. A loop is then entered in which *EnumerateFontFamilies* is called for each family name that had been stored during the enumeration process in *OnCreateMain*. The callback process for this callback sequence is somewhat more complex than the code we've seen so far.

The *PaintSingleFontFamily* callback procedure, used in the enumeration of the individual fonts, employs the *lParam* parameter to retrieve a pointer to a PAINTFONTINFO structure defined in FontList.h. This structure contains the current vertical drawing position as well as the handle to the device context. By using the *lParam* pointer, FontList avoids having to declare global variables to communicate with the callback procedure.

The callback procedure next creates the font using the pointer to LOGFONT that was passed to the callback procedure. The new font is then selected into the device context, while the handle to the previously selected font is retained in *hOldFont*. The point size of the enumerated font is computed using the inverse of the equation mentioned earlier in the chapter on page 49. The callback procedure then produces a line of text showing the name of the font family along with the point size of this particular font. Instead of using *DrawText*, the callback uses a different text output function:

```
BOOL ExtTextOut (HDC hdc, int X, int Y, UINT fuOptions,
                 const RECT *lprc, LPCTSTR lpString,
                 UINT cbCount, const int *lpDx);
```

The *ExtTextOut* function has a few advantages over *DrawText* in this situation. First, *ExtTextOut* tends to be faster for drawing single lines of text. Second, instead of formatting the text inside a rectangle, *x* and *y* starting coordinates are passed, specifying the upper left corner of the rectangle where the text will be drawn. The *rect* parameter that's passed is used as a clipping rectangle, or if the background mode is opaque, the area where the background color is drawn. This rectangle parameter can be NULL if you don't want any clipping or opaquing. The next two parameters are the text and the character count. The last parameter, *ExtTextOut,* allows an application to specify the horizontal distance between adjacent character cells. In our case, this parameter is set to NULL also, which results in the default separation between characters.

Windows CE differs from other versions of Windows in having only these two text drawing functions for displaying text. Most of what you can do with the other text functions typically used in other versions of Windows, such as *TextOut* and *TabbedTextOut*, can be emulated using either *DrawText* or *ExtTextOut*. This is one of the areas in which Windows CE has broken with earlier versions of Windows, sacrificing backward compatibility to achieve a smaller operating system.

After displaying the text, the function computes the height of the line of text just drawn using the combination of *tmHeight* and *tmExternalLeading* that was provided in the passed TEXTMETRIC structure. The new font is then deselected using a second call to *SelectObject*, this time passing the handle to the font that was the original selected font. The new font is then deleted using *DeleteObject*. Finally, the callback function returns a nonzero value to indicate to Windows that it is okay to make another call to the *enumerate* callback.

Figure 2-5 shows the FontListing window. Notice that the font names are displayed in that font and that each font has a specific set of available sizes.

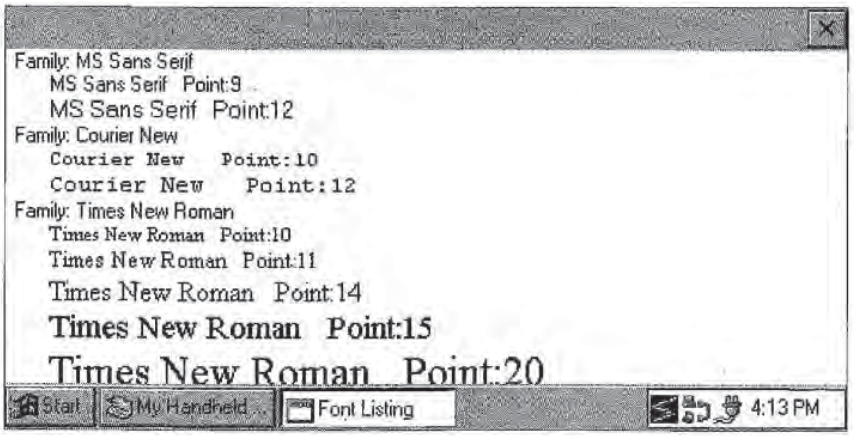


Figure 2-5. *The FontList window shows some of the available fonts for a Handheld PC.*

Unfinished business

If you look closely at Figure 2-5, you'll notice a problem with the display. The list of fonts just runs off the bottom edge of the FontList window. At this point in a book covering the desktop versions of Windows, the author might add a window style flag for a vertical scroll bar and a small amount of code, and magically, the program would have a scrollable window. But if you do that to a Windows CE main window, you end up with the look shown in Figure 2-6.

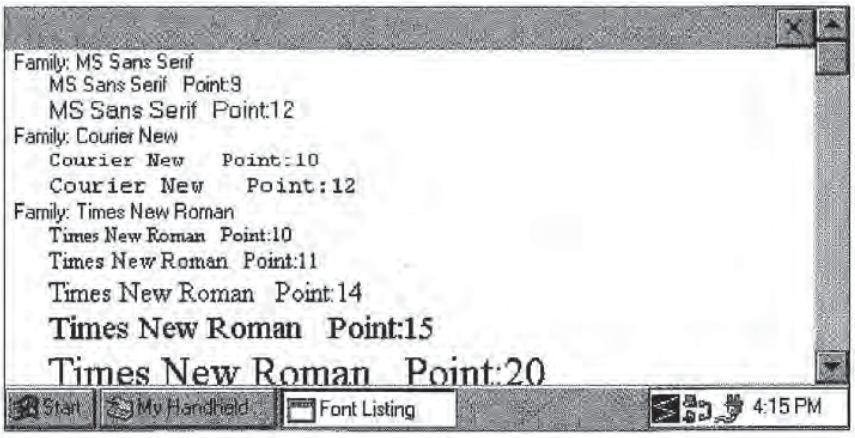


Figure 2-6. *The FontList window with a scrollbar attached to the main window.*

Notice how the scroll bar extends past the right side of the command bar up to the top of the window. The scroll bar should stop below the command bar and the command bar should extend to the right edge of the window. The problem is that the command bar lies in the client area of the window, and the default scroll bar style provided by all Windows operating systems places the scroll bar outside the client area, in the nonclient space along the edge of the window. The solution to this problem involves creating a child window inside our main window and letting it do the scrolling. But since I'll provide a complete explanation of child windows in Chapter 4, I'll hold off describing how to properly implement a scroll bar until then.

BITMAPS

Bitmaps are graphical objects that can be used to create, draw, manipulate, and retrieve images in a device context. Bitmaps are everywhere within Windows, from the little Windows logo on the Start button to the Close button on the command bar. Think of a bitmap as a picture composed of an array of pixels that can be painted onto the screen. Like any picture, a bitmap has height and width. It also has a method for determining what color or colors it uses. Finally, a bitmap has an array of bits that describe each pixel in the bitmap.

Historically, bitmaps under Windows have been divided into two types; *device dependent bitmaps* (DDBs) and *device independent bitmaps* (DIBs). DDBs are bitmaps that are tied to the characteristics of a specific DC and can't easily be rendered on DCs with different characteristics. DIBs, on the other hand, are independent of any device and therefore must carry around enough information so that they can be rendered accurately on any device.

Windows CE contains many of the bitmap functions available in other versions of Windows. The differences include a new four-color bitmap format not supported anywhere but on Windows CE and a different method for manipulating DIBs.

Device Dependent Bitmaps

A device dependent bitmap can be created with this function:

```
HBITMAP CreateBitmap (int nWidth, int nHeight, UINT cPlanes,
                      UINT cBitsPerPel, CONST VOID *lpvBits);
```

The *nWidth* and *nHeight* parameters indicate the dimensions of the bitmap. The *cPlanes* parameter is an historical artifact from the days when display hardware implemented each color within a pixel in a different hardware plane. For Windows CE, this parameter must be set to 1. The *cBitspPerPel* parameter indicates the number of bits used to describe each pixel. The number of colors is 2 to the power of the *cBitspPerPel* parameter. Under Windows CE, the allowable values are 1, 2, 4, 8, 16, and 24. As I said, the four-color bitmap is unique to Windows CE and isn't supported under other Windows platforms, including the Windows CE emulator that runs on top of Windows NT.

The final parameter is a pointer to the bits of the bitmap. Under Windows CE, the bits are always arranged in a packed pixel format; that is, each pixel is stored as a series of bits within a byte, with the next pixel starting immediately after the first. The first pixel in the array of bits is the pixel located in the upper left corner of the bitmap. The bits continue across the top row of the bitmap, then across the second row, and so on. Each row of the bitmap must be double-word (4-byte) aligned. If any pad bytes are required at the end of a row to align the start of the next row, they should be set to 0. Figure 2-7 illustrates this scheme, showing a 126-by-64 pixel bitmap with 8 bits per pixel.

The function

```
HBITMAP CreateCompatibleBitmap (HDC hdc, int nWidth, int nHeight);
```

creates a bitmap whose format is compatible with the device context passed to the function. So, if the device context is a four-color DC, the resulting bitmap is a four-

color bitmap as well. This function comes in handy when you're manipulating images on the screen because it makes it easy to produce a blank bitmap that's directly color compatible with the screen.

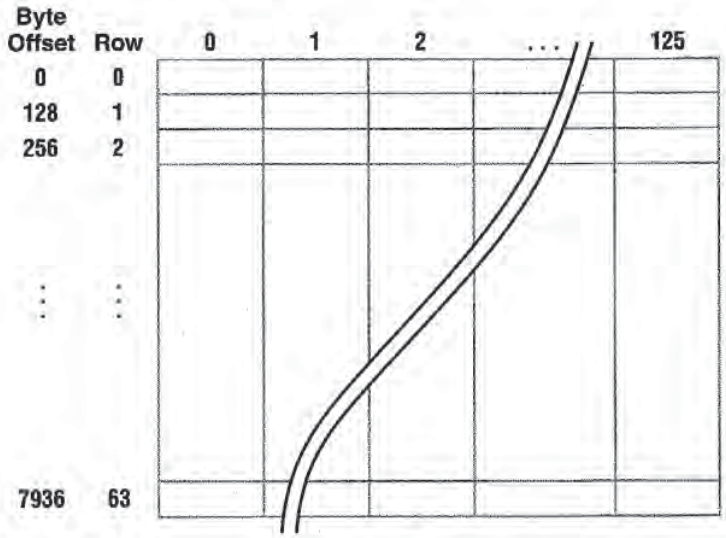


Figure 2-7. *Layout of bytes within a bitmap.*

Device Independent Bitmaps

The fundamental difference between DIBs and their device dependent cousins is that the image stored in a DIB comes with its own color information. Almost every bitmap file since Windows 3.0, which used the files with the BMP extension, contains information that can be directly matched with the information needed to create a DIB in Windows.

In the early days of Windows, it was a rite of passage for a programmer to write a routine that manually read a DIB file and converted the data to a bitmap. These days, the same arduous task can be accomplished with the following function, unique to Windows CE:

```
HBITMAP SHLoadDIBitmap (LPCTSTR szFileName);
```

It loads a bitmap directly from a bitmap file and provides a handle to the bitmap. In Windows NT and Windows 98, the same process can be accomplished with *LoadImage* using the LR_LOADFROMFILE flag, but this flag isn't supported under the Windows CE implementation of *LoadImage*.

DIB Sections

While Windows CE makes it easy to load a bitmap file, sometimes you must read what is on the screen, manipulate it, and redraw the image back to the screen. This is another case in which DIBs are better than DDBs. While the bits of a device dependent bitmap are obtainable, the format of the buffer is directly dependent on the screen format. By using a DIB, or more precisely, something called a DIB section, your program can read the bitmap into a buffer that has a predefined format without worrying about the format of the display device.

While Windows has a number of DIB creation functions that have been added over the years since Windows 3.0, Windows CE carries over only one DIB section function from Windows NT and Windows 98. Here it is:

```
HBITMAP CreateDIBSection (HDC hdc, const BITMAPINFO *pbmi,
                          UINT iUsage, void *ppvBits,
                          HANDLE hSection, DWORD dwOffset);
```

Because it's a rather late addition to the Win32 API, DIB sections might be new to Windows programmers. DIB Sections were invented to improve the performance of applications on Windows NT that directly manipulated bitmaps. In short, a DIB section allows a programmer to select a DIB in a device context while still maintaining direct access to the bits that compose the bitmap. To achieve this, a DIB section associates a memory DC with a buffer that also contains the bits of that DC. Because the image is mapped to a DC, other graphics calls can be made to modify the image. At the same time, the raw bits of the DC, in DIB format, are available for direct manipulation. While the improved performance is all well and good on NT, the relevance to the Windows CE programmer is the ease in which an application can work with bitmaps and manipulate their contents.

The parameters of this call lead off with the pointer to a BITMAPINFO structure. This structure describes the layout and color composition of a device independent bitmap and is a combination of a BITMAPINFOHEADER structure and an array of RGBQUAD values that represent the palette of colors used by the bitmap.

The BITMAPINFOHEADER structure is defined as the following:

```
typedef struct tagBITMAPINFOHEADER{
    DWORD biSize;
    LONG biWidth;
    LONG biHeight;
    WORD biPlanes;
    WORD biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
```

```
    LONG biXPelsPerMeter;
    LOG biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;
```

As you can see, this structure contains much more information than just the parameters passed to *CreateBitmap*. The first field is the size of the structure and must be filled in by the calling program to differentiate this structure from the similar BITMAPCOREINFOHEADER structure that's a holdover from the OS/2 presentation manager. The *biWidth*, *biHeight*, *biPlanes*, and *biBitCount* fields are similar to their like-named parameters to the *CreateBitmap* call—with one exception. The sign of the *biHeight* field specifies the organization of the bit array. If *biHeight* is negative, the bit array is organized in a top-down format, as is *CreateBitmap*. If *biHeight* is positive, the array is organized in a bottom-up format, in which the bottom row of the bitmap is defined by the first bits in the array. As with the *CreateBitmap* call, the *biPlanes* field must be set to 1.

The *biCompression* field specifies the compression method used in the bit array. Under Windows CE, the only allowable setting for this field is BI_RGB, indicating that the buffer isn't compressed. The *biSizeImage* parameter is used to indicate the size of the bit array; when used with BI_RGB, however, the *biSizeImage* field can be set to 0, meaning the array size is computed using the dimensions and bits per pixel information provided in the BITMAPINFOHEADER structure.

The *biXPelsPerMeter* and *biYPelsPerMeter* fields provide information to accurately scale the image. For *CreateDIBSection*, however, these parameters can be set to 0. The *biClrUsed* parameter specifies the number of colors in the palette that are actually used. In a 256-color image, the palette will have 256 entries, but the bitmap itself might need only 100 or so distinct colors. This field helps the palette manager, the part of the Windows that manages color matching, to match the colors in the system palette with the colors required by the bitmap. The *biClrImportant* field further defines the colors that are *really* required as opposed to those that are used. For most color bitmaps, these two fields are set to 0, indicating that all colors are used and that all colors are important.

As I mentioned above, an array of RGBQUAD structures immediately follows the BITMAPINFOHEADER structure. The RGBQUAD structure is defined as follows:

```
typedef struct tagRGBQUAD { /* rgbq */
    BYTE rgbBlue;
    BYTE rgbGreen;
    BYTE rgbRed;
    BYTE rgbReserved;
} RGBQUAD;
```

This structure allows for 256 shades of red, green, and blue. While almost any shade of color can be created using this structure, the color that's actually rendered on the device will, of course, be limited by what the device can display.

The array of RGBQUAD structures, taken as a whole, describe the palette of the DIB. The palette is the list of colors in the bitmap. If a bitmap has a palette, each entry in the bitmap array contains not colors, but an index into the palette that contains the color for that pixel. While redundant on a monochrome bitmap, the palette is quite important when rendering color bitmaps on color devices. For example a 256 color bitmap has one byte for each pixel, but that byte points to a 24 bit value that represents equal parts red, green, and blue colors. So, while a 256-color bitmap can only contain 256 distinct colors, each of those colors can be one of 16 million colors rendered using the 24-bit palette entry. For convenience in a 32-bit world, each palette entry, while containing only 24 bits of color information, is padded out to a 32-bit wide entry—hence the name of the data type: RGBQUAD.

Of the remaining four *CreateDIBSection* parameters, only two are used under Windows CE. The *iUsage* parameter indicates how the colors in the palette are represented. For Windows CE, this field must be set to DIB_RGB_COLORS. The *ppvBits* parameter is a pointer to a variable that receives the pointer to the bitmap bits that compose the bitmap image. The final two parameters, *hSection* and *dwOffset*, aren't supported under Windows CE and must be set to 0. In other versions of Windows, they allow the bitmap bits to be specified by a memory mapped file. While Windows CE does support memory mapped files, they aren't supported by *CreateDIBSection*.

Drawing Bitmaps

Creating and loading bitmaps is all well and good, but there's not much point to it unless the bitmaps you create can be rendered on the screen. Drawing a bitmap isn't as straightforward as you might think. Before a bitmap can be drawn in a screen DC, it must be selected into a DC and then copied over to the screen device context. While this process sounds convoluted, there is rhyme to this reason.

The process of selecting a bitmap into a device context is similar to selecting a logical font into a device context; it converts the ideal to the actual. Just as Windows finds the best possible match to a requested font, the bitmap selection process must match the available colors of the device to the colors requested by a bitmap. Only after this is done can the bitmap be rendered on the screen. To help with this intermediate step, Windows provides a shadow type of DC, a *memory device context*.

To create a memory device context, use this function:

```
HDC CreateCompatibleDC (HDC hdc);
```

This function creates a memory DC that's compatible with the current screen DC. Once created, the source bitmap is selected into this memory DC using the same *SelectObject* function you used to select in a logical font. Finally, the bitmap is copied from the memory DC to the screen DC using one of the blit functions, *BitBlt* or *StretchBlt*.

The workhorse of bitmap functions is the following:

```
BOOL BitBlt (HDC hdcDest, int nXDest, int nYDest, int nWidth,
             int nHeight, HDC hdcSrc, int nXSrc,  int nYSrc,
             DWORD dwRop);
```

Fundamentally, the *BitBlt* function, pronounced *bit blit*, is just a fancy *memcopy* function, but since it operates on device contexts, not memory, it's something far more special. The first parameter is a handle to the destination device context—the DC to which the bitmap is to be copied. The next four parameters specify the location and size of the destination rectangle where the bitmap is to end up. The next three parameters specify the handle to the source device context and the location within that DC of the upper left corner of the source image.

The final parameter, *dwRop*, specifies how the image is to be copied from the source to the destination device contexts. The ROP code defines how the source bitmap and the current destination are combined to produce the final image. The ROP code for a simple copy of the source image is SRCCOPY. The ROP code for combining the source image with the current destination is SRCPAINT. Copying a logically inverted image, essentially a negative of the source image, is accomplished using SRCINVERT. Some ROP codes also combine the currently selected brush into the equation to compute the resulting image. A large number of ROP codes are available, too many for me to cover here. For a complete list, check out the Windows CE programming documentation.

The following code fragment sums up how to paint a bitmap:

```
// Create a DC that matches the device.
hdcMem = CreateCompatibleDC (hdc);

// Select the bitmap into the compatible device context.
hOldSel = SelectObject (hdcMem, hBitmap);

// Get the bitmap dimensions from the bitmap.
GetObject (hBitmap, sizeof (BITMAP), &bmp);
// Copy the bitmap image from the memory DC to the screen DC.
BitBlt (hdc, rect.left, rect.top, bmp.bmWidth, bmp.bmHeight,
        hdcMem, 0, 0, SRCCOPY);
```

(continued)

```
// Restore original bitmap selection and destroy the memory DC.
SelectObject (hdcMem, hOldSel);
DeleteDC (hdcMem);
```

The memory device context is created and the bitmap to be painted is selected into that DC. Since you might not have stored the dimensions of the bitmap to be painted, the routine makes a call to *GetObject*. *GetObject* returns information about a graphics object, in this case, a bitmap. Information about fonts and other graphic objects can be queried using this useful function. Next, *BitBlt* is used to copy the bitmap into the screen DC. To clean up, the bitmap is deselected from the memory device context and the memory DC is deleted using *DeleteDC*. Don't confuse *DeleteDC* with *ReleaseDC*, which is used to free a display DC. *DeleteDC* should be paired only with *CreateCompatibleDC* and *ReleaseDC* should be paired only with *GetDC* or *GetWindowDC*.

Instead of merely copying the bitmap, stretch or shrink it using this function:

```
BOOL StretchBlt (HDC hdcDest, int nXOriginDest, int nYOriginDest,
                 int nWidthDest, int nHeightDest, HDC hdcSrc,
                 int nXOriginSrc, int nYOriginSrc, int nWidthSrc,
                 int nHeightSrc, DWORD dwRop);
```

The parameters in *StretchBlt* are the same as those used in *BitBlt*, with the exception that now the width and height of the source image can be specified. Here again, the ROP codes specify how the source and destination are combined to produce the final image.

Windows CE 2.0 added a new, and quite handy, bitmap function. It is

```
BOOL TransparentImage (HDC hdcDest, LONG DstX, LONG DstY, LONG DstCx,
                       LONG DstCy, HANDLE hSrc, LONG SrcX, LONG SrcY,
                       LONG SrcCx, LONG SrcCy, COLORREF TransparentColor);
```

This function is similar to *StretchBlt* with two very important exceptions. First, you can specify a color in the bitmap to be the transparent color. When the bitmap is copied to the destination, the pixels in the bitmap that are the transparent color are not copied. The second difference is that the *hSrc* parameter can either be a device context or a handle to a bitmap, which allows you to bypass the requirement to select the source image into a device context before rendering it on the screen.

As in other versions of Windows, Windows CE supports two other blit functions: *PatBlt* and *MaskBlt*. The *PatBlt* function combines the currently selected brush with the current image in the destination DC to produce the resulting image. I cover brushes later in this chapter. The *MaskBlt* function is similar to *BitBlt* but encompasses a masking image that provides the ability to draw only a portion of the source image onto the destination DC.

LINES AND SHAPES

One of the areas in which Windows CE provides substantially less functionality than other versions of Windows is in the primitive line-drawing and shape-drawing functions. Gone are the *Chord*, *Arc*, and *Pie* functions that created complex circular shapes. Gone too is the concept of *current point*. Other versions of Windows track a current point, which is then used as the starting point for the next drawing command. So drawing a series of connected lines and curves by calling *MoveTo* to move the current point followed by calls to *LineTo*, *ArcTo*, *PolyBezierTo* and so forth is no longer possible. But even with the loss of a number of graphic functions, Windows CE still provides the essential functions necessary to draw lines and shapes.

Lines

Drawing one or more lines is as simple as a call to

```
BOOL Polyline (HDC hdc, const POINT *lppt, int cPoints);
```

The second parameter is a pointer to an array of POINT structures that are defined as the following:

```
typedef struct tagPOINT {
    LONG x;
    LONG y;
} POINT;
```

Each *x* and *y* combination describes a pixel from the upper left corner of the screen. The third parameter is the number of point structures in the array. So to draw a line from (0, 0) to (50, 100), the code would look like this:

```
POINTS pts[2];

pts[0].x = 0;
pts[0].y = 0;
pts[1].x = 50;
pts[1].y = 100;
PolyLine (hdc, &pts, 2);
```

Just as in the early text examples, this code fragment makes a number of assumptions about the default state of the device context. For example, just what does the line drawn between (0,0) and (50, 100) look like? What is its width and its color, and is it a solid line? All versions of Windows, including Windows CE, allow these parameters to be specified.

The tool for specifying the appearance of lines and the outline of shapes is called, appropriately enough, a *pen*. A pen is another GDI object and, like the others described in this chapter, is created, selected into a device context, used, deselected, and then destroyed. Among other stock GDI objects, stock pens can be retrieved using the following code:

```
HGDIOBJ GetStockObject (int fnObject);
```

All versions of Windows provide three stock pens, each 1 pixel wide. The stock pens come in 3 colors: white, black, and null. Using *GetStockObject*, the call to retrieve one of those pens employs the parameters WHITE_PEN, BLACK_PEN, and NULL_PEN respectively. Unlike standard graphic objects created by applications, stock objects should never be deleted by the application. Instead, the application should simply deselect the pen from the device context when it's no longer needed.

To create a custom pen under Windows, two functions are available. The first is this:

```
HPEN CreatePen ( int fnPenStyle, int nWidth, COLORREF crColor);
```

The *fnPenStyle* parameter specifies the appearance of the line to be drawn. For example, the PS_DASH flag can be used to create a dashed line. The *nWidth* parameter specifies the width of the pen. Finally, the *crColor* parameter specifies the color of the pen. The *crColor* parameter is typed as COLORREF, which under Windows CE 2.0 is an RGB value. The RGB macro is as follows:

```
COLORREF RGB (BYTE bRed, BYTE bGreen, BYTE bBlue);
```

So to create a solid red pen, the code would look like this:

```
hPen = CreatePen (PS_SOLID, 1, RGB (0xff, 0, 0));
```

The other pen creation function is the following:

```
HPEN CreatePenIndirect (const LOGPEN *lplgpn);
```

where the logical pen structure LOGPEN is defined as

```
typedef struct tagLOGPEN {
    UINT lopnStyle;
    POINT lopnWidth;
    COLORREF lopnColor;
} LOGPEN;
```

CreatePenIndirect provides the same parameters to Windows, in a different form. To create the same 1-pixel-wide red pen with *CreatePenIndirect*, the code would look like this:

```
LOGPEN lp;
HPEN hPen;

lp.lopnStyle = PS_SOLID;
lp.lopnWidth.x = 1;
lp.lopnWidth.y = 1;
lp.lopnColor = RGB (0xff, 0, 0);

hPen = CreatePenIndirect (&lp);
```

Windows CE devices don't support complex pens such as wide (more than one pixel wide), dashed lines. To determine what's supported, our old friend *GetDeviceCaps* comes into play, taking LINECAPS as the second parameter. Refer to the Windows CE documentation for the different flags returned by this call.

Shapes

Lines are useful but Windows also provides functions to draw shapes, both filled and unfilled. Here, Windows CE does a good job supporting most of the functions familiar to Windows programmers. The *Rectangle*, *RoundRect*, *Ellipse*, and *Polygon* functions are all supported.

Brushes

Before I can talk about shapes such as rectangles and ellipses I need to describe another GDI object that I've only mentioned briefly before now, called a *brush*. A brush is a small 8-by-8 bitmap used to fill shapes. It's also used by Windows to fill the background of a client window. Windows CE provides a number of stock brushes and also the ability to create a brush from an application-defined pattern. A number of stock brushes, each a solid color, can be retrieved using *GetStockObject*. Among the brushes available is one for each of the grays of a four grayscale display: white, light gray, dark gray, and black.

To create solid color brushes, the function to call is the following:

```
HBRUSH CreateSolidBrush (COLORREF crColor);
```

This function isn't really necessary when you're writing an application for a four-color Windows CE device because those four solid brushes can be retrieved with the *GetStockObject* call. For higher color devices however, the *crColor* parameter can be generated using the RGB macro.

To create custom pattern brushes, Windows CE supports the Win32 function:

```
HBRUSH CreateDIBPatternBrushPt (const void *lpPackedDIB,
                                UINT iUsage);
```

The first parameter to this function is a pointer to a DIB in *packed* format. This means that the pointer points to a buffer that contains a BITMAPINFO structure immediately followed by the bits in the bitmap. Remember that a BITMAPINFO structure is actually a BITMAPINFOHEADER structure followed by a palette in RGBQUAD format, so the buffer contains everything necessary to create a DIB—that is, bitmap information, a palette, and the bits to the bitmap. The second parameter must be set to DIB_RGB_COLORS for Windows CE applications. This setting indicates that the palette specified contains RGBQUAD values in each entry. The complimentary flag, DIB_PAL_COLORS, used in other versions of Windows isn't supported in Windows CE.

The *CreateDIBPatternBrushPt* function is more important under Windows CE because the hatched brushes, supplied under other versions of Windows by the *CreateHachBrush* function, aren't supported under Windows CE. Hatched brushes are brushes composed of any combination of horizontal, vertical, or diagonal lines. Ironically, they're particularly useful with grayscale displays because you can use them to accentuate different areas of a chart with different hatch patterns. These brushes, however, can be reproduced by using *CreateDIBPatternBrushPt* and the proper bitmap patterns. The Shapes code example, later in the chapter, demonstrates a method for creating hatched brushes under Windows CE.

By default, the brush origin will be in the upper left corner of the window. This isn't always what you want. Take, for example, a bar graph where the bar filled with a hatched brush fills a rectangle from (100, 100) to (125, 220). Since this rectangle isn't divisible by 8 (brushes being 8 by 8 pixels square), the upper left corner of the bar will be filled with a partial brush that might not look pleasing to the eye.

To avoid this situation, you can move the origin of the brush so that each shape can be drawn with the brush aligned correctly in the corner of the shape to be filled. The function available for this remedy is the following:

```
BOOL SetBrushOrgEx (HDC hdc, int nXOrg, int nYOrg, LPPOINT lppt);
```

The *nXOrg* and *nYOrg* parameters allow the origin to be set between 0 and 7 so that you can position the origin anywhere in the 8-by-8 space of the brush. The *lppt* parameter is filled with the previous origin of the brush so that you can restore the previous origin if necessary.

Rectangles

The rectangle function draws either a filled or a hollow rectangle; the function is defined as the following:

```
BOOL Rectangle (HDC hdc, int nLeftRect, int nTopRect,
                int nRightRect, int nBottomRect);
```

The function uses the currently selected pen to draw the outline of the rectangle and the current brush to fill the interior. To draw a hollow rectangle, select the null brush into the device context before calling *Rectangle*.

The actual pixels drawn for the border are important to understand. Say we're drawing a 5-by-7 rectangle at 0, 0. The function call would look like this:

```
Rectangle (0, 0, 5, 7);
```

Assuming that the selected pen was 1 pixel wide, the resulting rectangle would look like the one shown in Figure 2-8.

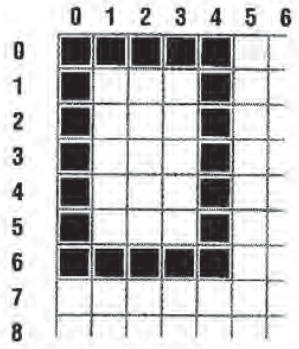


Figure 2-8. *Expanded view of a rectangle drawn with the* Rectangle *function.*

Notice how the right edge of the drawn rectangle is actually drawn in column 4 and that the bottom edge is drawn on row 6. This is standard Windows practice. The rectangle is drawn inside the right and bottom boundary specified for the *Rectangle* function. If the selected pen is wider than one pixel, the right and bottom edges are drawn with the pen centered on the bounding rectangle. (Other versions of Windows support the PS_INSIDEFRAME pen style that forces the rectangle to be drawn inside the frame regardless of the pen width.)

Circles and ellipses

Circles and ellipses can be drawn with this function:

```
BOOL Ellipse (HDC hdc, int nLeftRect, int nTopRect,
              int nRightRect, int nBottomRect);
```

The ellipse is drawn using the rectangle passed as a bounding rectangle, as shown in Figure 2-9. As with the *Rectangle* function, while the interior of the ellipse is filled with the current brush, the outline is drawn with the current pen.

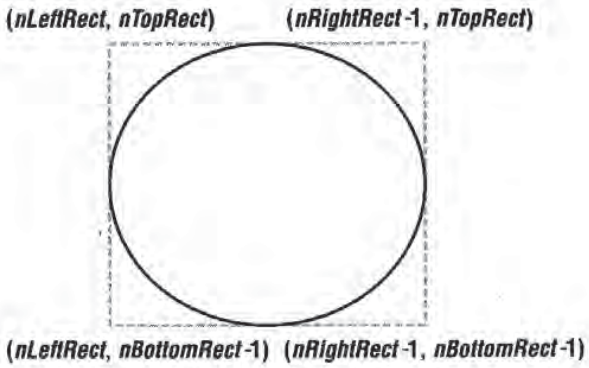


Figure 2-9. *The ellipse is drawn within the bounding rectangle passed to the* Ellipse *function.*

Round rectangles

The *RoundRect* function,

```
BOOL RoundRect (HDC hdc, int nLeftRect, int nTopRect,
                int nRightRect, int nBottomRect,
                int nWidth, int nHeight);
```

draws a rectangle with rounded corners. The roundedness of the corners is defined by the last two parameters that specify the width and height of the ellipse used to round the corners, as shown in Figure 2-10. Specifying the ellipse height and width enables your program to draw identically symmetrical rounded corners. Shortening the ellipse height flattens out the sides of the rectangle, while shortening the width of the ellipse flattens the top and bottom of the rectangle.

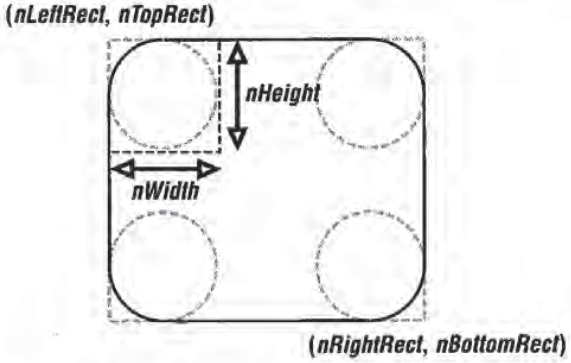


Figure 2-10. *The height and width of the ellipse define the round corners of the rectangle drawn by* RoundRect.

Polygons

Finally, the *Polygon* function,

```
BOOL Polygon (HDC hdc, const POINT *lpPoints, int nCount);
```

draws a many-sided shape. The second parameter is a pointer to an array of point structures defining the points that delineate the polygon. The resulting shape has one more side than the number of points because the function automatically completes the last line of the polygon by connecting the last point with the first. Under Windows CE 1.0, this function is limited to producing convex polygons.

The Shapes Example Program

The Shapes program, shown in Figure 2-11, demonstrates a number of these functions. In Shapes, five figures are drawn, each filled with a different brush:

Shapes.h

```
//------------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//------------------------------------------------------------------------
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

//------------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                             // Structure associates
    UINT Code;                                  // messages
                                                // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                              // Structure associates
    UINT Code;                                  // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);     // function.
};

//------------------------------------------------------------------------
// Generic defines used by application
#define  IDC_CMDBAR 1                           // Command bar ID
//------------------------------------------------------------------------
// Defines used by MyCreateHatchedBrush
//
```

Figure 2-11. *The Shapes program.* *(continued)*

Figure 2-11. *continued*

```
typedef struct {
    BITMAPINFOHEADER bmi;
    COLORREF dwPal[2];
    BYTE bBits[64];
} BRUSHBMP;

#define HS_HORIZONTAL       0       /* ----- */
#define HS_VERTICAL         1       /* ||||| */
#define HS_FDIAGONAL        2       /* \\\\\ */
#define HS_BDIAGONAL        3       /* ///// */
#define HS_CROSS            4       /* +++++ */
#define HS_DIAGCROSS        5       /* xxxxx */


//-------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);
```

Shapes.c

```
//====================================================================
// Shapes- Brush and shapes demo for Windows CE
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//====================================================================
#include <windows.h>                // For all that Windows stuff
#include <commctrl.h>               // Command bar includes
#include "shapes.h"                 // Program-specific stuff
```

```
//-----------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("Shapes");
HINSTANCE hInst;                        // Program instance handle

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_PAINT, DoPaintMain,
    WM_DESTROY, DoDestroyMain,
};

//=================================================================
//
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
    HWND hwndMain;

    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance(hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//-----------------------------------------------------------------
// InitApp - Application initialization
//
```

(continued)

Figure 2-11. *continued*

```
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application main window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = MainWndProc;                   // Callback function
    wc.cbClsExtra = 0;                              // Extra class data
    wc.cbWndExtra = 0;                              // Extra window data
    wc.hInstance = hInstance;                       // Owner handle
    wc.hIcon = NULL;                                // Application icon
    wc.hCursor = NULL;                              // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                        // Menu name
    wc.lpszClassName = szAppName;                   // Window class name

    if (RegisterClass (&wc) == 0) return 1;

    return 0;
}
//-----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow){
    HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    hWnd = CreateWindow (szAppName,                 // Window class
                         TEXT("Shapes"),            // Window title
                         WS_VISIBLE,                // Style flags
                         CW_USEDEFAULT,             // x position
                         CW_USEDEFAULT,             // y position
                         CW_USEDEFAULT,             // Initial width
                         CW_USEDEFAULT,             // Initial height
                         NULL,                      // Parent
                         NULL,                      // Menu, must be null
                         hInstance,                 // Application instance
                         NULL);                     // Pointer to create
                                                    // parameters
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
```

```
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {

    return nDefRC;
}
//======================================================================
// Message handling procedures for MainWindow
//

//----------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//----------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB;

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    return 0;
}
```

(continued)

Figure 2-11. *continued*

```
//-----------------------------------------------------------------
// MyCreateHachBrush - Creates hatched brushes
//
HBRUSH MyCreateHachBrush (INT fnStyle, COLORREF clrref) {
    BRUSHBMP brbmp;
    BYTE *pBytes;
    int i;
    DWORD dwBits[6][2] = {
        {0x000000ff,0x00000000}, {0x10101010,0x10101010},
        {0x01020408,0x10204080}, {0x80402010,0x08040201},
        {0x101010ff,0x10101010}, {0x81422418,0x18244281},
    };

    if ((fnStyle < 0) || (fnStyle > dim(dwBits)))
        return 0;
    memset (&brbmp, 0, sizeof (brbmp));

    brbmp.bmi.biSize = sizeof (BITMAPINFOHEADER);
    brbmp.bmi.biWidth = 8;
    brbmp.bmi.biHeight = 8;
    brbmp.bmi.biPlanes = 1;
    brbmp.bmi.biBitCount = 1;
    brbmp.bmi.biClrUsed = 2;
    brbmp.bmi.biClrImportant = 2;

    // Initialize the palette of the bitmap.
    brbmp.dwPal[0] = PALETTERGB(0xff,0xff,0xff);
    brbmp.dwPal[1] = PALETTERGB((BYTE)((clrref >> 16) & 0xff),
                                (BYTE)((clrref >> 8) & 0xff),
                                (BYTE)(clrref & 0xff));

    // Write the hatch data to the bitmap.
    pBytes = (BYTE *)&dwBits[fnStyle];
    for (i = 0; i < 8; i++)
        brbmp.bBits[i*4] = *pBytes++;

    // Return the handle of the brush created.
    return CreateDIBPatternBrushPt (&brbmp, DIB_RGB_COLORS);
}
//-----------------------------------------------------------------
// DoPaintMain - Process WM_PAINT message for window.
//
//#define ENDPOINTS 32
#define ENDPOINTS 64
```

```
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect;
    HDC hdc;
    POINT ptArray[ENDPOINTS];
    HBRUSH hBr, hOldBr;
    TCHAR szText[128];

    // Adjust the size of the client rect to take into account
    // the command bar height.
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    hdc = BeginPaint (hWnd, &ps);

    // Draw rectangle.
    hBr = GetStockObject (BLACK_BRUSH);
    hOldBr = SelectObject (hdc, hBr);
    Rectangle (hdc, 50, 50, 125, 150);
    SelectObject (hdc, hOldBr);

    // Draw ellipse.
    hBr = GetStockObject (DKGRAY_BRUSH);
    hOldBr = SelectObject (hdc, hBr);
    Ellipse (hdc, 150, 50, 225, 150);
    SelectObject (hdc, hOldBr);

    // Draw round rectangle.
    hBr = GetStockObject (LTGRAY_BRUSH);
    hOldBr = SelectObject (hdc, hBr);
    RoundRect (hdc, 250, 50, 325, 150, 30, 30);
    SelectObject (hdc, hOldBr);

    // Draw hexagon using Polygon.
    hBr = GetStockObject (WHITE_BRUSH);
    hOldBr = SelectObject (hdc, hBr);
    ptArray[0].x = 387;
    ptArray[0].y = 50;
    ptArray[1].x = 350;
    ptArray[1].y = 75;
    ptArray[2].x = 350;
    ptArray[2].y = 125;
```

(continued)

Figure 2-11. *continued*

```
    ptArray[3].x = 387;
    ptArray[3].y = 150;
    ptArray[4].x = 425;
    ptArray[4].y = 125;
    ptArray[5].x = 425;
    ptArray[5].y = 75;

    Polygon (hdc, ptArray, 6);
    SelectObject (hdc, hOldBr);

    hBr = MyCreateHachBrush (HS_DIAGCROSS, RGB (0, 0, 0));
    hOldBr = SelectObject (hdc, hBr);
    Rectangle (hdc, 50, 165, 425, 210);
    SelectObject (hdc, hOldBr);
    DeleteObject (hBr);

    SetBkMode (hdc, OPAQUE);
    lstrcpy (szText, TEXT ("Opaque background"));
    ExtTextOut (hdc, 60, 175, 0, NULL,
                szText, lstrlen (szText), NULL);

    SetBkMode (hdc, TRANSPARENT);
    lstrcpy (szText, TEXT ("Transparent background"));
    ExtTextOut (hdc, 250, 175, 0, NULL,
                szText, lstrlen (szText), NULL);

    EndPaint (hWnd, &ps);
    return 0;
}
//---------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
```

In Shapes, *OnPaintMain* draws the five figures using the different functions discussed earlier. For each of the shapes, a different brush is created, selected into the device context, and, after the shape has been drawn, deselected from the DC. The first four shapes are filled with solid grayscale shades, ranging from black to white. These solid brushes are loaded with the *GetStockObject* function. The final shape is

filled with a brush created with the *CreateDIBPatternBrushPt*. The creation of this brush is segregated into a function called *MyCreateHatchBrush* that mimics the *CreateHatchBrush* function not available under Windows CE. To create the hatched brushes, a black and white bitmap is built by filling in a bitmap structure and setting the bits to form the hatch patterns. The bitmap itself is the 8-by-8 bitmap specified by *CreateDIBPatternBrushPt*. Since the bitmap is monochrome, its total size, including the palette and header, is only around 100 bytes. Notice, however, that since each scan line of a bitmap must be double-word aligned, the last three bytes of each one-byte scan line are left unused.

Finally the program completes the painting by writing two lines of text into the lower rectangle. The text further demonstrate the difference between the opaque and transparent drawing modes of the system. In this case, the opaque mode of drawing the text might be a better match for the situation because the hatched lines tend to obscure letters drawn in transparent mode. A view of the Shapes window is shown in Figure 2-12.

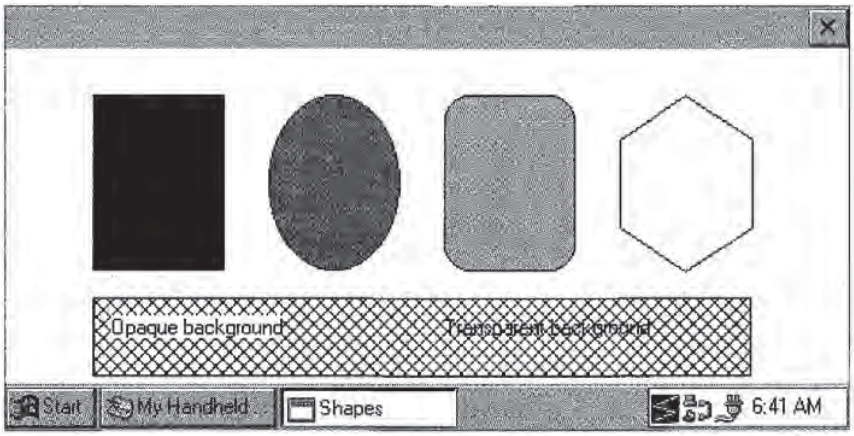


Figure 2-12. *The Shapes example demonstrates drawing different filled shapes.*

To keep things simple, the Shapes example assumes that it's running on at least a 480-pixel-wide display. To properly display the same shapes on a Palm-size PC requires a few minor changes to the coordinates used to position the shapes displayed.

I have barely scratched the surface of the abilities of the Windows CE GDI portion of GWE. The goal of this chapter wasn't to provide total presentation of all aspects of GDI programming. Instead, I wanted to demonstrate the methods available for basic drawing and text support under Windows CE. In other chapters in the book, I extend some of the techniques touched on in this chapter. I talk about these new

techniques and newly introduced functions at the point, generally, where I demonstrate how to use them in code. To further your knowledge, I recommend *Programming Windows 95*, by Charles Petzold (Microsoft Press, 1996), as the best source for learning about the Windows GDI.

Now that we've looked at output, it's time to turn our attention to the input side of the system, the keyboard and touch panel.

Chapter 3

Input: Keyboard, Stylus, and Menus

Traditionally, Microsoft Windows platforms have allowed users two methods of input: the keyboard and the mouse. Windows CE continues this tradition, but replaces the mouse with a stylus and touch screen. Programmatically, the change is minor because the messages from the stylus are mapped to the mouse messages used in other versions of Windows. A more subtle but also more important change from versions of Windows that run on PCs is that a system running Windows CE might have either a tiny keyboard or no keyboard at all. This makes the stylus input that much more important for Windows CE systems.

THE KEYBOARD

While keyboards play a lesser role in Windows CE, they're still the best means of entering large volumes of information. Even on systems without a physical keyboard such as the Palm-size PC, *soft* keyboards—controls that simulate keyboards on a touch screen—will most likely be available to the user. Given this, proper handling of keyboard input is critical to all but the most specialized of Windows CE applications. While I'll talk at length about soft keyboards later in the book, one point should be made here. To the application, input from a soft keyboard is no different from input from a traditional "hard" keyboard.

Input Focus

Under Windows operating systems, only one window at a time has the input focus. The focus window receives all keyboard input until it loses focus to another window. The system assigns the keyboard focus using a number of rules but most often the focus window is the current active window. The active window, you'll recall, is the top-level window, the one with which the user is currently interacting. With rare exceptions, the active window also sits at the top of the Z-order; that is, it's drawn on top of all other windows in the system. The user can change the active window by pressing Alt-Esc to switch between programs or by tapping on another top-level window's button on the task bar. The focus window is either the active window or one of its child windows.

Under Windows, a program can determine which window has the input focus by calling

```
HWND GetFocus (void);
```

The focus can be changed to another window by calling

```
HWND SetFocus (HWND hWnd);
```

Under Windows CE, the target window of *SetFocus* is limited. The window being given the focus by *SetFocus* must have been created by the thread calling *SetFocus*. An exception to this rule occurs if the window losing focus is related to the window gaining focus by a parent/child or sibling relationship; in this case, the focus can be changed even if the windows were created by different threads.

When a window loses focus, Windows sends a WM_KILLFOCUS message to that window informing it of its new state. The *wParam* parameter contains the handle of the window that will be gaining the focus. The window gaining focus receives a WM_SETFOCUS message. The *wParam* parameter of the WM_SETFOCUS message contains the handle of the window losing focus.

Now for a bit of motherhood. Programs shouldn't change the focus window without some input from the user. Otherwise, the user can easily become confused. A proper use of *SetFocus* is to set the input focus to a child window (more than likely a control) contained in the active window. In this case, a window would respond to the WM_SETFOCUS message by calling *SetFocus* with the handle of a child window contained in the window to which the program wants to direct keyboard messages.

Keyboard Messages

Windows CE practices the same keyboard message processing as its larger desktop relations with a few small exceptions, which I cover shortly. When a key is pressed, Windows sends a series of messages to the focus window, typically beginning with a WM_KEYDOWN message. If the key pressed represents a character such as letter or

number, Windows follows the WM_KEYDOWN with a WM_CHAR message. (Some keys, such as function keys and cursor keys don't represent characters, so WM_CHAR messages aren't sent in response to those keys. For those keys, a program must interpret the WM_KEYDOWN message to know when the keys are pressed.) When the key is released, Windows sends a WM_KEYUP message. If a key is held down long enough for the auto-repeat feature to kick in, multiple WM_KEYDOWN and WM_CHAR messages are sent for each auto-repeat until the key is released when the final WM_KEYUP message is sent. I used the word *typically* to qualify this process because if the Alt key is being held when another key is pressed, the messages I've just described are replaced by WM_SYSKEYDOWN, WM_SYSCHAR, and WM_SYSKEYUP messages.

For all of these messages, the generic parameters *wParam* and *lParam* are used in mostly the same manner. For WM_KEY*xx* and WM_SYSKEY*xx* messages, the *wParam* value contains the virtual key value, indicating the key being pressed. All versions of Windows provide a level of indirection between the keyboard hardware and applications by translating the scan codes returned by the keyboard into virtual key values. You see a list of the VK_*xx* values and their associated keys in Figure 3-1. While the table of virtual keys is extensive, not all keys listed in the table are present on Windows CE devices. For example, function keys, a mainstay on PC keyboards and listed in the virtual key table, aren't present on most Windows CE keyboards. In fact, a number of keys on a PC keyboard are left off the space-constrained Windows CE keyboards. A short list of the keys not typically used on Windows CE devices is presented in Figure 3-2 on page 92. This list is meant to inform you that these keys might not exist, not to indicate that the keys *never* exist on Windows CE keyboards.

VIRTUAL-KEY CODES

Constant	*Value*	*Keyboard Equivalent*
VK_LBUTTON	01	Stylus tap
VK_RBUTTON	02	Mouse right button[§]
VK_CANCEL	03	Control-break processing
VK_RBUTTON	04	Mouse middle button[§]
—	05–07	Undefined
VK_BACK	08	Backspace key
VK_TAB	09	Tab key
—	0A–0B	Undefined
VK_CLEAR	0C	Clear key

Figure 3-1. *Virtual key values in relation to the keys on the keyboard. Not all keys will be on all keyboards.* *(continued)*

Figure 3-1. *continued*

Constant	*Value*	*Keyboard Equivalent*
VK_RETURN	0D	Enter key
--	0E–0F	Undefined
VK_SHIFT	10	Shift key
VK_CONTROL	11	Ctrl key
VK_MENU	12	Alt key
VK_CAPITAL	14	Caps Lock key
--	15–19	Reserved for Kanji systems
--	1A	Undefined
VK_ESCAPE	1B	Escape key
--	1C–1F	Reserved for Kanji systems
VK_SPACE	20	Spacebar
VK_PRIOR	21	Page Up key
VK_NEXT	22	Page Down key
VK_END	23	End key
VK_HOME	24	Home key
VK_LEFT	25	Left Arrow key
VK_UP	26	Up Arrow key
VK_RIGHT	27	Right Arrow key
VK_DOWN	28	Down Arrow key
VK_SELECT	29	Select key
--	2A	Original equipment manufacturer (OEM)–specific
VK_EXECUTE	2B	Execute key
VK_SNAPSHOT	2C	Print Screen key for Windows 3.0 and later
VK_INSERT	2D	Insert *
VK_DELETE	2E	Delete†
VK_HELP	2F	Help key
VK_0–VK_9	30–39	0–9 keys
--	3A–40	Undefined
VK_A–VK_Z	41–5A	A through Z keys
VK_LWIN	5B	Windows key
VK_RWIN	5C	Windows key *

Constant	*Value*	*Keyboard Equivalent*
VK_APPS	5D	
--	5E–5F	Undefined
VK_NUMPAD0–9	60–69	Numeric keypad 0–9 keys
VK_MULTIPLY	6A	Numeric keypad Asterisk (*) key
VK_ADD	6B	Numeric keypad Plus sign (+) key
VK_SEPARATOR	6C	Separator key
VK_SUBTRACT	6D	Numeric keypad Minus sign (–) key
VK_DECIMAL	6E	Numeric keypad Period (.) key
VK_DIVIDE	6F	Numeric keypad Slash mark (/) key
VK_F1–VK_F24	70–87	F1–F24 *
--	88–8F	Unassigned
VK_NUMLOCK	90	Num Lock *
VK_SCROLL	91	Scroll Lock *
--	92–9F	Unassigned
VK_LSHIFT	A0	Left Shift‡
VK_RSHIFT	A1	Right Shift‡
VK_LCONTROL	A2	Left Control‡
VK_RCONTROL	A3	Right Control‡
VK_LMENU	A4	Left Alt‡
VK_RMENU	A5	Right Alt‡
--	A6–B9	Unassigned
VK_SEMICOLON	BA	; key
VK_EQUAL	BB	= key
VK_COMMA	BC	, key
VK_HYPHEN	BD	- key
VK_PERIOD	BE	. key
VK_SLASH	BF	/ key
VK_BACKQUOTE	C0	` key
--	C1–DA	Unassigned
VK_LBRACKET	DB	[key
VK_BACKSLASH	DC	\ key
VK_RBRACKET	DD	] key
VK_APOSTROPHE	DE	' key

(continued)

Figure 3-1. *continued*

Constant	*Value*	*Keyboard Equivalent*
VK_OFF	DF	Power button
—	E5	Unassigned
—	E6	OEM-specific
—	E7–E8	Unassigned
—	E9–F5	OEM-specific
VK_ATTN	F6	
VK_CRSEL	F7	
VK_EXSEL	F8	
VK_EREOF	F9	
VK_PLAY	FA	
VK_ZOOM	FB	
VK_NONAME	FC	
VK_PA1	FD	
VK_OEM_CLEAR	FE	

* Many Windows CE Systems don't have this key.

† On some Windows CE systems, Delete is simulated with Shift-Backspace

‡ These constants can be used only with *GetKeyState* and *GetAsyncKeyState*.

§ Mouse right and middle buttons are defined but are relevant only on a Windows CE system equipped with a mouse.

For the WM_CHAR and WM_SYSCHAR messages, the *wParam* value contains the Unicode character represented by the key. Most often an application can simply look for WM_CHAR messages and ignore WM_KEYDOWN and WM_KEYUP. The WM_CHAR message allows for a second level of abstraction so that the application doesn't have to worry about the up or down state of the keys and can concentrate on the characters being entered by means of the keyboard.

The *lParam* value of any of these keyboard messages contains further information about the pressed key. The format of the *lParam* parameter is shown in Figure 3-3 on the following page.

InsertDelete (Many Windows CE keyboards use Shift-Backspace for this function.)
Num LockPause
Print Screen
Scroll Lock
Function Keys
Windows Context Menu key

Figure 3-2. *Keys on a PC keyboard that are rarely on a Windows CE keyboard.*

The low word, bits 0 through 15, contains the repeat count of the key. Often, keys on a Windows CE device can be pressed faster than Windows CE can send messages to the focus application. In these cases, the repeat count contains the number of times the key has been pressed. Bit 29 contains the context flag. If the Alt key was being held down when the key was pressed, this bit will be set. Bit 30 contains the previous key state. If the key was previously down, this bit is set; otherwise it's 0. Bit 30 can be used to determine whether the key message is the result of an autorepeat sequence. Bit 31 indicates the transition state. If the key is in transition from down to up, Bit 31 is set. The Reserved field, bits 16 through 28, is used in the desktop versions of Windows to indicate the key scan code. In almost all cases, Windows CE doesn't support this field. However, on some of the newer Windows CE platforms where scan codes are necessary, this field does contain the scan code. You shouldn't plan on the scan code field being available unless you know it's supported on your specific platform.

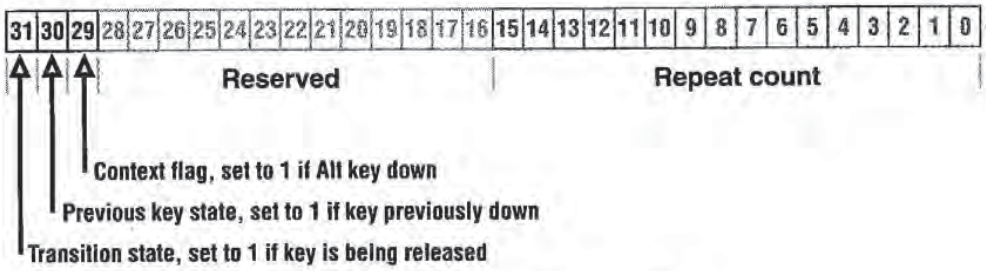


Figure 3-3. *The layout of the* lParam *value for key messages.*

One additional keyboard message, WM_DEADCHAR, can sometimes come into play. You send it when the pressed key represents a dead character, such as an umlaut, that you want to combine with a character to create a different character. In this case the WM_DEADCHAR message can be used to prevent the text entry point (the caret) from advancing to the next space until the second key is pressed so that you can complete the combined character.

The WM_DEADCHAR message has always been present under Windows, but under Windows CE it takes on a somewhat larger role. With the internationalization of small consumer devices that run Windows CE, programmers should plan for, and if necessary use, the WM_DEADCHAR message that is so often necessary in foreign language systems.

Keyboard Functions

You will find useful a few other keyboard-state-determining functions for Windows applications. Among the keyboard functions, two are closely related but often confused: *GetKeyState* and *GetAsyncKeyState*.

GetKeyState, prototyped as

```
SHORT GetKeyState (int nVirtKey);
```

returns the up/down state of the shift keys, Ctrl, Alt, and Shift, and indicates whether any of these keys is in a toggled state. If the keyboard has two keys with the same function—for example, two Shift keys, one on each side of the keyboard—this function can also be used to differentiate which of them is being pressed. (Most keyboards have left and right Shift keys, and some include left and right Ctrl and Alt keys.)

You pass to the function the virtual key code for the key being queried. If the high bit of the return value is set, the key is down. If the least significant bit of the return value is set, the key is in a toggled state; that is, it has been pressed an odd number of times since the system was started. The state returned is the state at the time the most recent message was read from the message queue, which isn't necessarily the real-time state of the key. An interesting aside: notice that the virtual key label for the Alt key is VK_MENU, which relates to the windows convention that the Alt-shift key combination works in concert with other keys to access various menus from the keyboard.

Note that the *GetKeyState* function is limited under Windows CE to querying the state of the shift keys. Under other versions of Windows, *GetKeyState* can determine the state of every key on the keyboard.

To determine the real-time state of a key, use

```
SHORT GetAsyncKeyState (int vKey);
```

As with *GetKeyState*, you pass to this function the virtual key code for the key being queried. The *GetAsyncKeyState* function returns a value subtly different from the one returned by *GetKeyState*. As with the *GetKeyState* function, the high bit of the return value is set while the key is being pressed. However, the least significant bit is then set if the key was pressed after a previous call to *GetAsyncKeyState*. Like *GetKeyState*, the *GetAsyncKeyState* function can distinguish the left and right Shift, Ctrl, and Alt keys. In addition, by passing the VK_LBUTTON virtual key value, *GetAsyncKeyState* determines whether the stylus is currently touching the screen.

An application can simulate a keystroke using the *keybd_event* function:

```
VOID keybd_event (BYTE bVk, BYTE bScan, DWORD dwFlags,
                  DWORD dwExtraInfo);
```

The first parameter is the virtual key code of the key to simulate. The *bScan* code should be set to NULL under Windows CE. The *dwFlags* parameter can have two possible flags: KEYEVENTF_KEYUP indicates that the call is to emulate a key up event while KEYEVENTF_SILENT indicates that the simulated key press won't cause the standard keyboard click that you normally hear when you press a key. So, to fully simulate a key press, *keybd_event* should be called twice, once without

KEYEVENTF_KEYUP to simulate a key down, then once again, this time *with* KEYEVENTF_KEYUP to simulate the key release.

One final keyboard function, *MapVirtualKey*, translates virtual key codes to characters. *MapVirtualKey* in Windows CE doesn't translate keyboard scan codes to and from virtual key codes, although it does so in other versions of Windows. The prototype of the function is the following:

```
UINT MapVirtualKey (UINT uCode, UINT uMapType);
```

Under Windows CE, the first parameter is the virtual key code to be translated while the second parameter, *uMapType*, must be set to 2.

Testing for the keyboard

To determine whether a keyboard is even present in the system, first call *GetVersionEx* to find out which version of Windows CE is running. All systems that run Windows CE 1.0 have a keyboard. When running under Windows CE 2.0 or later, call

```
DWORD GetKeyboardStatus (VOID);
```

This function returns the KBDI_KEYBOARD_PRESENT flag if a hardware keyboard is present in the system. This function also returns a KBDI_KEYBOARD_ENABLED flag if the keyboard is enabled. To disable the keyboard, a call can be made to

```
BOOL EnableHardwareKeyboard (BOOL bEnable);
```

with the *bEnable* flag set to FALSE. You might want to disable the keyboard in a system for which the keyboard folds around behind the screen; in such a system, a user could accidentally hit keys while using the stylus. This function is also new to Windows CE 2.0.

If you build an application to run under Windows CE 1.0, you'll need to explicitly load both *GetKeyboardStatus* and *EnableHardwareKeyboard* using *LoadLibrary* and *GetProcAddress* to determine the address of these 2.0-specific functions. If a call is made directly to a 2.0 function from an application, that application is incompatible with Windows CE 1.0 and won't load.

The KeyTrac Example Program

The following example program, KeyTrac, displays the sequence of keyboard messages. Programmatically, KeyTrac isn't much of a departure from the earlier programs in the book. The difference is that the keyboard messages I've been describing are all trapped and recorded in an array that's then displayed during the WM_PAINT message. For each keyboard message, the message name is recorded along with the *wParam* and *lParam* values and a set of flags indicating the state of the shift keys. The key messages are recorded in an array because these messages can occur faster than the redraw can occur. Figure 3-4 shows the KeyTrac window after a few keys have been pressed.

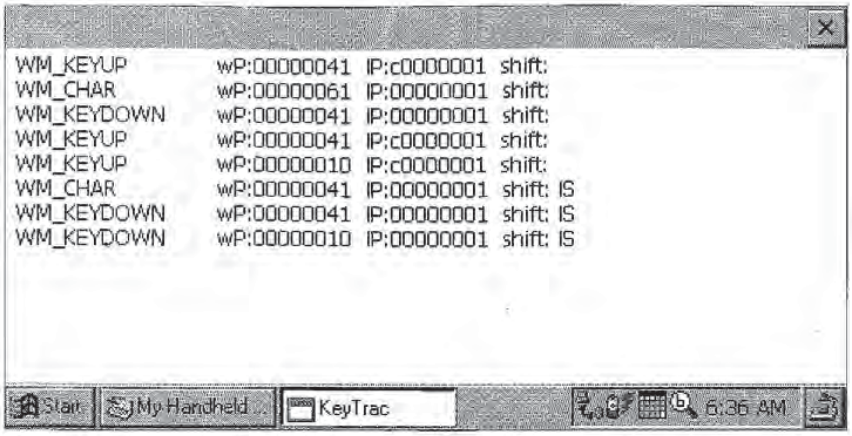


Figure 3-4. *The KeyTrac window after a Shift-A key combination followed by a lowercase* a *key press.*

The best way to learn about the sequence of the keyboard messages is to run KeyTrac, press a few keys, and watch the messages scroll down the screen. Pressing a character key such as the *a* results in three messages: WM_KEYDOWN, WM_CHAR, and WM_KEYUP. Holding down the Shift key while pressing the *a* and then releasing the Shift key produces a key-down message for the Shift key followed by the three messages for the *a* key followed by a key-up message for the Shift key. Because the Shift key itself isn't a character key, no WM_CHAR message is sent in response to it. However, the WM_CHAR message for the *a* key now contains a *0x41* in the *wParam* value, indicating that an uppercase *A* was entered instead of a lowercase *a*.

Figure 3-5 shows the source code for the KeyTrac program.

KeyTrac.h

```
//------------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//------------------------------------------------------------------------
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

//------------------------------------------------------------------------
// Generic defines and data types
//
```

Figure 3-5. *The KeyTrac program.*

```
struct decodeUINT {                                 // Structure associates
    UINT Code;                                      // messages
                                                    // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                                  // Structure associates
    UINT Code;                                      // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);         // function.
};

//------------------------------------------------------------------------
// Generic defines used by application
#define  IDC_CMDBAR 1                               // Command bar ID

//------------------------------------------------------------------------
// Program-specific defines and structures
//
#define FLAG_LMENU      0x0001
#define FLAG_RMENU      0x0002
#define FLAG_LCONTROL   0x0004
#define FLAG_RCONTROL   0x0008
#define FLAG_LSHIFT     0x0010
#define FLAG_RSHIFT     0x0020

typedef struct {
    UINT wKeyMsg;
    INT wParam;
    INT lParam;
    UINT wFlags;
    TCHAR szMsgTxt[64];
} KEYARRAY, *PKEYARRAY;

//------------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoKeysMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);
```

(continued)

Figure 3-5. *continued*

KeyTrac.c

```
//-----------------------------------------------------------------------
// KeyTrac - displays keyboard messages
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//-----------------------------------------------------------------------
#include <windows.h>                    // For all that Windows stuff
#include <commctrl.h>                   // Command bar includes
#include "keytrac.h"                    // Program-specific stuff

//-----------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("KeyTrac");
HINSTANCE hInst;                        // Program instance handle

// Program-specific global data
KEYARRAY ka[16];
UINT wKeyMsg = 0;
INT nKeyCnt = 0, nFontHeight;
TCHAR szMsgTxt[64];

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_PAINT, DoPaintMain,
    WM_KEYUP, DoKeysMain,
    WM_KEYDOWN, DoKeysMain,
    WM_CHAR, DoKeysMain,
    WM_DEADCHAR, DoKeysMain,
    WM_SYSCHAR, DoKeysMain,
    WM_SYSDEADCHAR, DoKeysMain,
    WM_SYSKEYDOWN, DoKeysMain,
    WM_SYSKEYUP, DoKeysMain,
    WM_DESTROY, DoDestroyMain,
};

//-----------------------------------------------------------------------
//
// Program entry point
//
```

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
    HWND hwndMain;

    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//------------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application main window class.
    wc.style = 0;                                   // Window style
    wc.lpfnWndProc = MainWndProc;                   // Callback function
    wc.cbClsExtra = 0;                              // Extra class data
    wc.cbWndExtra = 0;                              // Extra window data
    wc.hInstance = hInstance;                       // Owner handle
    wc.hIcon = NULL;                                // Application icon
    wc.hCursor = NULL;                              // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                        // Menu name
    wc.lpszClassName = szAppName;                   // Window class name

    if (RegisterClass(&wc) == 0) return 1;

    return 0;
}
```

(continued)

Figure 3-5. *continued*

```
//----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow) {
                   HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    hWnd = CreateWindow (szAppName,            // Window class
                         TEXT ("KeyTrac"),     // Window title
                         WS_VISIBLE,           // Style flags
                         CW_USEDEFAULT,        // x position
                         CW_USEDEFAULT,        // y position
                         CW_USEDEFAULT,        // Initial Width
                         CW_USEDEFAULT,        // Initial Height
                         NULL,                 // Parent
                         NULL,                 // Menu, must be null
                         hInstance,            // App instance
                         NULL);                // Pointer to create
                                               // parameters
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//----------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {

    return nDefRC;
}
//======================================================================
// Message handling procedures for MainWindow
//

//----------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
```

```
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//------------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB;
    HDC hdc;
    TEXTMETRIC tm;

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);

    // Get the height of the default font.
    hdc = GetDC (hWnd);
    GetTextMetrics (hdc, &tm);
    nFontHeight = tm.tmHeight + tm.tmExternalLeading;
    ReleaseDC (hWnd, hdc);

    return 0;
}
//------------------------------------------------------------------------
// DoPaintMain - Process WM_PAINT message for window.
//
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect, rectOut;
    TCHAR szOut[256];
    HDC hdc;
    INT i;

    // Adjust the size of the client rect to take into account
    // the command bar height.
```

(continued)

Figure 3-5. *continued*

```
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    // Create a drawing rectangle for the bottom line of the window.
    rectOut = rect;
    rectOut.top = rectOut.bottom - nFontHeight;

    hdc = BeginPaint (hWnd, &ps);

    if (nKeyCnt) {
        for (i = 0; i < nKeyCnt; i++) {
            // Scroll window up by one line.
            ScrollDC (hdc, 0, -nFontHeight, &rect, &rect, NULL, NULL);
            // Write key name, use opaque mode to erase background.
            ExtTextOut (hdc, 5, rect.bottom - nFontHeight, ETO_OPAQUE,
                        &rectOut, ka[i].szMsgTxt,
                        lstrlen (ka[i].szMsgTxt), NULL);
            // Write key variables.
            wsprintf (szOut, TEXT ("wParam:%08x    lParam:%08x    shift: "),
                      ka[i].wParam, ka[i].lParam);

            if (ka[i].wFlags & FLAG_LMENU)
                lstrcat (szOut, TEXT ("lA "));
            if (ka[i].wFlags & FLAG_RMENU)
                lstrcat (szOut, TEXT ("rA "));

            if (ka[i].wFlags & FLAG_LCONTROL)
                lstrcat (szOut, TEXT ("lC "));
            if (ka[i].wFlags & FLAG_RCONTROL)
                lstrcat (szOut, TEXT ("rC "));

            if (ka[i].wFlags & FLAG_LSHIFT)
                lstrcat (szOut, TEXT ("lS "));
            if (ka[i].wFlags & FLAG_RSHIFT)
                lstrcat (szOut, TEXT ("rS "));

            ExtTextOut (hdc, 125, rect.bottom - nFontHeight, 0, NULL,
                        szOut, lstrlen (szOut), NULL);
        }
        nKeyCnt = 0;
    }
    EndPaint (hWnd, &ps);
    return 0;
}
//---------------------------------------------------------------------
```

```
// DoKeysMain - Process all keyboard messages for window.
//
LRESULT DoKeysMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                    LPARAM lParam) {

    if (nKeyCnt >= 16)
        return 0;

    switch (wMsg) {
        case WM_KEYUP:
            lstrcpy (ka[nKeyCnt].szMsgTxt, TEXT ("WM_KEYUP"));
            break;

        case WM_KEYDOWN:
            lstrcpy (ka[nKeyCnt].szMsgTxt, TEXT ("WM_KEYDOWN"));
            break;

        case WM_CHAR:
            lstrcpy (ka[nKeyCnt].szMsgTxt, TEXT ("WM_CHAR"));
            break;

        case WM_DEADCHAR:
            lstrcpy (ka[nKeyCnt].szMsgTxt, TEXT ("WM_DEADCHAR"));
            break;

        case WM_SYSCHAR:
            lstrcpy (ka[nKeyCnt].szMsgTxt, TEXT ("WM_SYSCHAR"));
            break;

        case WM_SYSDEADCHAR:
            lstrcpy (ka[nKeyCnt].szMsgTxt, TEXT ("WM_SYSDEADCHAR"));
            break;

        case WM_SYSKEYDOWN:
            lstrcpy (ka[nKeyCnt].szMsgTxt, TEXT ("WM_SYSKEYDOWN"));
            break;

        case WM_SYSKEYUP:
            lstrcpy (ka[nKeyCnt].szMsgTxt, TEXT ("WM_SYSKEYUP"));
            break;

        default:
            lstrcpy (ka[nKeyCnt].szMsgTxt, TEXT ("unknown"));
            break;
    }
```

(continued)

Figure 3-5. *continued*

```
    ka[nKeyCnt].wKeyMsg = wMsg;
    ka[nKeyCnt].wParam = wParam;
    ka[nKeyCnt].lParam = lParam;

    // Capture the state of the shift flags.
    ka[nKeyCnt].wFlags = 0;
    if (GetKeyState (VK_LMENU))
        ka[nKeyCnt].wFlags |= FLAG_LMENU;
    if (GetKeyState (VK_RMENU))
        ka[nKeyCnt].wFlags |= FLAG_RMENU;

    if (GetKeyState (VK_LCONTROL))
        ka[nKeyCnt].wFlags |= FLAG_LCONTROL;
    if (GetKeyState (VK_RCONTROL))
        ka[nKeyCnt].wFlags |= FLAG_RCONTROL;

    if (GetKeyState (VK_LSHIFT))
        ka[nKeyCnt].wFlags |= FLAG_LSHIFT;
    if (GetKeyState (VK_RSHIFT))
        ka[nKeyCnt].wFlags |= FLAG_RSHIFT;

    nKeyCnt++;
    InvalidateRect (hWnd, NULL, FALSE);
    return 0;
}
//---------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
```

Here are a few more characteristics of KeyTrac to notice. After each keyboard message is recorded, an *InvalidateRect* function is called to force a redraw of the window and therefore also a WM_PAINT message. As I mentioned in Chapter 2, a program should never attempt to send or post a WM_PAINT message to a window because Windows needs to perform some setup before it calls a window with a WM_PAINT message.

Another device context function used in KeyTrac is

```
BOOL ScrollDC (HDC hDC, int dx, int dy, const RECT *lprcScroll,
               const RECT *lprcClip, HRGN hrgnUpdate,
               LPRECT lprcUpdate);
```

which scrolls an area of the device context either horizontally or vertically, but under Windows CE, not both directions at the same time. The three rectangle parameters define the area to be scrolled, the area within the scrolling area to be clipped, and the area to be painted after the scrolling ends. Alternatively, a handle to a region can be passed to *ScrollDC*. That region is defined by *ScrollDC* to encompass the region that needs painting after the scroll.

Finally, if the KeyTrac window is covered up for any reason and then re-exposed, the message information on the display is lost. This is because a device context doesn't store the bit information of the display. The application is responsible for saving any information necessary to completely restore the client area of the screen. Since Keytrac doesn't save this information, it's lost when the window is covered up.

THE STYLUS AND THE TOUCH SCREEN

The stylus/touch screen combination is new to Windows platforms, but fortunately, its integration into Windows CE applications is relatively painless. The best way to deal with the stylus is to treat it as a single-button mouse. The stylus creates the same mouse messages that are provided by the mouse in other versions of Windows and by Windows CE systems that use a mouse. The differences that do appear between a mouse and a stylus are due to the different physical realities of the two input devices.

Unlike a mouse, a stylus doesn't have a cursor to indicate the current position of the mouse. Therefore a stylus can't *hover* over a point on the screen in the way that the mouse cursor does. A cursor hovers when a user moves it over a window without pressing a mouse button. This concept can't be applied to programming for a stylus because the touch screen can't detect the position of the stylus when it isn't in contact with the screen.

Another consequence of the difference between a stylus and a mouse is that without a mouse cursor, an application can't provide feedback to the user by means of changes in appearance of a hovering cursor. Windows CE does support setting the cursor for one classic Windows method of user feedback. The busy hourglass cursor, indicating that the user must wait for the system to complete processing, is supported under Windows CE so that applications can display the busy hourglass in the same manner as applications running under other versions of Windows, using the *SetCursor* function.

Stylus Messages

When the user presses the stylus on the screen, the topmost window under that point receives the input focus if it didn't have it before and then receives a WM_LBUTTONDOWN message. When the user lifts the stylus, the window receives

a WM_LBUTTONUP message. Moving the stylus within the same window while it's down causes WM_MOUSEMOVE messages to be sent to the window. For all of these messages, the *wParam* and *lParam* parameters are loaded with the same values. The *wParam* parameter contains a set of bit flags indicating whether the Ctrl or Shift keys on the keyboard are currently held down. As in other versions of Windows, the Alt key state isn't provided in these messages. To get the state of the Alt key when the message was sent, use the *GetKeyState* function.

The *lParam* parameter contains two 16-bit values that indicate the position on the screen of the tap. The low-order 16 bits contains the *x* (horizontal) location relative to the upper left corner of the client area of the window while the high-order 16 bits contains the *y* (vertical) position.

If the user *double-taps,* that is, taps twice on the screen at the same location and within a predefined time, Windows sends a WM_LBUTTONDBLCLK message to the double-tapped window, but only if that window's class was registered with the CS_DBLCLKS style. The class style is set when the window class is registered with *RegisterClass*.

You can differentiate between a tap and a double-tap by comparing the messages sent to the window. When a double-tap occurs, a window first receives the WM_LBUTTONDOWN and WM_LBUTTONUP messages from the original tap. Then a WM_LBUTTONDBLCLK is sent followed by another WM_LBUTTONUP. The trick is to refrain from acting on a WM_LBUTTONDOWN message in any way that precludes action on a subsequent WM_LBUTTONDBLCLK. This is usually not a problem because taps usually select an object while double-tapping launches the default action for the object.

Inking

A typical application for a handheld device is capturing the user's writing on the screen and storing the result as *ink*. This isn't handwriting recognition—simply ink storage. At first pass, the best way to accomplish this would be to store the stylus points passed in each WM_MOUSEMOVE message. The problem is that sometimes small CE-type devices can't send these messages fast enough to achieve a satisfactory resolution. Under Windows CE 2.0, a new function call has been added to assist programmers in tracking the stylus.

```
BOOL GetMouseMovePoints (PPOINT pptBuf, UINT nBufPoints,
                         UINT *pnPointsRetrieved);
```

GetMouseMovePoints returns a number of stylus points that didn't result in WM_MOUSEMOVE messages. The function is passed an array of points, the size of the array (in points), and a pointer to an integer that will receive the number of points

passed back to the application. Once received, these additional points can be used to fill in the blanks between the last WM_MOUSEMOVE message and the current one.

GetMouseMovePoints does throw one curve at you. It returns points in the resolution of the touch panel, not the screen. This is generally set at four times the screen resolution, so you need to divide the coordinates returned by *GetMouseMovePoints* by four to convert them to screen coordinates. The extra resolution helps programs such as handwriting recognizers.

A short example program, PenTrac, illustrates the difference that *GetMouseMovePoints* can make. Figure 3-6 shows the PenTrac window. Notice the two lines of dots across the window. The top line was drawn using points from WM_MOUSEMOVE only. The second line included points that were queried with *GetMouseMovePoints*. The black dots were queried from WM_MOUSEMOVE while the red (lighter) dots were locations queried with *GetMouseMovePoints*.

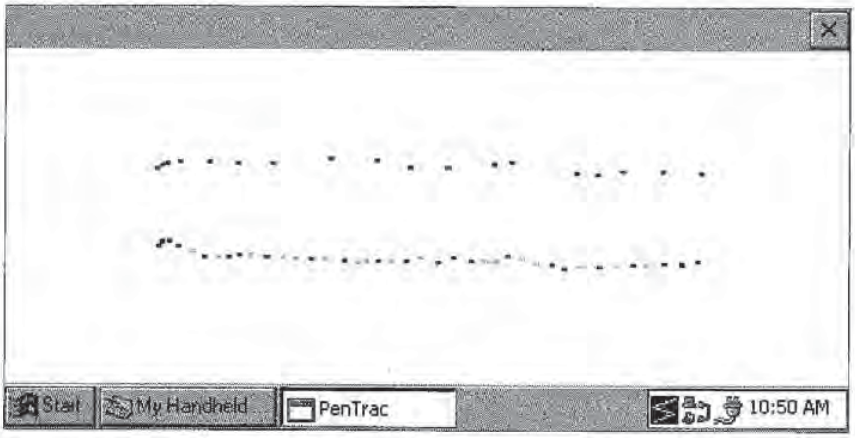


Figure 3-6. *The PenTrac window showing two lines drawn.*

The source code for PenTrac is shown in Figure 3-7. The program places a dot on the screen for each WM_MOUSEMOVE or WM_LBUTTONDOWN message it receives. If the Shift key is held down during the mouse move messages, PenTrac also calls *GetMouseMovePoints* and marks those points in the window in red to distinguish them from the points returned by the mouse messages alone.

PenTrac cheats a little to enhance the effect of *GetMouseMovePoints*. In the *DoMouseMain* routine called to handle WM_MOUSEMOVE and WM_LBUTTONDOWN messages, the routine calls the function *sleep* to kill a few milliseconds. This simulates a slow-responding application that might not have time to process every mouse move message in a timely manner.

PenTrac.h

```
//------------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//------------------------------------------------------------------------
// Returns number of elements.
#define dim(x) (sizeof(x) / sizeof(x[0]))

//------------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                             // Structure associates
    UINT Code;                                  // messages
                                                // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                              // Structure associates
    UINT Code;                                  // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);     // function.
};

//------------------------------------------------------------------------
// Generic defines used by application
#define  IDC_CMDBAR 1                           // Command bar ID

//------------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoMouseMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);
```

Figure 3-7. *The PenTrac program.*

PenTrac.c

```
//--------------------------------------------------------------------
// PenTrac - Tracks stylus movement
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//--------------------------------------------------------------------
#include <windows.h>                    // For all that Windows stuff
#include <commctrl.h>                   // Command bar includes
#include "pentrac.h"                    // Program-specific stuff

//--------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("PenTrac");
HINSTANCE hInst;                        // Program instance handle

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_LBUTTONDOWN, DoMouseMain,
    WM_MOUSEMOVE, DoMouseMain,
    WM_DESTROY, DoDestroyMain,
};

//--------------------------------------------------------------------
//
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
    HWND hwndMain;

    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;
```

(continued)

Figure 3-7. *continued*

```
    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//----------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application main window class.
    wc.style = 0;                           // Window style
    wc.lpfnWndProc = MainWndProc;           // Callback function
    wc.cbClsExtra = 0;                      // Extra class data
    wc.cbWndExtra = 0;                      // Extra window data
    wc.hInstance = hInstance;               // Owner handle
    wc.hIcon = NULL;                        // Application icon
    wc.hCursor = NULL;                      // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                // Menu name
    wc.lpszClassName = szAppName;           // Window class name

    if (RegisterClass (&wc) == 0) return 1;

    return 0;
}
//----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow) {
                   HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    hWnd = CreateWindow (szAppName,         // Window class
                         TEXT ("PenTrac"),  // Window title
                         WS_VISIBLE,        // Style flags
                         CW_USEDEFAULT,     // x position
                         CW_USEDEFAULT,     // y position
```

```
                        CW_USEDEFAULT,      // Initial width
                        CW_USEDEFAULT,      // Initial height
                        NULL,               // Parent
                        NULL,               // Menu, must be null
                        hInstance,          // App instance
                        NULL);              // Pointer to create
                                            // parameters
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//------------------------------------------------------------------------
// TermInstance - Program cleanup
//
int TermInstance (HINSTANCE hInstance, int nDefRC) {

    return nDefRC;
}
//========================================================================
// Message handling procedures for MainWindow
//

//------------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc (hWnd, wMsg, wParam, lParam);
}
//------------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
```

(continued)

Figure 3-7. *continued*

```
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB;

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    return 0;
}
//----------------------------------------------------------------------
// DoMouseMain - Process WM_LBUTTONDOWN and WM_MOUSEMOVE messages
// for window.
//
LRESULT DoMouseMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    POINT pt[64];
    POINT ptM;
    UINT i, uPoints = 0;
    HDC hdc;

    ptM.x = LOWORD (lParam);
    ptM.y = HIWORD (lParam);

    hdc = GetDC (hWnd);
    // If shift and mouse move, see if any lost points.
    if (wMsg == WM_MOUSEMOVE) {
        if (wParam & MK_SHIFT)
            GetMouseMovePoints (pt, 64, &uPoints);

        for (i = 0; i < uPoints; i++) {
            SetPixel (hdc, pt[i].x/4,   pt[i].y/4, RGB (255, 0, 0));
            SetPixel (hdc, pt[i].x/4+1, pt[i].y/4, RGB (255, 0, 0));
            SetPixel (hdc, pt[i].x/4,   pt[i].y/4+1, RGB (255, 0, 0));
            SetPixel (hdc, pt[i].x/4+1, pt[i].y/4+1, RGB (255, 0, 0));
        }
    }
    // The original point is drawn last in case one of the points
    // returned by GetMouseMovePoints overlaps it.
    SetPixel (hdc, ptM.x, ptM.y, RGB (0, 0, 0));
    SetPixel (hdc, ptM.x+1, ptM.y, RGB (0, 0, 0));
    SetPixel (hdc, ptM.x, ptM.y+1, RGB (0, 0, 0));
    SetPixel (hdc, ptM.x+1, ptM.y+1, RGB (0, 0, 0));
    ReleaseDC (hWnd, hdc);
```

```
    // Kill time to make believe we are busy.
    Sleep(25);
    return 0;
}
//---------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
```

Input focus and mouse messages

Here are some subtleties to note about circumstances that rule how and when mouse messages initiated by stylus input are sent to different windows. As I mentioned previously, the input focus of the system changes when the stylus is pressed against a window. However, dragging the stylus from one window to the next won't cause the new window to receive the input focus. The down tap sets the focus, not the process of dragging the stylus across a window. When the stylus is dragged outside the window, that window stops receiving WM_MOUSEMOVE messages but retains input focus. Because the tip of the stylus is still down, no other window will receive the WM_MOUSEMOVE messages. This is akin to using a mouse and dragging the mouse outside a window with a button held down.

To continue to receive mouse messages even if the stylus moves off its window, an application can call

```
HWND SetCapture (HWND hWnd);
```

passing the handle of the window to receive the mouse messages. The function returns the handle of the window that previously had captured the mouse or NULL if the mouse wasn't previously captured. To stop receiving the mouse messages initiated by stylus input, the window calls

```
BOOL ReleaseCapture (void);
```

Only one window can capture the stylus input at any one time. To determine whether the stylus has been captured, an application can call

```
HWND GetCapture (void);
```

which returns the handle of the window that has captured the stylus input or 0 if no window has captured the stylus input—although please note one caveat. *The window*

that has captured the stylus must be in the same thread context as the window calling the function. This means that if the stylus has been captured by a window in another application, *GetCapture* still returns 0.

If a window has captured the stylus input and another window calls *GetCapture*, the window that had originally captured the stylus receives a WM_CAPTURECHANGED message. The *lParam* parameter of the message contains the handle of the window that has gained the capture. You shouldn't attempt to take back the capture by calling *GetCapture* in response to this message. In general, since the stylus is a shared resource, applications should be wary of capturing the stylus for any length of time and they should be able to handle gracefully any loss of capture.

Another interesting tidbit: Just because a window has captured the mouse, that doesn't prevent a tap on another window gaining the input focus for that window. You can use other methods for preventing the change of input focus, but in almost all cases, it's better to let the user, not the applications, decide what top-level window should have the input focus.

Right-button clicks

When you click the right mouse button on an object in Windows systems, the action typically calls up a context menu, which is a stand-alone menu displaying a set of choices for what you can do with that particular object. On a system with a mouse, Windows sends WM_RBUTTONDOWN and WM_RBUTTONUP messages indicating a right-button click. When you use a stylus however, you don't have a right button. The Windows CE guidelines, however, allow you to simulate a right button click using a stylus. The guidelines specify that if a user holds down the Alt key while tapping the screen with the stylus, a program should act as if a right mouse button were being clicked and display any appropriate context menu. Because there's no MK_ALT flag in the *wParam* value of WM_LBUTTONDOWN, the best way to determine whether the Alt key is pressed is to use *GetKeyState* with VK_MENU as the parameter and test for the most significant bit of the return value to be set. *GetKeyState* is more appropriate in this case because the value returned will be the state of the key at the time the mouse message was pulled from the message queue.

The TicTac1 Example Program

To demonstrate stylus programming, I have written a trivial tic-tac-toe game. The TicTac1 window is shown in Figure 3-8. The source code for the program is shown in Figure 3-9. This program doesn't allow you to play the game against the computer, nor does it determine the end of the game—it simply draws the board and keeps track of the Xs and Os. Nevertheless, it demonstrates basic stylus interaction.

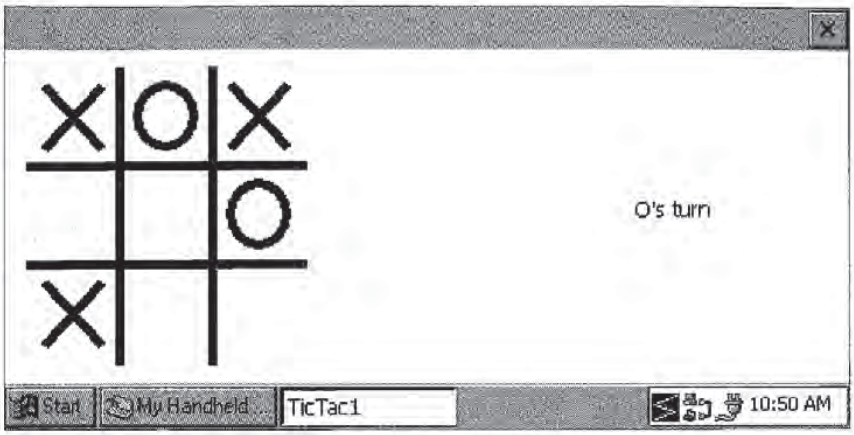


Figure 3-8. *The TicTac1 window.*

TicTac1.h

```
//------------------------------------------------------------------------
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//------------------------------------------------------------------------
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

//------------------------------------------------------------------------
// Generic defines and data types
//
struct decodeUINT {                             // Structure associates
    UINT Code;                                  // messages
                                                // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                              // Structure associates
    UINT Code;                                  // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);     // function.
};

//------------------------------------------------------------------------
// Generic defines used by application
#define  IDC_CMDBAR 1                           // Command bar ID
```

Figure 3-9. *The TicTac1 program.* *(continued)*

Figure 3-9. *continued*

```
//-------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoSizeMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoLButtonDownMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoLButtonUpMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);

// Game function prototypes
void DrawXO (HDC hdc, HPEN hPen, RECT *prect, INT nCell, INT nType);
void DrawBoard (HDC hdc, RECT *prect);
```

TicTac1.c

```
//===================================================================
// TicTac1 - Simple tic-tac-toe game
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//===================================================================
#include <windows.h>                    // For all that Windows stuff
#include <commctrl.h>                   // Command bar includes
#include "tictac1.h"                    // Program-specific stuff

//-------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("TicTac1");
HINSTANCE hInst;                        // Program instance handle

// State data for game
RECT rectBoard = {0, 0, 0, 0};          // Used to place game board.
RECT rectPrompt;                        // Used to place prompt.
BYTE bBoard[9];                         // Keeps track of Xs and Os.
BYTE bTurn = 0;                         // Keeps track of the turn.
```

```
// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_SIZE, DoSizeMain,
    WM_PAINT, DoPaintMain,
    WM_LBUTTONUP, DoLButtonUpMain,
    WM_DESTROY, DoDestroyMain,
};

//====================================================================
//
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
    HWND hwndMain;

    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//--------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;

    // Register application main window class.
    wc.style = 0;                       // Window style
    wc.lpfnWndProc = MainWndProc;       // Callback function
    wc.cbClsExtra = 0;                  // Extra class data
```

(continued)

Figure 3-9. *continued*

```
    wc.cbWndExtra = 0;                          // Extra window data
    wc.hInstance = hInstance;                   // Owner handle
    wc.hIcon = NULL;                            // Application icon
    wc.hCursor = NULL;                          // Default cursor
    wc.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wc.lpszMenuName =  NULL;                    // Menu name
    wc.lpszClassName = szAppName;               // Window class name

    if (RegisterClass (&wc) == 0) return 1;

    return 0;
}
//----------------------------------------------------------------------
// InitInstance - Instance initialization
//
HWND InitInstance (HINSTANCE hInstance, LPWSTR lpCmdLine, int nCmdShow) {
    HWND hWnd;

    // Save program instance handle in global variable.
    hInst = hInstance;

    // Create main window.
    hWnd = CreateWindow (szAppName,             // Window class
                         TEXT ("TicTac1"),      // Window title
                         WS_VISIBLE,            // Style flags
                         CW_USEDEFAULT,         // x position
                         CW_USEDEFAULT,         // y position
                         CW_USEDEFAULT,         // Initial width
                         CW_USEDEFAULT,         // Initial height
                         NULL,                  // Parent
                         NULL,                  // Menu, must be null
                         hInstance,             // App instance
                         NULL);                 // Pointer to create
                                                // parameters
    // Return fail code if window not created.
    if (!IsWindow (hWnd)) return 0;

    // Standard show and update calls
    ShowWindow (hWnd, nCmdShow);
    UpdateWindow (hWnd);
    return hWnd;
}
//----------------------------------------------------------------------
// TermInstance - Program cleanup
//
```

```
int TermInstance (HINSTANCE hInstance, int nDefRC) {

    return nDefRC;
}
//=========================================================================
// Message handling procedures for MainWindow
//

//-------------------------------------------------------------------------
// MainWndProc - Callback function for application window
//
LRESULT CALLBACK MainWndProc (HWND hWnd, UINT wMsg, WPARAM wParam,
                              LPARAM lParam) {
    INT i;
    //
    // Search message list to see if we need to handle this
    // message.  If in list, call procedure.
    //
    for (i = 0; i < dim(MainMessages); i++) {
        if (wMsg == MainMessages[i].Code)
            return (*MainMessages[i].Fxn)(hWnd, wMsg, wParam, lParam);
    }
    return DefWindowProc(hWnd, wMsg, wParam, lParam);
}
//-------------------------------------------------------------------------
// DoCreateMain - Process WM_CREATE message for window.
//
LRESULT DoCreateMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                      LPARAM lParam) {
    HWND hwndCB;

    // Create a command bar.
    hwndCB = CommandBar_Create (hInst, hWnd, IDC_CMDBAR);

    // Add exit button to command bar.
    CommandBar_AddAdornments (hwndCB, 0, 0);
    return 0;
}
//-------------------------------------------------------------------------
// DoSizeMain - Process WM_SIZE message for window.
//
LRESULT DoSizeMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                    LPARAM lParam) {
    RECT rect;
    INT i;
```

(continued)

Figure 3-9. *continued*

```
    // Adjust the size of the client rect to take into account
    // the command bar height.
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    // Init the board rectangle if not yet initialized.
    if (rectBoard.right == 0) {

        // Init the board.
        for (i = 0; i < dim(bBoard); i++)
            bBoard[i] = 0;
    }
    // Define the playing board rect.
    rectBoard = rect;
    rectPrompt = rect;
    // Layout depends on portrait or landscape screen.
    if (rect.right - rect.left > rect.bottom - rect.top) {
        rectBoard.left += 20;
        rectBoard.top += 10;
        rectBoard.bottom -= 10;
        rectBoard.right = rectBoard.bottom - rectBoard.top + 10;

        rectPrompt.left = rectBoard.right + 10;

    } else {
        rectBoard.left += 20;
        rectBoard.right -= 20;
        rectBoard.top += 10;
        rectBoard.bottom = rectBoard.right - rectBoard.left + 10;

        rectPrompt.top = rectBoard.bottom + 10;
    }
    return 0;
}
//-----------------------------------------------------------------------
// DoPaintMain - Process WM_PAINT message for window.
//
LRESULT DoPaintMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                     LPARAM lParam) {
    PAINTSTRUCT ps;
    RECT rect;
    HFONT hFont, hOldFont;
    HDC hdc;

    // Adjust the size of the client rect to take into account
    // the command bar height.
```

```
    GetClientRect (hWnd, &rect);
    rect.top += CommandBar_Height (GetDlgItem (hWnd, IDC_CMDBAR));

    hdc = BeginPaint (hWnd, &ps);

    // Draw the board.
    DrawBoard (hdc, &rectBoard);

    // Write the prompt to the screen.
    hFont = GetStockObject (SYSTEM_FONT);
    hOldFont = SelectObject (hdc, hFont);
    if (bTurn == 0)
        DrawText (hdc, TEXT (" X's turn"), -1, &rectPrompt,
                  DT_CENTER | DT_VCENTER | DT_SINGLELINE);
    else
        DrawText (hdc, TEXT (" O's turn"), -1, &rectPrompt,
                  DT_CENTER | DT_VCENTER | DT_SINGLELINE);

    SelectObject (hdc, hOldFont);
    EndPaint (hWnd, &ps);
    return 0;
}
//------------------------------------------------------------------------
// DoLButtonUpMain - Process WM_LBUTTONUP message for window.
//
LRESULT DoLButtonUpMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                         LPARAM lParam) {
    POINT pt;
    INT cx, cy, nCell = 0;

    pt.x = LOWORD (lParam);
    pt.y = HIWORD (lParam);

    // See if pen on board.  If so, determine which cell.
    if (PtInRect (&rectBoard, pt)){
        // Normalize point to upper left corner of board.
        pt.x -= rectBoard.left;
        pt.y -= rectBoard.top;

        // Compute size of each cell.
        cx = (rectBoard.right - rectBoard.left)/3;
        cy = (rectBoard.bottom - rectBoard.top)/3;

        // Find column.
        nCell = (pt.x / cx);
```

(continued)

Figure 3-9. *continued*

```
        // Find row.
        nCell += (pt.y / cy) * 3;

        // If cell empty, fill it with mark.
        if (bBoard[nCell] == 0) {
            if (bTurn) {
                bBoard[nCell] = 2;
                bTurn = 0;
            } else {
                bBoard[nCell] = 1;
                bTurn = 1;
            }
            InvalidateRect (hWnd, NULL, FALSE);
        } else {
            // Inform the user of the filled cell.
            MessageBeep (0);
            return 0;
        }
    }
    return 0;
}
//----------------------------------------------------------------------
// DoDestroyMain - Process WM_DESTROY message for window.
//
LRESULT DoDestroyMain (HWND hWnd, UINT wMsg, WPARAM wParam,
                       LPARAM lParam) {
    PostQuitMessage (0);
    return 0;
}
//======================================================================
// Game-specific routines
//
//----------------------------------------------------------------------
// DrawXO - Draw a single X or O in a square.
//
void DrawXO (HDC hdc, HPEN hPen, RECT *prect, INT nCell, INT nType) {
    POINT pt[2];
    INT cx, cy;
    RECT rect;

    cx = (prect->right - prect->left)/3;
    cy = (prect->bottom - prect->top)/3;

    // Compute the dimensions of the target cell.
    rect.left = (cx * (nCell % 3) + prect->left) + 10;
    rect.right = rect.right =  rect.left + cx - 20;
    rect.top = cy * (nCell / 3) + prect->top + 10;
```

```
    rect.bottom = rect.top + cy - 20;

    // Draw an X ?
    if (nType == 1) {
        pt[0].x = rect.left;
        pt[0].y = rect.top;
        pt[1].x = rect.right;
        pt[1].y = rect.bottom;
        Polyline (hdc, pt, 2);

        pt[0].x = rect.right;
        pt[1].x = rect.left;
        Polyline (hdc, pt, 2);
    // How about an O ?
    } else if (nType == 2) {
        Ellipse (hdc, rect.left, rect.top, rect.right, rect.bottom);
    }
    return;
}
//---------------------------------------------------------------------
// DrawBoard - Draw the tic-tac-toe board.
//  VK_MENU
void DrawBoard (HDC hdc, RECT *prect) {
    HPEN hPen, hOldPen;
    POINT pt[2];
    LOGPEN lp;
    INT i, cx, cy;

    // Create a nice thick pen.
    lp.lopnStyle = PS_SOLID;
    lp.lopnWidth.x = 5;
    lp.lopnWidth.y = 5;
    lp.lopnColor = RGB (0, 0, 0);
    hPen = CreatePenIndirect (&lp);

    hOldPen = SelectObject (hdc, hPen);

    cx = (prect->right - prect->left)/3;
    cy = (prect->bottom - prect->top)/3;

    // Draw lines down.
    pt[0].x = cx + prect->left;
    pt[1].x = cx + prect->left;
    pt[0].y = prect->top;
    pt[1].y = prect->bottom;
    Polyline (hdc, pt, 2);
```

(continued)

Figure 3-9. *continued*

```
    pt[0].x += cx;
    pt[1].x += cx;
    Polyline (hdc, pt, 2);

    // Draw lines across.
    pt[0].x = prect->left;
    pt[1].x = prect->right;
    pt[0].y = cy + prect->top;
    pt[1].y = cy + prect->top;
    Polyline (hdc, pt, 2);

    pt[0].y += cy;
    pt[1].y += cy;
    Polyline (hdc, pt, 2);

    // Fill in Xs and Os.
    for (i = 0; i < dim (bBoard); i++)
        DrawXO (hdc, hPen, &rectBoard, i, bBoard[i]);

    SelectObject (hdc, hOldPen);
    DeleteObject (hPen);
    return;
}
```

The action in TicTac1 is centered around three routines: *DrawBoard*, *DrawXO*, and *OnLButtonUpMain*. The first two perform the tasks of drawing the playing board. The routine that determines the location of a tap on the board (and therefore is more relevant to our current train of thought) is *OnLButtonUpMain*. As the name suggests, this routine is called in response to a WM_LBUTTONUP message. The first action to take is to call

```
BOOL PtInRect (const RECT *lprc, POINT pt);
```

which determines whether the tap is even on the game board. The program knows the location of the tap because it's passed in the *lParam* value of the message. The board rectangle is computed when the program starts in *OnSizeMain*. Once the tap is localized to the board, the program determines the location of the relevant cell within the playing board by dividing the coordinates of the tap point within the board by the number of cells across and down.

I mentioned that the board rectangle was computed during the *OnSizeMain* routine, which is called in response to a WM_SIZE message. While it might seem strange that Windows CE supports the WM_SIZE message common to other versions of Windows, it needs to support this message because a window is sized frequently: first right after it's created, and then each time it's minimized and restored. You might

think that another possibility for determining the size of the window would be during the WM_CREATE message. The *lParam* parameter points to a CREATESTRUCT structure that contains, among other things, the initial size and position of the window. The problem with using those numbers is that the size obtained is the total size of the window, not the size of client area, which is what we need. Under Windows CE, most windows have no title bar and no border, but some have both and many have scroll bars, so using these values can cause trouble. So now, with the TicTac1 example, we have a simple program that uses the stylus effectively but isn't complete. To restart the game, we must exit and restart TicTac1. We can't take back a move nor have O start first. We need a method for sending these commands to the program. Sure, using keys would work. Another solution would be to create hot spots on the screen that when tapped, provided the input necessary. However, the standard method of exercising these types of commands in a program is through menus.

MENUS

Menus are a mainstay of Windows input. While each application might have a different keyboard and stylus interface, almost all have sets of menus that are organized in a structure familiar to the Windows user.

Windows CE programs use menus a little differently from other Windows programs, the most obvious difference being that in Windows CE, menus aren't part of the standard window. Instead, menus are attached to the command bar control that has been created for the window. Other than this change, the functions of the menu and the way menu selections are processed by the application match the other versions of Windows, for the most part. Because of this general similarity, I give you only a basic introduction to Windows menu management in this section.

Creating a menu is as simple as calling

```
HMENU CreateMenu (void);
```

The function returns a handle to an empty menu. To add an item to a menu, two calls can be used. The first,

```
BOOL AppendMenu (HMENU hMenu, UINT fuFlags, UINT idNewItem,
                 LPCTSTR lpszNewItem);
```

appends a single item to the end of a menu. The *fuFlags* parameter is set with a series of flags indicating the initial condition of the item. For example, the item might be initially disabled (thanks to the MF_GRAYED flag) or have a check mark next to it (courtesy of the MF_CHECKED flag). Almost all calls specify the MF_STRING flag, indicating that the *lpszNewItem* parameter contains a string that will be the text for the item. The *idNewItem* parameter contains an ID value that will be used to identify the item when it's selected by the user or that the state of the menu item needs to be changed.

Another call that can be used to add a menu item is this one:

```
BOOL InsertMenu (HMENU hMenu, UINT uPosition, UINT uFlags,
                 UINT uIDNewItem, LPCTSTR lpNewItem);
```

This call is similar to *AppendMenu* with the added flexibility that the item can be inserted anywhere within a menu structure. For this call, the *uFlags* parameter can be passed one of two additional flags: MF_BYCOMMAND or MF_BYPOSITION, which specify how to locate where the menu item is to be inserted into the menu.

Under Windows CE 2.0, menus can be nested to provide a cascading effect. This feature brings Windows CE up to the level of other versions of Windows, which have always allowed cascading menus. To add a cascading menu, or submenu, create the menu you want to attach using *CreateMenu* and *InsertMenu*. Then insert or append the submenu to the main menu using either *InsertMenu* or *AppendMenu* with the MF_POPUP flag in the flags parameter. In this case, the *uIDNewItem* parameter contains the handle to the submenu while the *lpNewItem* contains the string that will be on the menu item.

You can query and manipulate a menu item to add or remove check marks or to enable or disable it by means of a number of functions. This function,

```
BOOL EnableMenuItem (HMENU hMenu, UINT uIDEnableItem, UINT uEnable);
```

can be used to enable or disable an item. The flags used in the *uEnable* parameter are similar to the flags used with other menu functions. Under Windows CE, the flag you use to disable a menu item is MF_GRAYED, not MF_DISABLED. The function

```
DWORD CheckMenuItem (HMENU hmenu, UINT uIDCheckItem, UINT uCheck);
```

can be used to check and uncheck a menu item. Many other functions are available to query and manipulate menu items. Check the SDK documentation for more details.

The following code fragment creates a simple menu structure:

```
hMainMenu = CreatePopupMenu ();

hMenu = CreateMenu ();
AppendMenu (hMenu, MF_STRING | MF_ENABLED, 100, TEXT ("&New"));
AppendMenu (hMenu, MF_STRING | MF_ENABLED, 101, TEXT ("&Open"));
AppendMenu (hMenu, MF_STRING | MF_ENABLED, 101, TEXT ("&Save"));
AppendMenu (hMenu, MF_STRING | MF_ENABLED, 101, TEXT ("E&xit"));

AppendMenu (hMainMenu, MF_STRING | MF_ENABLED | MF_POPUP, (UINT)hMenu,
            TEXT ("&File"));

hMenu = CreateMenu ();
AppendMenu (hMenu, MF_STRING | MF_ENABLED, 100, TEXT ("C&ut"));
AppendMenu (hMenu, MF_STRING | MF_ENABLED, 101, TEXT ("&Copy"));
AppendMenu (hMenu, MF_STRING | MF_ENABLED, 101, TEXT ("&Paste"));
```

```
AppendMenu (hMainMenu, MF_STRING | MF_ENABLED | MF_POPUP, hMenu,
            TEXT ("&Edit"));

hMenu = CreateMenu ();
AppendMenu (hMenu, MF_STRING | MF_ENABLED, 100, TEXT ("&About"));

AppendMenu (hMainMenu, MF_STRING | MF_ENABLED | MF_POPUP, hMenu,
            TEXT ("&Help"));
```

Once a menu has been created, it can be attached to a command bar using this function:

```
BOOL CommandBar_InsertMenubarEx (HWND hwndCB, HINSTANCE hInst,
                                 LPTSTR pszMenu, int iButton);
```

The menu handle is passed in the third parameter while the second parameter, *hInst*, must be 0. The final parameter, *iButton*, indicates the button that will be to the immediate right of the menu. The Windows CE user interface guidelines recommend that the menu be on the far left of the command bar, so this value is almost always 0.

Handling Menu Commands

When a user selects a menu item, Windows sends a WM_COMMAND message to the window that owns the menu. The low word of the *wParam* parameter contains the ID of the menu item that was selected. The high word of *wParam* contains the notification code. For a menu selection, this value is always 0. The *lParam* parameter is 0 for WM_COMMAND messages sent due to a menu selection. Those familiar with Windows 3.*x* programming might notice that the layout of *wParam* and *lParam* match the standard Win32 assignments and are different from Win16 programs. So, to act on a menu selection, a window needs to field the WM_COMMAND message, decode the ID passed, and act according to the menu item that was selected.

Now that I've covered the basics of menu creation, you might wonder where all this menu creation code sits in a Windows program. The answer is, it doesn't. Instead of dynamically creating menus on the fly, most Windows programs simply load a menu template from a *resource*. To learn more about this, let's take a detour from the description of input methods and look at resources.

RESOURCES

Resources are read-only data segments of an application or a DLL that are linked to the file after it has been compiled. The point of a resource is to give a developer a compiler-independent place for storing content data such as dialog boxes, strings, bitmaps, icons, and yes, menus. Since resources aren't compiled into a program, they can be changed without having to recompile the application.

You create a resource by building an ASCII file—called a *resource script*—describing the resources. Your ASCII file has an extension of RC. You compile this file with a resource compiler, which is provided by every maker of Windows development tools, and then you link them into the compiled executable again using the linker. These days, these steps are masked by a heavy layer of visual tools, but the fundamentals remain the same. For example, Visual C++ 5.0 creates and maintains an ASCII resource (RC) file even though few programmers directly look at the resource file text any more.

It's always a struggle for the author of a programming book to decide how to approach tools. Some lay out a very high level of instruction, talking about menu selections and describing dialog boxes for specific programming tools. Others show the reader how to build all the components of a program from the ground up, using ASCII files and command line compilers. Resources can be approached the same way: I could describe how to use the visual tools or how to create the ASCII files that are the basis for the resources. In this book, I stay primarily at the ASCII resource script level since the goal is to teach Windows CE programming, not how to use a particular set of tools. I'll show how to create and use the ASCII RC file for adding menus and the like, but later in the book in places where the resource file isn't relevant, I won't always include the RC file in the listings. The files are, of course, on the CD included with this book.

Resource Scripts

Creating a resource script is as simple as using Notepad to create a text file. The language used is simple, with C-like tendencies. Comment lines are prefixed by a double slash (//) and files can be included using a *#include* statement.

An example menu template would be the following:

```
//
// A menu template
//
ID_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Open...",                    100
        MENUITEM "&Save...",                    101
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                       120
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About",                      200
    END
END
```

The initial ID_MENU is the ID value for the resource. Alternatively, this ID value can be replaced by a string identifying the resource. The ID value method provides more compact code while using a string may provide more readable code when the application loads the resource in the source file. The next word, *MENU*, identifies the type of resource. The menu starts with *POPUP*, indicating that the menu item *File* is actually a pop-up (cascade) menu attached to the main menu. Because it's a menu within a menu, it too has *BEGIN* and *END* keywords surrounding the description of the File menu. The ampersand (&) character tells Windows that the next character should be the key assignment for that menu item. The character following the ampersand is automatically underlined by Windows when the menu item is displayed, and if the user presses the Alt key along with the character, that menu item is selected. Each item in a menu is then specified by the *MENUITEM* keyword followed by the string used on the menu. The ellipsis following the *Open* and *Save* strings is a Windows UI custom indicating to the user that selecting that item displays a dialog box. The numbers following the *Open*, *Save*, *Exit*, and *About* menu items are the menu identifiers. These values identify the menu items in the WM_COMMAND message. It's good programming practice to replace these values with equates that are defined in a common include file so that they match the WM_COMMAND handler code.

Figure 3-10 lists other resource types that you might find in a resource file. The *DISCARDABLE* keyword is optional and tells Windows that the resource can be discarded from memory if it's not in use. The remainder of the menu is couched in *BEGIN* and *END* keywords, although bracket characters { and } are recognized as well.

Resource Type	***Explanation***
MENU	Defines a menu
ACCELERATORS	Defines a keyboard accelerator table
DIALOG	Defines a dialog box template
BITMAP	Includes a bitmap file as a resource
ICON	Includes an icon file as a resource
FONT	Includes a font file as a resource
RCDATA	Defines application-defined binary data block
STRINGTABLE	Defines a list of strings
VERSIONINFO	Includes file version information

Figure 3-10. *The resource types allowed by the resource compiler.*

Icons

Now that we're working with resource files, it's a trivial matter to modify the icon that the Windows CE shell uses to display a program. Simply create an icon with your favorite icon editor and add to the resource file an icon statement such as

```
ID_ICON ICON "tictac2.ico"
```

When Windows displays a program in Windows Explorer, it looks inside the EXE file for the first icon in the resource list and uses it to represent the program.

Having that icon represent an application's window is somewhat more of a chore. Windows CE uses a small 16-by-16-pixel icon on the taskbar to represent windows on the desktop. Under other versions of Windows, the *RegisterClassEx* function could be used to associate a small icon with a window, but Windows CE doesn't support this function. Instead, the icon must be explicitly loaded and assigned to the window. The following code fragment assigns a small icon to a window.

```
hIcon = (HICON) SendMessage (hWnd, WM_GETICON, FALSE, 0);
if (hIcon == 0) {
    hIcon = LoadImage (hInst, MAKEINTRESOURCE (ID_ICON1), IMAGE_ICON,
                       16, 16, 0);
    SendMessage (hWnd, WM_SETICON, FALSE, (LPARAM)hIcon);
}
```

The first *SendMessage* call gets the currently assigned icon for the window. The FALSE value in *wParam* indicates that we're querying the small icon for the window. If this returns 0, indicating that no icon has been assigned, a call to *LoadImage* is made to load the icon from the application resources. The *LoadImage* function can take either a text string or an ID value to identify the resource being loaded. In this case, the MAKEINTRESOURCE macro is used to label an ID value to the function. The icon being loaded must be a 16-by-16 icon because under Windows CE, *LoadImage* won't resize the icon to fit the requested size. Also under Windows CE, *LoadImage* is limited to loading icons and bitmaps from resources. Windows CE provides the function *ShLoadDIBitmap* to load a bitmap from a file.

Unlike other versions of Windows, Windows CE stores window icons on a per class basis. This means if two windows in an application have the same class, they share the same window icon. A subtle caveat here—window classes are specific to a particular instance of an application. So, if you have two different instances of the application FOOBAR, they each have different window classes, so they may have different window icons even though they were registered with the same class information. If the second instance of FOOBAR had two windows of the same class open, those two windows would share the same icon, independent of the window icon in the first instance of FOOBAR.

Accelerators

Another resource that can be loaded is a keyboard accelerator table. This table is used by Windows to enable developers to designate shortcut keys for specific menus or controls in your application. Specifically, accelerators provide a direct method for a key combination to result in a WM_COMMAND message being sent to a window. These accelerators are different from the Alt-F key combination that, for example, can be used to access a File menu. File menu key combinations are handled automatically as long as the File menu item string was defined with the & character, as in *&File*. The keyboard accelerators are independent of menus or any other controls, although their assignments typically mimic menu operations, as in using Ctrl-O to open a file.

Below is a short resource script that defines a couple of accelerator keys.

```
ID_ACCEL ACCELERATORS DISCARDABLE
BEGIN
    "N", IDM_NEWGAME, VIRTKEY, CONTROL
    "Z", IDM_UNDO,  VIRTKEY, CONTROL
END
```

As with the menu resource, the structure starts with an ID value. The ID value is followed by the type of resource and, again optionally, the discardable keyword. The entries in the table consist of the letter identifying the key, followed by the ID value of the command, *VIRTKEY*, which indicates that the letter is actually a virtual key value, followed finally by the *CONTROL* keyword, indicating that the control shift must be pressed with the key.

Simply having the accelerator table in the resource doesn't accomplish much. The application must load the accelerator table and, for each message it pulls from the message queue, see whether an accelerator has been entered. Fortunately, this is accomplished with a few simple modifications to the main message loop of a program. Here's a modified main message loop that handles keyboard accelerators.

```
// Load accelerator table.
hAccel = LoadAccelerators (hInst, MAKEINTRESOURCE (ID_ACCEL));

// Application message loop
while (GetMessage (&msg, NULL, 0, 0)) {
    // Translate accelerators
    if (!TranslateAccelerator (hwndMain, hAccel, &msg)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
}
```

The first difference in this main message loop is the loading of the accelerator table using the *LoadAccelerators* function. Then after each message is pulled from the message queue, a call is made to *TranslateAccelerator*. If this function translates the message, it returns TRUE, which skips the standard *TranslateMessage* and *DispatchMessage* loop body. If no translation was performed, the loop body executes normally.

Bitmaps

Bitmaps can also be stored as resources. Windows CE works with bitmap resources somewhat differently from other versions of Windows. With Windows CE, the call

```
HBITMAP LoadBitmap(HINSTANCE hInstance, LPCTSTR lpBitmapName);
```

loads a read-only version of the bitmap. This means that after the bitmap is selected into a device context, the image can't be modified by other drawing actions in that DC. To load a read/write version of a bitmap resource, use the *LoadImage* function.

Strings

String resources are a good method for reducing the memory footprint of an application while keeping language-specific information out of the code to be compiled. An application can call

```
int LoadString(HINSTANCE hInstance, UINT uID, LPTSTR lpBuffer,
               int nBufferMax);
```

to load a string from a resource. The ID of the string resource is *uID*, the *lpBuffer* parameter points to a buffer to receive the string, and *nBufferMax* is the size of the buffer. To conserve memory, *LoadString* has a new feature under Windows CE. If *lpBuffer* is NULL, *LoadString* returns a read-only pointer to the string as the return value. Simply cast the return value as a pointer to a constant Unicode string (*LPCTSTR*) and use the string as needed. The length of the string, not including any null terminator, will be located in the word immediately preceding the start of the string.

While I will be covering memory management and strategies for memory conservation in Chapter 6, one quick note here. It's not a good idea to load a number of strings from a resource into memory. This just uses memory both in the resource and in RAM. If you need a number of strings at the same time, it might be a better strategy to use the new feature of *LoadString* to return a pointer directly to the resource itself. As an alternative, you can have the strings in a read-only segment compiled with the program. You lose the advantage of a separate string table, but you reduce your memory footprint.

The TicTac2 Example Program

The final program in this chapter encompasses all of the information presented up to this point as well as a few new items. The TicTac2 program is an extension of TicTac1; the additions are a menu, a window icon, and keyboard accelerators. The TicTac2 window, complete with menu, is shown in Figure 3-11, while the source is shown in Figure 3-12.

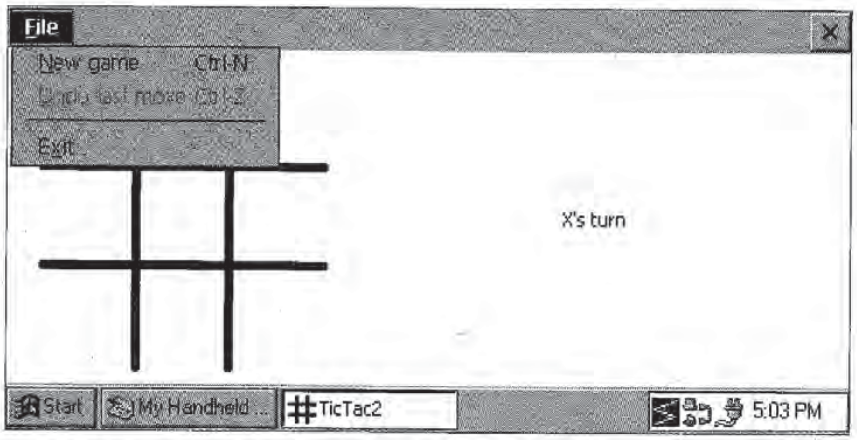


Figure 3-11. *The TicTac2 window wInsertDelete (Many Windows CE keyboards use Shift-Backspace for this function.)*

TicTac2.rc

```
//------------------------------------------------------------------------
// TicTac2 - Resource file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//------------------------------------------------------------------------

#include "tictac2.h"

//------------------------------------------------------------------------
//
// Icon
//
ID_ICON ICON "tictac2.ico"

//------------------------------------------------------------------------
//
```

Figure 3-12. *The Tictac2 program.* *(continued)*

Figure 3-12. *continued*

```
// Menu
//
ID_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New game\tCtrl-N",          IDM_NEWGAME
        MENUITEM "&Undo last move\tCtrl-Z",    IDM_UNDO
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                      IDM_EXIT
    END
END
//----------------------------------------------------------------------
//
// Accelerator table
//
ID_ACCEL ACCELERATORS DISCARDABLE
BEGIN
    "N", IDM_NEWGAME, VIRTKEY, CONTROL
    "Z", IDM_UNDO,  VIRTKEY, CONTROL
END
//----------------------------------------------------------------------
//
// String table
//
STRINGTABLE DISCARDABLE
BEGIN
    IDS_XTURN, " Xs turn"
    IDS_OTURN, " Os turn"
END
```

TicTac2.h

```
//=====================================================================
// Header file
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//=====================================================================
// Returns number of elements
#define dim(x) (sizeof(x) / sizeof(x[0]))

//----------------------------------------------------------------------
// Generic defines and data types
```

```
//
struct decodeUINT {                              // Structure associates
    UINT Code;                                   // messages
                                                 // with a function.
    LRESULT (*Fxn)(HWND, UINT, WPARAM, LPARAM);
};
struct decodeCMD {                               // Structure associates
    UINT Code;                                   // menu IDs with a
    LRESULT (*Fxn)(HWND, WORD, HWND, WORD);      // function.
};

//------------------------------------------------------------------
// Generic defines used by application
#define  IDC_CMDBAR 1                            // Command bar ID

#define  ID_ICON           10                    // Icon resource ID
#define  ID_MENU           11                    // Main menu resource ID
#define  ID_ACCEL          12                    // Main menu resource ID

#define  IDM_NEWGAME       100                   // Menu item ID
#define  IDM_UNDO          101                   // Menu item ID
#define  IDM_EXIT          102                   // Menu item ID

#define  IDS_XTURN         201                   // String ID
#define  IDS_OTURN         202                   // String ID

//------------------------------------------------------------------
// Function prototypes
//
int InitApp (HINSTANCE);
HWND InitInstance (HINSTANCE, LPWSTR, int);
int TermInstance (HINSTANCE, int);

// Window procedures
LRESULT CALLBACK MainWndProc (HWND, UINT, WPARAM, LPARAM);

// Message handlers
LRESULT DoCreateMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoSizeMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoPaintMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoInitMenuPopMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoCommandMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoLButtonUpMain (HWND, UINT, WPARAM, LPARAM);
LRESULT DoDestroyMain (HWND, UINT, WPARAM, LPARAM);
```

(continued)

Figure 3-12. *continued*

```
// Command functions
LPARAM DoMainCommandNewGame (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandUndo (HWND, WORD, HWND, WORD);
LPARAM DoMainCommandExit (HWND, WORD, HWND, WORD);

// Game functon prototypes
void ResetGame (void);
void DrawXO (HDC hdc, HPEN hPen, RECT *prect, INT nCell, INT nType);
void DrawBoard (HDC hdc, RECT *prect);
```

TicTac2.c

```
//--------------------------------------------------------------------
// TicTac2 - Simple tic-tac-toe game with menus
//
// Written for the book Programming Windows CE
// Copyright (C) 1998 Douglas Boling
//
//--------------------------------------------------------------------
#include <windows.h>                    // For all that Windows stuff
#include <commctrl.h>                   // Command bar includes
#include "tictac2.h"                    // Program-specific stuff

//--------------------------------------------------------------------
// Global data
//
const TCHAR szAppName[] = TEXT ("TicTac2");
HINSTANCE hInst;                        // Program instance handle

// State data for game
RECT rectBoard = {0, 0, 0, 0};          // Used to place game board.
RECT rectPrompt;                        // Used to place prompt.
BYTE bBoard[9];                         // Keeps track of Xs and Os.
BYTE bTurn;                             // Keeps track of the turn.
char bLastMove;                         // Last cell changed

// Message dispatch table for MainWindowProc
const struct decodeUINT MainMessages[] = {
    WM_CREATE, DoCreateMain,
    WM_SIZE, DoSizeMain,
    WM_PAINT, DoPaintMain,
    WM_INITMENUPOPUP, DoInitMenuPopMain,
    WM_COMMAND, DoCommandMain,
    WM_LBUTTONUP, DoLButtonUpMain,
    WM_DESTROY, DoDestroyMain,
};
```

```
// Command Message dispatch for MainWindowProc
const struct decodeCMD MainCommandItems[] = {
    IDM_NEWGAME, DoMainCommandNewGame,
    IDM_UNDO, DoMainCommandUndo,
    IDM_EXIT, DoMainCommandExit,
};

//------------------------------------------------------------------------
//
// Program entry point
//
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPWSTR lpCmdLine, int nCmdShow) {
    MSG msg;
    int rc = 0;
    HWND hwndMain;
    HACCEL hAccel;

    // Initialize application.
    rc = InitApp (hInstance);
    if (rc) return rc;

    // Initialize this instance.
    hwndMain = InitInstance (hInstance, lpCmdLine, nCmdShow);
    if (hwndMain == 0)
        return 0x10;

    // Load accelerator table.
    hAccel = LoadAccelerators (hInst, MAKEINTRESOURCE (ID_ACCEL));

    // Application message loop
    while (GetMessage (&msg, NULL, 0, 0)) {
        // Translate accelerators
        if (!TranslateAccelerator (hwndMain, hAccel, &msg)) {
            TranslateMessage (&msg);
            DispatchMessage (&msg);
        }
    }
    // Instance cleanup
    return TermInstance (hInstance, msg.wParam);
}
//------------------------------------------------------------------------
// InitApp - Application initialization
//
int InitApp (HINSTANCE hInstance) {
    WNDCLASS wc;
```

(continued)