

Designed for  
Microsoft®  
Windows NT®  
Windows 98



CD-ROM  
Included



Microsoft® Programming Series

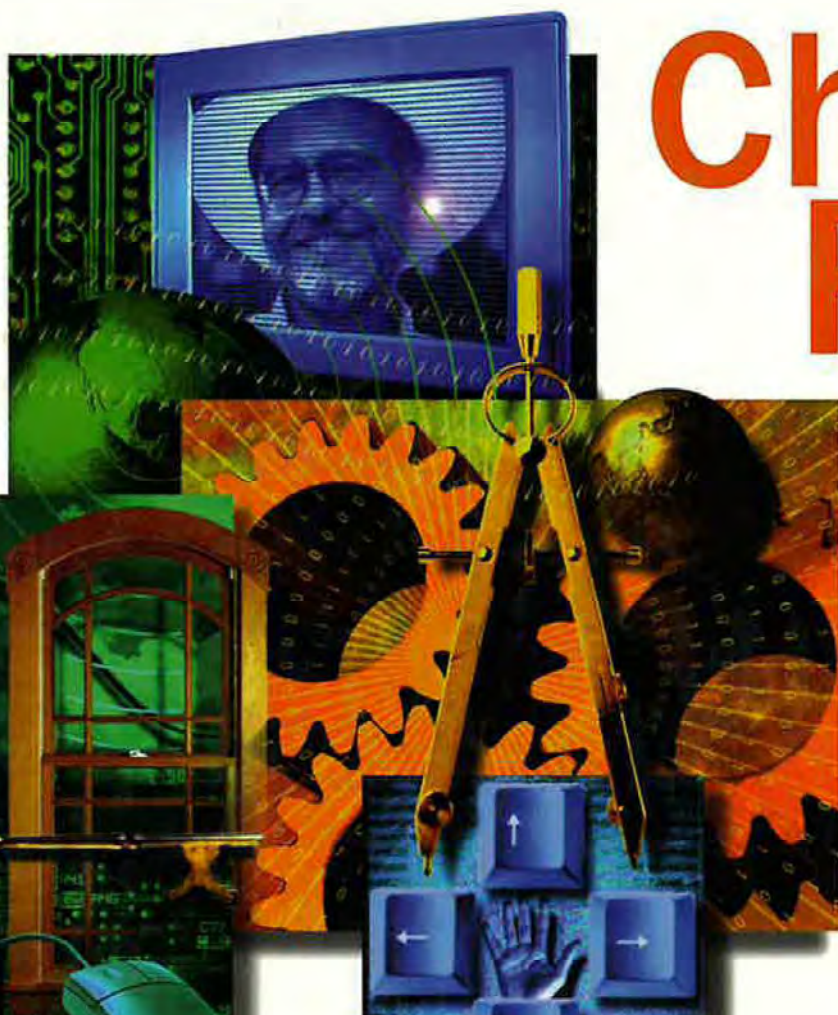
**Completely  
Revised and  
Updated!**

# Programming **Windows**®

## Fifth Edition

# Charles Petzold

**The definitive  
guide to the  
Win32® API**



Petitioner has added letters to pages (a)-(t).  
Otherwise, it leaves the original page numbering.

MICROSOFT EXHIBIT 1097  
*Microsoft v. X1 Discovery*  
IPR2025-00253

**Microsoft Press** (a)

## **Critical acclaim for *Programming Windows***

“The classic book on Windows programming, of course, is Charles Petzold’s PROGRAMMING WINDOWS. If you don’t already have a copy, you need one.”

***Windows Tech Journal***

“The serious programmer’s guide . . . has been and continues to be Charles Petzold’s PROGRAMMING WINDOWS. If you want to program Windows applications and haven’t read this book, you need to do so.”

***IEEE Micro***

“There are just three books that I regard as crucial texts for serious Windows programming: First, of course, is the classic PROGRAMMING WINDOWS by Charles Petzold.”

***INFOWORLD***

“As you are building your library of the best Windows books, you want to make sure you include the classic—Petzold’s best-selling PROGRAMMING WINDOWS. . . .

***PC Techniques***

“This remains the classic Windows programming guide.”

***Programmer’s Journal***

“Just take it as a given—if you’re going to program for Windows, buy this book. It will pay for itself in a matter of hours.”

***Computer Language***

“Broad in scope and omitting little, this book is a must for anyone serious about Windows.”

***Byte Magazine***

# PROGRAMMING WINDOWS®

Fifth Edition

*Charles Petzold*

**Microsoft** Press

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

Copyright © 1999 by Charles Petzold

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Cataloging-in-Publication Data

Petzold, Charles, 1953–

Programming Windows / Charles Petzold. -- 5th ed.

p. cm.

Rev. ed. of: Programming Windows 95.

Includes index.

ISBN 1-57231-995-X

1. Microsoft Windows (Computer file) 2. Operating systems  
(Computers) I. Petzold, Charles, 1953–. Programming Windows 95.

II. Title.

QA76.76.O63P533 1998

005.265--dc21

98-42529

CIP

Printed and bound in the United States of America.

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

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](http://mspress.microsoft.com).

Macintosh, QuickTime, and TrueType fonts are registered trademarks of Apple Computer, Inc. Intel is a registered trademark of Intel Corporation. Developer Studio, DirectX, IntelliMouse, Microsoft, Microsoft Press, Microsoft QuickBasic, MS-DOS, MSDN, Natural, Visual Basic, Visual C++, Visual Studio, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The names of example companies, products, people, characters, and/or data mentioned herein are fictitious and are in no way intended to represent any real individual, company, product, or event, unless otherwise noted.

**Acquisitions Editor:** Ben Ryan  
**Project Editor:** Devon Musgrave  
**Technical Editor:** Dail Magee, Jr.

# Contents at a Glance

## Section I The Basics

|                   |                                   |            |
|-------------------|-----------------------------------|------------|
| <i>Chapter 1</i>  | <b>Getting Started</b>            | <b>3</b>   |
| <i>Chapter 2</i>  | <b>An Introduction to Unicode</b> | <b>19</b>  |
| <i>Chapter 3</i>  | <b>Windows and Messages</b>       | <b>41</b>  |
| <i>Chapter 4</i>  | <b>An Exercise in Text Output</b> | <b>71</b>  |
| <i>Chapter 5</i>  | <b>Basic Drawing</b>              | <b>121</b> |
| <i>Chapter 6</i>  | <b>The Keyboard</b>               | <b>211</b> |
| <i>Chapter 7</i>  | <b>The Mouse</b>                  | <b>273</b> |
| <i>Chapter 8</i>  | <b>The Timer</b>                  | <b>327</b> |
| <i>Chapter 9</i>  | <b>Child Window Controls</b>      | <b>357</b> |
| <i>Chapter 10</i> | <b>Menus and Other Resources</b>  | <b>417</b> |
| <i>Chapter 11</i> | <b>Dialog Boxes</b>               | <b>483</b> |
| <i>Chapter 12</i> | <b>The Clipboard</b>              | <b>567</b> |

## Section II More Graphics

|                   |                                      |             |
|-------------------|--------------------------------------|-------------|
| <i>Chapter 13</i> | <b>Using the Printer</b>             | <b>597</b>  |
| <i>Chapter 14</i> | <b>Bitmaps and Bitblts</b>           | <b>641</b>  |
| <i>Chapter 15</i> | <b>The Device-Independent Bitmap</b> | <b>723</b>  |
| <i>Chapter 16</i> | <b>The Palette Manager</b>           | <b>821</b>  |
| <i>Chapter 17</i> | <b>Text and Fonts</b>                | <b>997</b>  |
| <i>Chapter 18</i> | <b>Metafiles</b>                     | <b>1097</b> |

## Section III Advanced Topics

|                   |                                        |             |
|-------------------|----------------------------------------|-------------|
| <i>Chapter 19</i> | <b>The Multiple-Document Interface</b> | <b>1173</b> |
| <i>Chapter 20</i> | <b>Multitasking and Multithreading</b> | <b>1197</b> |
| <i>Chapter 21</i> | <b>Dynamic-Link Libraries</b>          | <b>1243</b> |
| <i>Chapter 22</i> | <b>Sound and Music</b>                 | <b>1273</b> |
| <i>Chapter 23</i> | <b>A Taste of the Internet</b>         | <b>1405</b> |

# Contents

Author's Note

xxiii

## Section I The Basics

|                  |                                          |           |
|------------------|------------------------------------------|-----------|
| <i>Chapter 1</i> | <b>Getting Started</b>                   | <b>3</b>  |
|                  | <b>THE WINDOWS ENVIRONMENT</b>           | <b>4</b>  |
|                  | A History of Windows                     | 4         |
|                  | Aspects of Windows                       | 6         |
|                  | Dynamic Linking                          | 8         |
|                  | <b>WINDOWS PROGRAMMING OPTIONS</b>       | <b>9</b>  |
|                  | APIs and Memory Models                   | 9         |
|                  | Language Options                         | 10        |
|                  | The Programming Environment              | 11        |
|                  | API Documentation                        | 12        |
|                  | <b>YOUR FIRST WINDOWS PROGRAM</b>        | <b>13</b> |
|                  | A Character-Mode Model                   | 13        |
|                  | The Windows Equivalent                   | 14        |
|                  | The Header Files                         | 15        |
|                  | Program Entry Point                      | 15        |
|                  | The <i>MessageBox</i> Function           | 16        |
|                  | Compile, Link, and Run                   | 18        |
| <i>Chapter 2</i> | <b>An Introduction to Unicode</b>        | <b>19</b> |
|                  | <b>A BRIEF HISTORY OF CHARACTER SETS</b> | <b>20</b> |
|                  | American Standards                       | 20        |
|                  | The World Beyond                         | 21        |
|                  | Extending ASCII                          | 22        |
|                  | Double-Byte Character Sets               | 24        |
|                  | Unicode to the Rescue                    | 25        |
|                  | <b>WIDE CHARACTERS AND C</b>             | <b>26</b> |
|                  | The <i>char</i> Data Type                | 26        |
|                  | Wider Characters                         | 27        |
|                  | Wide-Character Library Functions         | 28        |
|                  | Maintaining a Single Source              | 29        |

## CONTENTS

|                                                        |               |
|--------------------------------------------------------|---------------|
| <b>WIDE CHARACTERS AND WINDOWS</b>                     | <b>31</b>     |
| Windows Header File Types                              | 31            |
| The Windows Function Calls                             | 33            |
| Windows' String Functions                              | 34            |
| Using <i>printf</i> in Windows                         | 34            |
| A Formatting Message Box                               | 37            |
| Internationalization and This Book                     | 38            |
| <br><i>Chapter 3</i> <b>Windows and Messages</b>       | <br><b>41</b> |
| <b>A WINDOW OF ONE'S OWN</b>                           | <b>41</b>     |
| An Architectural Overview                              | 42            |
| The HELLOWIN Program                                   | 44            |
| Thinking Globally                                      | 47            |
| Registering the Window Class                           | 51            |
| Creating the Window                                    | 57            |
| Displaying the Window                                  | 59            |
| The Message Loop                                       | 60            |
| The Window Procedure                                   | 62            |
| Processing the Messages                                | 62            |
| Playing a Sound File                                   | 63            |
| The WM_PAINT Message                                   | 64            |
| The WM_DESTROY Message                                 | 66            |
| <b>THE WINDOWS PROGRAMMING HURDLES</b>                 | <b>66</b>     |
| Don't Call Me, I'll Call You                           | 66            |
| Queued and Nonqueued Messages                          | 68            |
| Get In and Out Fast                                    | 70            |
| <br><i>Chapter 4</i> <b>An Exercise in Text Output</b> | <br><b>71</b> |
| <b>PAINTING AND REPAINTING</b>                         | <b>72</b>     |
| The WM_PAINT Message                                   | 72            |
| Valid and Invalid Rectangles                           | 74            |
| <b>AN INTRODUCTION TO GDI</b>                          | <b>74</b>     |
| The Device Context                                     | 75            |
| Getting a Device Context Handle: Method One            | 75            |
| The Paint Information Structure                        | 77            |
| Getting a Device Context Handle: Method Two            | 79            |
| <i>TextOut</i> : The Details                           | 80            |
| The System Font                                        | 82            |
| The Size of a Character                                | 82            |
| Text Metrics: The Details                              | 83            |

|                                        |            |
|----------------------------------------|------------|
| Formatting Text                        | 85         |
| Putting It All Together                | 86         |
| The SYSMETS1.C Window Procedure        | 94         |
| Not Enough Room                        | 95         |
| The Size of the Client Area            | 95         |
| <b>SCROLL BARS</b>                     | <b>97</b>  |
| Scroll Bar Range and Position          | 99         |
| Scroll Bar Messages                    | 100        |
| Scrolling SYSMETS                      | 102        |
| Structuring Your Program for Painting  | 107        |
| <b>BUILDING A BETTER SCROLL</b>        | <b>108</b> |
| The Scroll Bar Information Functions   | 108        |
| How Low Can You Scroll?                | 110        |
| The New SYSMETS                        | 111        |
| But I Don't Like to Use the Mouse      | 118        |
| <i>Chapter 5</i> <b>Basic Drawing</b>  | <b>121</b> |
| <b>THE STRUCTURE OF GDI</b>            | <b>121</b> |
| The GDI Philosophy                     | 121        |
| The GDI Function Calls                 | 123        |
| The GDI Primitives                     | 124        |
| Other Stuff                            | 125        |
| <b>THE DEVICE CONTEXT</b>              | <b>126</b> |
| Getting a Device Context Handle        | 126        |
| Getting Device Context Information     | 128        |
| The DEVCAPS1 Program                   | 129        |
| The Size of the Device                 | 133        |
| Finding Out About Color                | 138        |
| The Device Context Attributes          | 140        |
| Saving Device Contexts                 | 142        |
| <b>DRAWING DOTS AND LINES</b>          | <b>143</b> |
| Setting Pixels                         | 143        |
| Straight Lines                         | 144        |
| The Bounding Box Functions             | 149        |
| Bezier Splines                         | 156        |
| Using Stock Pens                       | 161        |
| Creating, Selecting, and Deleting Pens | 162        |
| Filling in the Gaps                    | 165        |
| Drawing Modes                          | 166        |

## CONTENTS

|                                                          |            |
|----------------------------------------------------------|------------|
| <b>DRAWING FILLED AREAS</b>                              | <b>168</b> |
| The <i>Polygon</i> Function and the Polygon-Filling Mode | 169        |
| Brushing the Interior                                    | 174        |
| <b>THE GDI MAPPING MODE</b>                              | <b>176</b> |
| Device Coordinates and Logical Coordinates               | 177        |
| The Device Coordinate Systems                            | 178        |
| The Viewport and the Window                              | 179        |
| Working with MM_TEXT                                     | 181        |
| The Metric Mapping Modes                                 | 183        |
| The “Roll Your Own” Mapping Modes                        | 187        |
| The WHATSIZE Program                                     | 192        |
| <b>RECTANGLES, REGIONS, AND CLIPPING</b>                 | <b>196</b> |
| Working with Rectangles                                  | 196        |
| Random Rectangles                                        | 198        |
| Creating and Painting Regions                            | 202        |
| Clipping with Rectangles and Regions                     | 204        |
| The CLOVER Program                                       | 205        |
| <i>Chapter 6</i> <b>The Keyboard</b>                     | <b>211</b> |
| <b>KEYBOARD BASICS</b>                                   | <b>211</b> |
| Ignoring the Keyboard                                    | 212        |
| Who’s Got the Focus?                                     | 212        |
| Queues and Synchronization                               | 213        |
| Keystrokes and Characters                                | 214        |
| <b>KEYSTROKE MESSAGES</b>                                | <b>214</b> |
| System and Nonsystem Keystrokes                          | 215        |
| Virtual Key Codes                                        | 216        |
| The <i>lParam</i> Information                            | 219        |
| Shift States                                             | 221        |
| Using Keystroke Messages                                 | 222        |
| Enhancing SYSMETS for the Keyboard                       | 223        |
| <b>CHARACTER MESSAGES</b>                                | <b>231</b> |
| The Four Character Messages                              | 232        |
| Message Ordering                                         | 233        |
| Control Character Processing                             | 234        |
| Dead-Character Messages                                  | 235        |
| <b>KEYBOARD MESSAGES AND CHARACTER SETS</b>              | <b>236</b> |
| The KEYVIEW1 Program                                     | 236        |
| The Foreign-Language Keyboard Problem                    | 242        |

|                                       |            |
|---------------------------------------|------------|
| Character Sets and Fonts              | 244        |
| What About Unicode?                   | 255        |
| TrueType and Big Fonts                | 256        |
| <b>THE CARET (NOT THE CURSOR)</b>     | <b>263</b> |
| The Caret Functions                   | 264        |
| The TYPER Program                     | 265        |
| <i>Chapter 7</i> <b>The Mouse</b>     | <b>273</b> |
| <b>MOUSE BASICS</b>                   | <b>274</b> |
| Some Quick Definitions                | 274        |
| The Plural of Mouse Is...             | 275        |
| <b>CLIENT-AREA MOUSE MESSAGES</b>     | <b>276</b> |
| Simple Mouse Processing: An Example   | 277        |
| Processing Shift Keys                 | 281        |
| Mouse Double-Clicks                   | 283        |
| <b>NONCLIENT-AREA MOUSE MESSAGES</b>  | <b>284</b> |
| The Hit-Test Message                  | 285        |
| Messages Beget Messages               | 286        |
| <b>HIT-TESTING IN YOUR PROGRAMS</b>   | <b>287</b> |
| A Hypothetical Example                | 287        |
| A Sample Program                      | 288        |
| Emulating the Mouse with the Keyboard | 291        |
| Add a Keyboard Interface to CHECKER   | 293        |
| Using Child Windows for Hit-Testing   | 297        |
| Child Windows in CHECKER              | 297        |
| Child Windows and the Keyboard        | 303        |
| <b>CAPTURING THE MOUSE</b>            | <b>309</b> |
| Blocking Out a Rectangle              | 309        |
| The Capture Solution                  | 313        |
| The BLOKOUT2 Program                  | 314        |
| <b>THE MOUSE WHEEL</b>                | <b>318</b> |
| Still to Come                         | 326        |
| <i>Chapter 8</i> <b>The Timer</b>     | <b>327</b> |
| <b>TIMER BASICS</b>                   | <b>328</b> |
| The System and the Timer              | 328        |
| Timer Messages Are Not Asynchronous   | 329        |

|                                               |            |
|-----------------------------------------------|------------|
| <b>USING THE TIMER: THREE METHODS</b>         | <b>330</b> |
| Method One                                    | 330        |
| Method Two                                    | 334        |
| Method Three                                  | 337        |
| <b>USING THE TIMER FOR A CLOCK</b>            | <b>338</b> |
| Building a Digital Clock                      | 338        |
| Getting the Current Time                      | 343        |
| Displaying Digits and Colons                  | 344        |
| Going International                           | 344        |
| Building an Analog Clock                      | 346        |
| <b>USING THE TIMER FOR A STATUS REPORT</b>    | <b>351</b> |
| <i>Chapter 9</i> <b>Child Window Controls</b> | <b>357</b> |
| <b>THE BUTTON CLASS</b>                       | <b>359</b> |
| Creating the Child Windows                    | 363        |
| The Child Talks to Its Parent                 | 365        |
| The Parent Talks to Its Child                 | 366        |
| Push Buttons                                  | 367        |
| Check Boxes                                   | 368        |
| Radio Buttons                                 | 369        |
| Group Boxes                                   | 369        |
| Changing the Button Text                      | 369        |
| Visible and Enabled Buttons                   | 370        |
| Buttons and Input Focus                       | 370        |
| <b>CONTROLS AND COLORS</b>                    | <b>371</b> |
| System Colors                                 | 372        |
| The Button Colors                             | 373        |
| The WM_CTLCOLORBTN Message                    | 374        |
| Owner-Draw Buttons                            | 375        |
| <b>THE STATIC CLASS</b>                       | <b>382</b> |
| <b>THE SCROLL BAR CLASS</b>                   | <b>383</b> |
| The COLORS1 Program                           | 384        |
| The Automatic Keyboard Interface              | 392        |
| Window Subclassing                            | 393        |
| Coloring the Background                       | 393        |
| Coloring the Scroll Bars and Static Text      | 394        |
| <b>THE EDIT CLASS</b>                         | <b>395</b> |
| The Edit Class Styles                         | 398        |
| Edit Control Notification                     | 399        |

|                                                      |            |
|------------------------------------------------------|------------|
| Using the Edit Controls                              | 399        |
| Messages to an Edit Control                          | 400        |
| <b>THE LISTBOX CLASS</b>                             | <b>401</b> |
| List Box Styles                                      | 401        |
| Putting Strings in the List Box                      | 402        |
| Selecting and Extracting Entries                     | 403        |
| Receiving Messages from List Boxes                   | 404        |
| A Simple List Box Application                        | 405        |
| Listing Files                                        | 409        |
| A <i>head</i> for Windows                            | 411        |
| <i>Chapter 10</i> <b>Menus and Other Resources</b>   | <b>417</b> |
| <b>ICONS, CURSORS, STRINGS, AND CUSTOM RESOURCES</b> | <b>418</b> |
| Adding an Icon to a Program                          | 418        |
| Getting a Handle on Icons                            | 424        |
| Using Icons in Your Program                          | 426        |
| Using Customized Cursors                             | 427        |
| Character String Resources                           | 428        |
| Custom Resources                                     | 430        |
| <b>MENUS</b>                                         | <b>439</b> |
| Menu Concepts                                        | 440        |
| Menu Structure                                       | 440        |
| Defining the Menu                                    | 441        |
| Referencing the Menu in Your Program                 | 441        |
| Menus and Messages                                   | 442        |
| A Sample Program                                     | 445        |
| Menu Etiquette                                       | 451        |
| Defining a Menu the Hard Way                         | 452        |
| Floating Popup Menus                                 | 453        |
| Using the System Menu                                | 459        |
| Changing the Menu                                    | 462        |
| Other Menu Commands                                  | 463        |
| An Unorthodox Approach to Menus                      | 464        |
| <b>KEYBOARD ACCELERATORS</b>                         | <b>469</b> |
| Why You Should Use Keyboard Accelerators             | 469        |
| Some Rules on Assigning Accelerators                 | 470        |
| The Accelerator Table                                | 470        |
| Loading the Accelerator Table                        | 471        |
| Translating the Keystrokes                           | 471        |

## CONTENTS

|                                                     |            |
|-----------------------------------------------------|------------|
| Receiving the Accelerator Messages                  | 472        |
| POPPAD with a Menu and Accelerators                 | 473        |
| Enabling Menu Items                                 | 480        |
| Processing the Menu Options                         | 480        |
| <i>Chapter 11</i> <b>Dialog Boxes</b>               | <b>483</b> |
| <b>MODAL DIALOG BOXES</b>                           | <b>484</b> |
| Creating an "About" Dialog Box                      | 484        |
| The Dialog Box and Its Template                     | 488        |
| The Dialog Box Procedure                            | 491        |
| Invoking the Dialog Box                             | 493        |
| Variations on a Theme                               | 494        |
| A More Complex Dialog Box                           | 497        |
| Working with Dialog Box Controls                    | 505        |
| The OK and Cancel Buttons                           | 507        |
| Avoiding Global Variables                           | 509        |
| Tab Stops and Groups                                | 510        |
| Painting on the Dialog Box                          | 512        |
| Using Other Functions with Dialog Boxes             | 513        |
| Defining Your Own Controls                          | 513        |
| <b>MODELESS DIALOG BOXES</b>                        | <b>520</b> |
| Differences Between Modal and Modeless Dialog Boxes | 521        |
| The New COLORS Program                              | 523        |
| HEXCALC: Window or Dialog Box?                      | 529        |
| <b>THE COMMON DIALOG BOXES</b>                      | <b>538</b> |
| POPPAD Revisited                                    | 538        |
| Unicode File I/O                                    | 563        |
| Changing the Font                                   | 564        |
| Search and Replace                                  | 564        |
| The One-Function-Call Windows Program               | 565        |
| <i>Chapter 12</i> <b>The Clipboard</b>              | <b>567</b> |
| <b>SIMPLE USE OF THE CLIPBOARD</b>                  | <b>568</b> |
| The Standard Clipboard Data Formats                 | 568        |
| Memory Allocation                                   | 570        |
| Transferring Text to the Clipboard                  | 572        |
| Getting Text from the Clipboard                     | 573        |

|                                                  |                          |
|--------------------------------------------------|--------------------------|
| Opening and Closing the Clipboard                | 574                      |
| The Clipboard and Unicode                        | 575                      |
| <b>BEYOND SIMPLE CLIPBOARD USE</b>               | <b>581</b>               |
| Using Multiple Data Items                        | 581                      |
| Delayed Rendering                                | 583                      |
| Private Data Formats                             | 584                      |
| <b>BECOMING A CLIPBOARD VIEWER</b>               | <b>587</b>               |
| The Clipboard Viewer Chain                       | 587                      |
| Clipboard Viewer Functions and Messages          | 587                      |
| A Simple Clipboard Viewer                        | 590                      |
| <br><b>Section II</b>                            | <br><b>More Graphics</b> |
| <i>Chapter 13</i> <b>Using the Printer</b>       | <b>597</b>               |
| <b>PRINTING FUNDAMENTALS</b>                     | <b>598</b>               |
| Printing and Spooling                            | 598                      |
| The Printer Device Context                       | 602                      |
| The Revised DEVCAPS Program                      | 605                      |
| The <i>PrinterProperties</i> Call                | 615                      |
| Checking for <i>BitBlt</i> Capability            | 616                      |
| The Simplest Printing Program                    | 617                      |
| <b>PRINTING GRAPHICS AND TEXT</b>                | <b>619</b>               |
| Bare-Bones Printing                              | 622                      |
| Canceling Printing with an Abort Procedure       | 623                      |
| How Windows Uses <i>AbortProc</i>                | 625                      |
| Implementing an Abort Procedure                  | 625                      |
| Adding a Printing Dialog Box                     | 628                      |
| Adding Printing to POPPAD                        | 633                      |
| <br><i>Chapter 14</i> <b>Bitmaps and Bitblts</b> | <br><b>641</b>           |
| <b>BITMAP BASICS</b>                             | <b>641</b>               |
| Where Do Bitmaps Come From?                      | 642                      |
| <b>BITMAP DIMENSIONS</b>                         | <b>643</b>               |
| Color and Bitmaps                                | 644                      |
| Real-World Devices                               | 644                      |
| Bitmap Support in GDI                            | 647                      |

## CONTENTS

|                                                        |            |
|--------------------------------------------------------|------------|
| <b>THE BIT-BLOCK TRANSFER</b>                          | <b>648</b> |
| A Simple BitBlt                                        | 648        |
| Stretching the Bitmap                                  | 652        |
| The StretchBlt Mode                                    | 656        |
| The Raster Operations                                  | 657        |
| The Pattern Blt                                        | 659        |
| <b>THE GDI BITMAP OBJECT</b>                           | <b>662</b> |
| Creating a DDB                                         | 662        |
| The Bitmap Bits                                        | 665        |
| The Memory Device Context                              | 666        |
| Loading Bitmap Resources                               | 667        |
| The Monochrome Bitmap Format                           | 671        |
| Brushes from Bitmaps                                   | 674        |
| Drawing on Bitmaps                                     | 677        |
| The Shadow Bitmap                                      | 682        |
| Using Bitmaps in Menus                                 | 687        |
| Nonrectangular Bitmap Images                           | 701        |
| Some Simple Animation                                  | 707        |
| Bitmaps Outside the Window                             | 711        |
| <i>Chapter 15</i> <b>The Device-Independent Bitmap</b> | <b>723</b> |
| <b>THE DIB FILE FORMAT</b>                             | <b>724</b> |
| The OS/2-Style DIB                                     | 725        |
| Bottoms Up!                                            | 728        |
| The DIB Pixel Bits                                     | 728        |
| The Expanded Windows DIB                               | 730        |
| Reality Check                                          | 733        |
| DIB Compression                                        | 734        |
| Color Masking                                          | 737        |
| The Version 4 Header                                   | 740        |
| The Version 5 Header                                   | 744        |
| Displaying DIB Information                             | 746        |
| <b>DISPLAYING AND PRINTING</b>                         | <b>755</b> |
| Digging into the DIB                                   | 755        |
| Pixel to Pixel                                         | 758        |
| The Topsy-Turvy World of DIBs                          | 769        |
| Sequential Display                                     | 777        |
| Stretching to Fit                                      | 786        |
| Color Conversion, Palettes, and Performance            | 797        |

|                                              |            |
|----------------------------------------------|------------|
| <b>THE UNION OF DIBs AND DDBs</b>            | <b>799</b> |
| Creating a DDB from a DIB                    | 799        |
| From DDB to DIB                              | 808        |
| The DIB Section                              | 808        |
| More DIB Section Differences                 | 817        |
| The File-Mapping Option                      | 818        |
| In Summary                                   | 820        |
| <i>Chapter 16</i> <b>The Palette Manager</b> | <b>821</b> |
| <b>USING PALETTES</b>                        | <b>821</b> |
| Video Hardware                               | 822        |
| Displaying Gray Shades                       | 823        |
| The Palette Messages                         | 831        |
| The Palette Index Approach                   | 832        |
| Querying the Palette Support                 | 836        |
| The System Palette                           | 837        |
| Other Palette Functions                      | 837        |
| The Raster-Op Problem                        | 838        |
| Looking at the System Palette                | 839        |
| <b>PALETTE ANIMATION</b>                     | <b>850</b> |
| The Bouncing Ball                            | 851        |
| One-Entry Palette Animation                  | 859        |
| Engineering Applications                     | 865        |
| <b>PALETTES AND REAL-WORLD IMAGES</b>        | <b>870</b> |
| Palettes and Packed DIBs                     | 870        |
| The All-Purpose Palette                      | 883        |
| The Halftone Palette                         | 890        |
| Indexing Palette Colors                      | 896        |
| Palettes and Bitmap Objects                  | 902        |
| Palettes and DIB Sections                    | 909        |
| <b>A LIBRARY FOR DIBs</b>                    | <b>915</b> |
| The DIBSTRUCT Structure                      | 917        |
| The Information Functions                    | 918        |
| Reading and Writing Pixels                   | 927        |
| Creating and Converting                      | 932        |
| The DIBHELP Header File and Macros           | 946        |
| The DIBBLE Program                           | 949        |
| Simple Palettes; Optimized Palettes          | 975        |
| Converting Formats                           | 991        |

|                   |                                     |             |
|-------------------|-------------------------------------|-------------|
| <i>Chapter 17</i> | <b>Text and Fonts</b>               | <b>997</b>  |
|                   | <b>SIMPLE TEXT OUTPUT</b>           | <b>997</b>  |
|                   | The Text Drawing Functions          | 997         |
|                   | Device Context Attributes for Text  | 1000        |
|                   | Using Stock Fonts                   | 1002        |
|                   | <b>BACKGROUND ON FONTS</b>          | <b>1002</b> |
|                   | The Types of Fonts                  | 1003        |
|                   | TrueType Fonts                      | 1004        |
|                   | Attributes or Styles?               | 1005        |
|                   | The Point Size                      | 1005        |
|                   | Leading and Spacing                 | 1005        |
|                   | The Logical Inch Problem            | 1006        |
|                   | <b>THE LOGICAL FONT</b>             | <b>1007</b> |
|                   | Logical Font Creation and Selection | 1007        |
|                   | The PICKFONT Program                | 1009        |
|                   | The Logical Font Structure          | 1026        |
|                   | The Font-Mapping Algorithm          | 1030        |
|                   | Finding Out About the Font          | 1031        |
|                   | Character Sets and Unicode          | 1033        |
|                   | The EZFONT System                   | 1035        |
|                   | Font Rotation                       | 1044        |
|                   | <b>FONT ENUMERATION</b>             | <b>1046</b> |
|                   | The Enumeration Functions           | 1046        |
|                   | The <i>ChooseFont</i> Dialog        | 1047        |
|                   | <b>PARAGRAPH FORMATTING</b>         | <b>1057</b> |
|                   | Simple Text Formatting              | 1058        |
|                   | Working with Paragraphs             | 1059        |
|                   | Previewing Printer Output           | 1069        |
|                   | <b>THE FUN AND FANCY STUFF</b>      | <b>1082</b> |
|                   | The GDI Path                        | 1082        |
|                   | Extended Pens                       | 1083        |
|                   | Four Sample Programs                | 1088        |
| <i>Chapter 18</i> | <b>Metafiles</b>                    | <b>1097</b> |
|                   | <b>THE OLD METAFILE FORMAT</b>      | <b>1098</b> |
|                   | Simple Use of Memory Metafiles      | 1098        |
|                   | Storing Metafiles on Disk           | 1101        |
|                   | Old Metafiles and the Clipboard     | 1102        |

|                                         |             |
|-----------------------------------------|-------------|
| <b>ENHANCED METAFILES</b>               | <b>1107</b> |
| The Basic Procedure                     | 1107        |
| Looking Inside                          | 1111        |
| Metafiles and GDI Objects               | 1118        |
| Metafiles and Bitmaps                   | 1123        |
| Enumerating the Metafile                | 1127        |
| Embedding Images                        | 1135        |
| An Enhanced Metafile Viewer and Printer | 1139        |
| Displaying Accurate Metafile Images     | 1150        |
| Scaling and Aspect Ratios               | 1161        |
| Mapping Modes in Metafiles              | 1163        |
| Mapping and Playing                     | 1166        |

## **Section III    *Advanced Topics***

### *Chapter 19*    **The Multiple-Document Interface**    **1173**

|                                      |             |
|--------------------------------------|-------------|
| <b>MDI CONCEPTS</b>                  | <b>1173</b> |
| The Elements of MDI                  | 1173        |
| MDI Support                          | 1175        |
| <b>A SAMPLE MDI IMPLEMENTATION</b>   | <b>1177</b> |
| Three Menus                          | 1190        |
| Program Initialization               | 1191        |
| Creating the Children                | 1192        |
| More Frame Window Message Processing | 1193        |
| The Child Document Windows           | 1194        |
| Cleaning Up                          | 1196        |

### *Chapter 20*    **Multitasking and Multithreading**    **1197**

|                                     |             |
|-------------------------------------|-------------|
| <b>MODES OF MULTITASKING</b>        | <b>1198</b> |
| Multitasking Under DOS?             | 1198        |
| Nonpreemptive Multitasking          | 1198        |
| PM and the Serialized Message Queue | 1200        |
| The Multithreading Solution         | 1200        |
| Multithreaded Architecture          | 1201        |
| Thread Hassles                      | 1202        |
| The Windows Advantage               | 1202        |
| New! Improved! Now with Threads!    | 1203        |

## CONTENTS

|                                                 |             |
|-------------------------------------------------|-------------|
| <b>WINDOWS MULTITHREADING</b>                   | <b>1204</b> |
| Random Rectangles Revisited                     | 1205        |
| The Programming Contest Problem                 | 1208        |
| The Multithreaded Solution                      | 1216        |
| Any Problems?                                   | 1226        |
| The Benefits of Sleep                           | 1226        |
| <b>THREAD SYNCHRONIZATION</b>                   | <b>1227</b> |
| The Critical Section                            | 1227        |
| <b>EVENT SIGNALING</b>                          | <b>1229</b> |
| The BIGJOB1 Program                             | 1230        |
| The Event Object                                | 1235        |
| <b>THREAD LOCAL STORAGE</b>                     | <b>1240</b> |
| <i>Chapter 21</i> <b>Dynamic-Link Libraries</b> | <b>1243</b> |
| <b>LIBRARY BASICS</b>                           | <b>1243</b> |
| Library: One Word, Many Meanings                | 1244        |
| A Simple DLL                                    | 1245        |
| The Library Entry and Exit Point                | 1248        |
| The Test Program                                | 1249        |
| Shared Memory in DLLs                           | 1252        |
| The STRPROG Program                             | 1258        |
| Sharing Data Among STRPROG Instances            | 1264        |
| <b>MISCELLANEOUS DLL TOPICS</b>                 | <b>1264</b> |
| Dynamic Linking Without Imports                 | 1265        |
| Resource-Only Libraries                         | 1266        |
| <i>Chapter 22</i> <b>Sound and Music</b>        | <b>1273</b> |
| <b>WINDOWS AND MULTIMEDIA</b>                   | <b>1273</b> |
| Multimedia Hardware                             | 1273        |
| An API Overview                                 | 1274        |
| Exploring MCI with TESTMCI                      | 1275        |
| MCITEXT and CD Audio                            | 1280        |
| <b>WAVEFORM AUDIO</b>                           | <b>1285</b> |
| Sound and Waveforms                             | 1285        |
| Pulse Code Modulation                           | 1286        |
| The Sampling Rate                               | 1287        |
| The Sample Size                                 | 1287        |

|                                                  |             |
|--------------------------------------------------|-------------|
| Generating Sine Waves in Software                | 1288        |
| A Digital Sound Recorder                         | 1299        |
| The MCI Alternative                              | 1312        |
| The MCI Command String Approach                  | 1320        |
| The Waveform Audio File Format                   | 1325        |
| Experimenting with Additive Synthesis            | 1326        |
| Waking Up to Waveform Audio                      | 1336        |
| <b>MIDI AND MUSIC</b>                            | <b>1345</b> |
| The Workings of MIDI                             | 1345        |
| The Program Change                               | 1347        |
| The MIDI Channel                                 | 1347        |
| MIDI Messages                                    | 1349        |
| An Introduction to MIDI Sequencing               | 1351        |
| Playing a MIDI Synthesizer from the PC Keyboard  | 1357        |
| A MIDI Drum Machine                              | 1375        |
| The Multimedia <i>time</i> Functions             | 1399        |
| RIFF File I/O                                    | 1402        |
| <i>Chapter 23</i> <b>A Taste of the Internet</b> | <b>1405</b> |
| <b>WINDOWS SOCKETS</b>                           | <b>1405</b> |
| Sockets and TCP/IP                               | 1406        |
| Network Time Services                            | 1406        |
| The NETTIME Program                              | 1407        |
| <b>WININET AND FTP</b>                           | <b>1421</b> |
| Overview of the FTP API                          | 1421        |
| The Update Demo                                  | 1423        |
| Index                                            | 1437        |

# Author's Note

Visit my web site at *www.cpetzold.com* for updated information regarding this book, including possible bug reports and new code listings. You can address mail regarding problems in this book to *charles@cpetzold.com*. Although I'll also try to answer any easy questions you may have, I can't make any promises. I'm usually pretty busy, and my cat refuses to learn the Windows API.

I'd like to thank everyone at Microsoft Press for another great job in putting together this book. I think this "10<sup>th</sup> Anniversary Edition" of *Programming Windows* is the best edition yet. Many other people at Microsoft (including some of the early developers of Microsoft Windows) also helped out when I was writing the earlier editions, and these fine people are listed in those editions.

Thanks also to my family and friends, and in particular those more recent friends (you know who you are!) whose support has made this book possible. To you this book is dedicated.

**Charles Petzold**

October 5, 1998

# The Keyboard

The keyboard and the mouse are the two standard sources of user input in Microsoft Windows 98, often complementing each other with some overlap. The mouse is, of course, much more utilized in today's applications than those of a decade ago. We are even accustomed to using the mouse almost exclusively in some applications, such as games, drawing programs, music programs, and Web browsers. Yet while we could probably make do without the mouse, removing the keyboard from the average PC would be disastrous.

Compared with the other components of the personal computer, the keyboard has a positively ancient ancestry beginning with the first Remington typewriter in 1874. Early computer programmers used keyboards to punch holes in Hollerith cards and later used keyboards on dumb terminals to communicate directly with large mainframe computers. The PC has been expanded somewhat to include function keys, cursor positioning keys, and (usually) a separate numeric keypad, but the principles of typing are basically the same.

## KEYBOARD BASICS

You've probably already surmised how a Windows program gets keyboard input: Keyboard input is delivered to your program's window procedures in the form of messages. Indeed, when first learning about messages, the keyboard is an obvious example of the type of information that messages might deliver to applications.

There are eight different messages that Windows uses to indicate various keyboard events. This may seem like a lot, but (as we'll see) your program can safely ignore at least half of them. Also, in most cases, the keyboard information encoded in these messages is probably more than your program needs. Part of the job of handling the keyboard is knowing which messages are important and which are not.

## Ignoring the Keyboard

Although the keyboard is often the primary source of user input in Windows programs, your program does not need to act on every keyboard message it receives. Windows handles many keyboard functions itself.

For instance, you can usually ignore keystrokes that pertain to system functions. These keystrokes generally involve the Alt key. You do not need to monitor these actual keystrokes because Windows notifies a program of the *effect* of the keystrokes. (A program can monitor the keystrokes itself if it wants to, however.) The keystrokes that invoke a program's menu come through a window's window procedure, but they are usually passed on to *DefWindowProc* for default processing. Eventually, the window procedure gets a message indicating that a menu item has been selected. This is generally all the window procedure needs to know. (Menus are covered in Chapter 10.)

Many Windows programs use keyboard accelerators to invoke common menu items. The accelerators usually involve the Ctrl key in combination with a function key or a letter key (for example, Ctrl-S to save a file). These keyboard accelerators are defined in a program's resource script along with a program's menu, as we'll see in Chapter 10. Windows translates these keyboard accelerators into menu command messages. You don't have to do the translation yourself.

Dialog boxes also have a keyboard interface, but programs usually do not need to monitor the keyboard when a dialog box is active. The keyboard interface is handled by Windows, and Windows sends messages to your program about the effects of the keystrokes. Dialog boxes can contain edit controls for text input. These are generally small boxes in which the user types a character string. Windows handles all the edit control logic and gives your program the final contents of the edit control when the user is done. See Chapter 11 for more on dialog boxes.

Edit controls don't have to be limited to a single line, and they don't have to be located only in dialog boxes. A multiline edit control in your program's main window can function as a rudimentary text editor. (This is shown in the POPPAD programs in Chapters 9, 10, 11, and 13.) And Windows even has a fancier rich-text edit control that lets you edit and display formatted text. (See */Platform SDK/User Interface Services/Controls/Rich Edit Controls*.)

You'll also find that when structuring your Windows programs, you can use child window controls to process keyboard and mouse input to deliver a higher level of information back to the parent window. Accumulate enough of these controls and you'll never have to be bothered with processing keyboard messages at all.

## Who's Got the Focus?

Like all personal computer hardware, the keyboard must be shared by all applications running under Windows. Some applications might have more than one window, and the keyboard must be shared by all the windows within the application.

As you'll recall, the `MSG` structure that a program uses to retrieve messages from the message queue includes a *hwnd* field. This field indicates the handle of the window that is to receive the message. The *DispatchMessage* function in the message loop sends that message to the window procedure associated with the window for which the message is intended. When a key on the keyboard is pressed, only one window procedure receives a keyboard message, and this message includes a handle to the window that is to receive the message.

The window that receives a particular keyboard event is the window that has the input focus. The concept of input focus is closely related to the concept of the active window. The window with the input focus is either the active window or a descendant window of the active window—that is, a child of the active window, or a child of a child of the active window, and so forth.

The active window is usually easy to identify. It is always a top-level window—that is, its parent window handle is `NULL`. If the active window has a title bar, Windows highlights the title bar. If the active window has a dialog frame (a form most commonly seen in dialog boxes) instead of a title bar, Windows highlights the frame. If the active window is currently minimized, Windows highlights its entry in the task bar by showing it as a depressed button.

If the active window has child windows, the window with the input focus can be either the active window or one of its descendants. The most common child windows are controls such as push buttons, radio buttons, check boxes, scroll bars, edit boxes, and list boxes that appear in dialog boxes. Child windows are never themselves active windows. A child window can have the input focus only if it is a descendent of the active window. Child window controls indicate that they have the input focus generally by displaying a flashing caret or a dotted line.

Sometimes no window has the input focus. This is the case if all your programs have been minimized. Windows continues to send keyboard messages to the active window, but these messages are in a different form from keyboard messages sent to active windows that are not minimized.

A window procedure can determine when its window has the input focus by trapping `WM_SETFOCUS` and `WM_KILLFOCUS` messages. `WM_SETFOCUS` indicates that the window is receiving the input focus, and `WM_KILLFOCUS` signals that the window is losing the input focus. I'll have more to say about these messages later in this chapter.

## Queues and Synchronization

As the user presses and releases keys on the keyboard, Windows and the keyboard device driver translate the hardware scan codes into formatted messages. However, these messages are not placed in an application's message queue right away. Instead, Windows stores these messages in something called the *system message queue*. The system message queue is a single message queue maintained by Windows specifically for the preliminary

storage of user input from the keyboard and the mouse. Windows will take the next message from the system message queue and place it in an application's message queue only when a Windows application has finished processing a previous user input message.

The reasons for this two-step process—storing messages first in the system message queue and then passing them to the application message queue—involves synchronization. As we just learned, the window that is supposed to receive keyboard input is the window with the input focus. A user can be typing faster than an application can handle the keystrokes, and a particular keystroke might have the effect of switching focus from one window to another. Subsequent keystrokes should then go to another window. But they won't if the subsequent keystrokes have already been addressed with a destination window and placed in an application message queue.

## Keystrokes and Characters

The messages that an application receives from Windows about keyboard events distinguish between keystrokes and characters. This is in accordance with the two ways you can view the keyboard.

First, you can think of the keyboard as a collection of keys. The keyboard has only one key labeled "A." Pressing that key is a keystroke. Releasing that key is also considered a keystroke. But the keyboard is also an input device that generates displayable characters or control characters. The "A" key can generate several different characters depending on the status of the Ctrl, Shift, and Caps Lock keys. Normally, the character is a lowercase "a." If the Shift key is down or Caps Lock is toggled on, the character is an uppercase "A." If Ctrl is down, the character is a Ctrl-A (which has meaning in ASCII but in Windows is probably a keyboard accelerator if anything). On some keyboards, the "A" keystroke might be preceded by a dead-character key or by Shift, Ctrl, or Alt in various combinations. The combinations could generate a lowercase or uppercase letter with an accent mark, such as à, á, â, ã, Ä, or Å.

For keystroke combinations that result in displayable characters, Windows sends a program both keystroke messages and character messages. Some keys do not generate characters. These include the shift keys, the function keys, the cursor movement keys, and special keys such as Insert and Delete. For these keys, Windows generates only keystroke messages.

## KEYSTROKE MESSAGES

When you press a key, Windows places either a WM\_KEYDOWN or WM\_SYSKEYDOWN message in the message queue of the window with the input focus. When you release a key, Windows places either a WM\_KEYUP or WM\_SYSKEYUP message in the message queue.

|                      | <i>Key Pressed</i> | <i>Key Released</i> |
|----------------------|--------------------|---------------------|
| Nonsystem Keystroke: | WM_KEYDOWN         | WM_KEYUP            |
| System Keystroke:    | WM_SYSKEYDOWN      | WM_SYSKEYUP         |

Usually the up and down messages occur in pairs. However, if you hold down a key so that the typematic (autorepeat) action takes over, Windows sends the window procedure a series of WM\_KEYDOWN (or WM\_SYSKEYDOWN) messages and a single WM\_KEYUP (or WM\_SYSKEYUP) message when the key is finally released. Like all queued messages, keystroke messages are time-stamped. You can retrieve the relative time a key was pressed or released by calling *GetMessageTime*.

## System and Nonsystem Keystrokes

The “SYS” in WM\_SYSKEYDOWN and WM\_SYSKEYUP stands for “system” and refers to keystrokes that are more important to Windows than to Windows applications. The WM\_SYSKEYDOWN and WM\_SYSKEYUP messages are usually generated for keys typed in combination with the Alt key. These keystrokes invoke options on the program’s menu or system menu, or they are used for system functions such as switching the active window (Alt-Tab or Alt-Esc) or for system menu accelerators (Alt in combination with a function key such as Alt-F4 to close an application). Programs usually ignore the WM\_SYSKEYUP and WM\_SYSKEYDOWN messages and pass them to *DefWindowProc*. Because Windows takes care of all the Alt-key logic, you really have no need to trap these messages. Your window procedure will eventually receive other messages concerning the result of these keystrokes (such as a menu selection). If you want to include code in your window procedure to trap the system keystroke messages (as we will do in the KEYVIEW1 and KEYVIEW2 programs shown later in this chapter), pass the messages to *DefWindowProc* after you process them so that Windows can still use them for their intended purposes.

But think about this for a moment. Almost everything that affects your program’s window passes through your window procedure first. Windows does something with the message only if you pass the message to *DefWindowProc*. For instance, if you add the lines

```
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
case WM_SYSCHAR:
    return 0 ;
```

to a window procedure, you effectively disable all Alt-key operations when your program’s main window has the input focus. (I’ll discuss the WM\_SYSCHAR message later in this chapter.) This includes Alt-Tab, Alt-Esc, and menu operations. Although I doubt you would want to do this, I trust you sense the power inherent in the window procedure.

The WM\_KEYDOWN and WM\_KEYUP messages are usually generated for keys that are pressed and released without the Alt key. Your program can use or discard these keystroke messages. Windows doesn't care about them.

For all four keystroke messages, *wParam* is a virtual key code that identifies the key being pressed or released and *lParam* contains other data pertaining to the keystroke.

## Virtual Key Codes

The virtual key code is stored in the *wParam* parameter of the WM\_KEYDOWN, WM\_KEYUP, WM\_SYSKEYDOWN, and WM\_SYSKEYUP messages. This code identifies the key being pressed or released.

Ah, that ubiquitous word “virtual.” Don't you love it? It's supposed to refer to something that exists in the mind rather than in the real world, but only veteran programmers of DOS assembly language applications might figure out why the key codes so essential to Windows keyboard processing are considered virtual rather than real.

To old-time programmers, the *real* keyboard codes are generated by the hardware of the physical keyboard. These are referred to in the Windows documentation as *scan codes*. On IBM compatibles, a scan code of 16 is the Q key, 17 is the W key, 18 is E, 19 is R, 20 is T, 21 is Y, and so on. You get the idea—the scan codes are based on the physical layout of the keyboard. The developers of Windows considered these scan codes too device-dependent. They thus attempted to treat the keyboard in a device-independent manner by defining the so-called virtual key codes. Some of these virtual key codes cannot be generated on IBM compatibles but may be found on other manufacturer's keyboards, or perhaps on keyboards of the future.

The virtual key codes you use most often have names beginning with VK\_ defined in the WINUSER.H header file. The tables below show these names along with the numeric values (in both decimal and hexadecimal) and the IBM-compatible keyboard key that corresponds to the virtual key. The tables also indicate whether these keys are required for Windows to run properly. The tables show the virtual key codes in numeric order.

Three of the first four virtual key codes refer to mouse buttons:

| <i>Decimal</i> | <i>Hex</i> | <i>WINUSER.H<br/>Identifier</i> | <i>Required?</i> | <i>IBM-Compatible<br/>Keyboard</i> |
|----------------|------------|---------------------------------|------------------|------------------------------------|
| 1              | 01         | VK_LBUTTON                      |                  | Mouse Left Button                  |
| 2              | 02         | VK_RBUTTON                      |                  | Mouse Right Button                 |
| 3              | 03         | VK_CANCEL                       | ✓                | Ctrl-Break                         |
| 4              | 04         | VK_MBUTTON                      |                  | Mouse Middle Button                |

You will never get these mouse button codes in the keyboard messages. They are found in mouse messages, as we'll see in the next chapter. The VK\_CANCEL code is the only virtual

key code that involves pressing two keys at once (Ctrl-Break). Windows applications generally do not use this key.

Several of the following keys—Backspace, Tab, Enter, Escape, and Spacebar—are commonly used by Windows programs. However, Windows programs generally use character messages (rather than keystroke messages) to process these keys.

| <i>Decimal</i> | <i>Hex</i> | <i>WINUSER.H<br/>Identifier</i> | <i>Required?</i> | <i>IBM-Compatible<br/>Keyboard</i>      |
|----------------|------------|---------------------------------|------------------|-----------------------------------------|
| 8              | 08         | VK_BACK                         | ✓                | Backspace                               |
| 9              | 09         | VK_TAB                          | ✓                | Tab                                     |
| 12             | 0C         | VK_CLEAR                        |                  | Numeric keyboard 5 with<br>Num Lock OFF |
| 13             | 0D         | VK_RETURN                       | ✓                | Enter (either one)                      |
| 16             | 10         | VK_SHIFT                        | ✓                | Shift (either one)                      |
| 17             | 11         | VK_CONTROL                      | ✓                | Ctrl (either one)                       |
| 18             | 12         | VK_MENU                         | ✓                | Alt (either one)                        |
| 19             | 13         | VK_PAUSE                        |                  | Pause                                   |
| 20             | 14         | VK_CAPITAL                      | ✓                | Caps Lock                               |
| 27             | 1B         | VK_ESCAPE                       | ✓                | Esc                                     |
| 32             | 20         | VK_SPACE                        | ✓                | Spacebar                                |

Also, Windows programs usually do not need to monitor the status of the Shift, Ctrl, or Alt keys.

The first eight codes listed in the following table are perhaps the most commonly used virtual key codes along with VK\_INSERT and VK\_DELETE:

| <i>Decimal</i> | <i>Hex</i> | <i>WINUSER.H<br/>Identifier</i> | <i>Required?</i> | <i>IBM-Compatible<br/>Keyboard</i> |
|----------------|------------|---------------------------------|------------------|------------------------------------|
| 33             | 21         | VK_PRIOR                        | ✓                | Page Up                            |
| 34             | 22         | VK_NEXT                         | ✓                | Page Down                          |
| 35             | 23         | VK_END                          | ✓                | End                                |
| 36             | 24         | VK_HOME                         | ✓                | Home                               |
| 37             | 25         | VK_LEFT                         | ✓                | Left Arrow                         |
| 38             | 26         | VK_UP                           | ✓                | Up Arrow                           |
| 39             | 27         | VK_RIGHT                        | ✓                | Right Arrow                        |
| 40             | 28         | VK_DOWN                         | ✓                | Down Arrow                         |

(continued)

*continued*

| <i>Decimal</i> | <i>Hex</i> | <i>WINUSER.H<br/>Identifier</i> | <i>Required?</i> | <i>IBM-Compatible<br/>Keyboard</i> |
|----------------|------------|---------------------------------|------------------|------------------------------------|
| 41             | 29         | VK_SELECT                       |                  |                                    |
| 42             | 2A         | VK_PRINT                        |                  |                                    |
| 43             | 2B         | VK_EXECUTE                      |                  |                                    |
| 44             | 2C         | VK_SNAPSHOT                     |                  | Print Screen                       |
| 45             | 2D         | VK_INSERT                       | ✓                | Insert                             |
| 46             | 2E         | VK_DELETE                       | ✓                | Delete                             |
| 47             | 2F         | VK_HELP                         |                  |                                    |

Notice that many of the names (such as VK\_PRIOR and VK\_NEXT) are unfortunately quite different from the labels on the keys and also not consistent with the identifiers used in scroll bars. The Print Screen key is largely ignored by Windows applications. Windows itself responds to the key by storing a bitmap copy of the video display into the clipboard. VK\_SELECT, VK\_PRINT, VK\_EXECUTE, and VK\_HELP might be found on a hypothetical keyboard that few of us have ever seen.

Windows also includes virtual key codes for the letter keys and number keys on the main keyboard. (The number pad is handled separately.)

| <i>Decimal</i> | <i>Hex</i> | <i>WINUSER.H<br/>Identifier</i> | <i>Required?</i> | <i>IBM-Compatible<br/>Keyboard</i> |
|----------------|------------|---------------------------------|------------------|------------------------------------|
| 48–57          | 30–39      | None                            | ✓                | 0 through 9 on<br>main keyboard    |
| 65–90          | 41–5A      | None                            | ✓                | A through Z                        |

Notice that the virtual key codes are the ASCII codes for the numbers and letters. Windows programs almost never use these virtual key codes; instead, the programs rely on character messages for ASCII characters.

The following keys are generated from the Microsoft Natural Keyboard and compatibles:

| <i>Decimal</i> | <i>Hex</i> | <i>WINUSER.H<br/>Identifier</i> | <i>Required?</i> | <i>Microsoft Natural<br/>Keyboard</i> |
|----------------|------------|---------------------------------|------------------|---------------------------------------|
| 91             | 5B         | VK_LWIN                         |                  | Left Windows key                      |
| 92             | 5C         | VK_RWIN                         |                  | Right Windows key                     |
| 93             | 5D         | VK_APPS                         |                  | Applications key                      |

The VK\_LWIN and VK\_RWIN keys are handled by Windows to open the Start menu or (in older versions) to launch the Task Manager. Together, they can log on or off Windows (in Microsoft Windows NT only), or log on or off a network (in Windows for Workgroups). Applications can process the application key by displaying help information or shortcuts.

The following codes are for the keys on the numeric keypad (if present):

| <i>Decimal</i> | <i>Hex</i> | <i>WINUSER.H Identifier</i>   | <i>Required?</i> | <i>IBM-Compatible Keyboard</i>              |
|----------------|------------|-------------------------------|------------------|---------------------------------------------|
| 96–105         | 60–69      | VK_NUMPAD0 through VK_NUMPAD9 |                  | Numeric keypad 0 through 9 with Num Lock ON |
| 106            | 6A         | VK_MULTIPLY                   |                  | Numeric keypad *                            |
| 107            | 6B         | VK_ADD                        |                  | Numeric keypad +                            |
| 108            | 6C         | VK_SEPARATOR                  |                  |                                             |
| 109            | 6D         | VK_SUBTRACT                   |                  | Numeric keypad –                            |
| 110            | 6E         | VK_DECIMAL                    |                  | Numeric keypad .                            |
| 111            | 6F         | VK_DIVIDE                     |                  | Numeric keypad /                            |

Finally, although most keyboards have 12 function keys, Windows requires only 10 but has numeric identifiers for 24. Again, programs generally use the function keys as keyboard accelerators so they usually don't process the keystrokes in this table:

| <i>Decimal</i> | <i>Hex</i> | <i>WINUSER.H Identifier</i> | <i>Required?</i> | <i>IBM-Compatible Keyboard</i> |
|----------------|------------|-----------------------------|------------------|--------------------------------|
| 112–121        | 70–79      | VK_F1 through VK_F10        | ✓                | Function keys F1 through F10   |
| 122–135        | 7A–87      | VK_F11 through VK_F24       |                  | Function keys F11 through F24  |
| 144            | 90         | VK_NUMLOCK                  |                  | Num Lock                       |
| 145            | 91         | VK_SCROLL                   |                  | Scroll Lock                    |

Some other virtual key codes are defined, but they are reserved for keys specific to nonstandard keyboards or for keys most commonly found on mainframe terminals. Check */Platform SDK/User Interface Services/User Input/Virtual-Key Codes* for a complete list.

## The *lParam* Information

In the four keystroke messages (WM\_KEYDOWN, WM\_KEYUP, WM\_SYSKEYDOWN, and WM\_SYSKEYUP), the *wParam* message parameter contains the virtual key code as described

above, and the *lParam* message parameter contains other information useful in understanding the keystroke. The 32 bits of *lParam* are divided into six fields as shown in Figure 6-1.

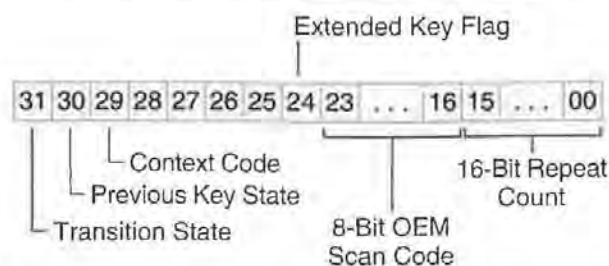


**Figure 6-1.** The six keystroke-message fields of the *lParam* variable.

### Repeat Count

The repeat count is the number of keystrokes represented by the message. In most cases, this will be set to 1. However, if a key is held down and your window procedure is not fast enough to process key-down messages at the typematic rate (which you can set in the Keyboard applet in the Control Panel), Windows combines several WM\_KEYDOWN or WM\_SYSKEYDOWN messages into a single message and increases the Repeat Count field accordingly. The Repeat Count is always 1 for a WM\_KEYUP or WM\_SYSKEYUP message.

Because a Repeat Count greater than 1 indicates that typematic keystrokes are occurring faster than your program can process them, you may want to ignore the Repeat Count when processing the keyboard messages. Almost everyone has had the experience of “overscrolling” a word-processing document or spreadsheet because extra keystrokes have accumulated. If your program ignores the Repeat Count in cases where your program spends some time processing each keystroke, you can eliminate this problem. However, in other cases you will want to use the Repeat Count. You may want to try using the programs both ways and see which feels the most natural.

### OEM Scan Code

The OEM Scan Code is the code generated by the hardware of the keyboard. This is familiar to middle-aged assembly language programmers as the value obtained from the ROM BIOS services of PC compatibles. (OEM refers to the Original Equipment Manufacturer of the PC and in this context is synonymous with “IBM Standard.”) We don’t need this stuff anymore. Windows programs can almost always ignore the OEM Scan Code except when dependent on the physical layout of the keyboard, such as the KBMIDI program in Chapter 22.

### Extended Key Flag

The Extended Key Flag is 1 if the keystroke results from one of the additional keys on the IBM enhanced keyboard. (The enhanced keyboard has 101 or 102 keys. Function keys are across the top. Cursor movement keys are separate from the numeric keypad, but the numeric keypad also duplicates the cursor movement keys.) This flag is set to 1 for the Alt and Ctrl keys at the right of the keyboard, the cursor movement keys (including Insert and Delete) that are not part of the numeric keypad, the slash (/) and Enter keys

on the numeric keypad, and the Num Lock key. Windows programs generally ignore the Extended Key Flag.

### Context Code

The Context Code is 1 if the Alt key is depressed during the keystroke. This bit will always be 1 for the WM\_SYSKEYUP and WM\_SYSKEYDOWN messages and 0 for the WM\_KEYUP and WM\_KEYDOWN messages, with two exceptions:

- If the active window is minimized, it does not have the input focus. All keystrokes generate WM\_SYSKEYUP and WM\_SYSKEYDOWN messages. If the Alt key is not pressed, the Context Code field is set to 0. Windows uses WM\_SYSKEYUP and WM\_SYSKEYDOWN messages so that a minimized active window doesn't process these keystrokes.
- On some foreign-language keyboards, certain characters are generated by combining Shift, Ctrl, or Alt with another key. In these cases, the Context Code is set to 1 but the messages are not system keystroke messages.

### Previous Key State

The Previous Key State is 0 if the key was previously up and 1 if the key was previously down. It is always set to 1 for a WM\_KEYUP or WM\_SYSKEYUP message, but it can be 0 or 1 for a WM\_KEYDOWN or WM\_SYSKEYDOWN message. A 1 indicates second and subsequent messages that are the result of typematic repeats.

### Transition State

The Transition State is 0 if the key is being pressed and 1 if the key is being released. The field is set to 0 for a WM\_KEYDOWN or WM\_SYSKEYDOWN message and to 1 for a WM\_KEYUP or WM\_SYSKEYUP message.

## Shift States

When you process a keystroke message, you may need to know whether any of the shift keys (Shift, Ctrl, and Alt) or toggle keys (Caps Lock, Num Lock, and Scroll Lock) are pressed. You can obtain this information by calling the *GetKeyState* function. For instance:

```
iState = GetKeyState (VK_SHIFT) ;
```

The *iState* variable will be negative (that is, the high bit is set) if the Shift key is down. The value returned from

```
iState = GetKeyState (VK_CAPITAL) ;
```

has the low bit set if the Caps Lock key is toggled on. This bit will agree with the little light on the keyboard.

Generally, you'll use *GetKeyState* with the virtual key codes VK\_SHIFT, VK\_CONTROL, and VK\_MENU (which you'll recall indicates the Alt key). You can also use the following

identifiers with *GetKeyState* to determine if the left or right Shift, Ctrl, or Alt keys are pressed: VK\_LSHIFT, VK\_RSHIFT, VK\_LCONTROL, VK\_RCONTROL, VK\_LMENU, VK\_RMENU. These identifiers are used *only* with *GetKeyState* and *GetAsyncKeyState* (described below).

You can also obtain the state of the mouse buttons using the virtual key codes VK\_LBUTTON, VK\_RBUTTON, and VK\_MBUTTON. However, most Windows programs that need to monitor a combination of mouse buttons and keystrokes usually do it the other way around—by checking keystrokes when they receive a mouse message. In fact, shift-state information is conveniently included in the mouse messages, as we'll see in the next chapter.

Be careful with *GetKeyState*. It is not a real-time keyboard status check. Rather, it reflects the keyboard status up to and including the current message being processed. For the most part, this is exactly what you want. If you need to determine if the user typed Shift-Tab, you can call *GetKeyState* with the VK\_SHIFT parameter while processing the WM\_KEYDOWN message for the Tab key. If the return value of *GetKeyState* is negative, you know that the Shift key was pressed *before* the Tab key. And it doesn't matter if the Shift key has already been released by the time you get around to processing the Tab key. You know that the Shift key was down when Tab was pressed.

*GetKeyState* does not let you retrieve keyboard information independent of normal keyboard messages. For instance, you may feel a need to hold up processing in your window procedure until the user presses the F1 function key:

```
while (GetKeyState (VK_F1) >= 0) ;    // WRONG !!!
```

Don't do it! This is guaranteed to hang your program (unless, of course, the WM\_KEYDOWN message for F1 was retrieved from the message queue before you executed the statement). If you really need to know the current real-time state of a key, you can use *GetAsyncKeyState*.

## Using Keystroke Messages

A Windows program gets information about each and every keystroke that occurs while the program is running. This is certainly helpful. However, most Windows programs ignore all but a few keystroke messages. The WM\_SYSKEYDOWN and WM\_SYSKEYUP messages are for Windows system functions, and you don't need to look at them. If you process WM\_KEYDOWN messages, you can usually also ignore WM\_KEYUP messages.

Windows programs generally use WM\_KEYDOWN messages for keystrokes that do not generate characters. Although you may think that it's possible to use keystroke messages in combination with shift-state information to translate keystroke messages into characters, don't do it. You'll have problems with non-English keyboards. For example, if you get a WM\_KEYDOWN message with *wParam* equal to 0x33, you know the user pressed the 3 key. So far, so good. If you use *GetKeyState* and find out that the Shift key is down, you might assume that the user is typing a pound sign (#). Not necessarily. A British user is typing another type of pound sign, the one that looks like this: £.

The WM\_KEYDOWN messages are most useful for the cursor movement keys, the function keys, Insert, and Delete. However, Insert, Delete, and the function keys often appear as menu accelerators. Because Windows translates menu accelerators into menu command messages, you don't have to process the keystrokes themselves.

It was common for pre-Windows applications for MS-DOS to use the function keys extensively in combination with the Shift, Ctrl, and Alt keys. You can do something similar in your Windows programs (indeed, Microsoft Word uses the function keys extensively as command short cuts), but it's not really recommended. If you want to use the function keys, they should duplicate menu commands. One objective in Windows is to provide a user interface that doesn't require memorization or consultation of complex command charts.

So, it comes down to this: Most of the time, you will process WM\_KEYDOWN messages only for cursor movement keys, and sometimes for Insert and Delete. When you use these keys, you can check the Shift-key and Ctrl-key states through *GetKeyState*. Windows programs often use the Shift key in combination with the cursor keys to extend a selection in (for instance) a word-processing document. The Ctrl key is often used to alter the meaning of the cursor key. For example, Ctrl in combination with the Right Arrow key might mean to move the cursor one word to the right.

One of the best ways to determine how to use the keyboard in your application is to examine how the keyboard is used in existing popular Windows programs. If you don't like those definitions, you are free to do something different. But keep in mind that doing so might be detrimental to a user's ability to learn your program quickly.

## Enhancing SYSMETS for the Keyboard

The three versions of the SYSMETS program in Chapter 4 were written without any knowledge of the keyboard. We were able to scroll the text only by using the mouse on the scroll bars. Now that we know how to process keystroke messages, let's add a keyboard interface to the program. This is obviously a job for cursor movement keys. We'll use most of these keys (Home, End, Page Up, Page Down, Up Arrow, and Down Arrow) for vertical scrolling. The Left Arrow and Right Arrow keys can take care of the less important horizontal scrolling.

One obvious way to create a keyboard interface is to add some WM\_KEYDOWN logic to the window procedure that parallels and essentially duplicates all the WM\_VSCROLL and WM\_HSCROLL logic. However, this is unwise, because if we ever wanted to change the scroll bar logic we'd have to make the same changes in WM\_KEYDOWN.

Wouldn't it be better to simply translate each of these WM\_KEYDOWN messages into an equivalent WM\_VSCROLL or WM\_HSCROLL message? Then we could perhaps fool *WndProc* into thinking that it's getting a scroll bar message, perhaps by sending a phony message to the window procedure.

Windows lets you do this. The function is named *SendMessage*, and it takes the same parameters as those passed to the window procedure:

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

When you call *SendMessage*, Windows calls the window procedure whose window handle is *hwnd*, passing to it these four function arguments. When the window procedure has completed processing the message, Windows returns control to the next statement following the *SendMessage* call. The window procedure you send the message to could be the same window procedure, another window procedure in the same program, or even a window procedure in another application.

Here's how we might use *SendMessage* for processing WM\_KEYDOWN codes in the SYSMETS program:

```
case WM_KEYDOWN:
    switch (wParam)
    {
    case VK_HOME:
        SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;
        break ;

    case VK_END:
        SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;
        break ;

    case VK_PRIOR:
        SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0) ;
        break ;
```

And so forth. You get the general idea. Our goal was to add a keyboard interface to the scroll bars, and that's exactly what we've done. We've made the cursor movement keys duplicate scroll bar logic by actually sending the window procedure a scroll bar message. Now you can see why I included SB\_TOP and SB\_BOTTOM processing for WM\_VSCROLL messages in the SYSMETS3 program. It wasn't used then, but it's used now for processing the Home and End keys. The SYSMETS4 program, shown in Figure 6-2, incorporates these changes. You'll also need the SYSMETS.H file from Chapter 4 to compile this program.

### SYSMETS4.C

```
/*-----
SYSMETS4.C -- System Metrics Display Program No. 4
              (c) Charles Petzold, 1998
-----*/

#include <windows.h>
#include "sysmets.h"
```

Figure 6-2. The SYSMETS4 program.

```

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

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("SysMets4") ;
    HWND          hwnd ;
    MSG           msg ;
    WNDCLASS       wndclass ;

    wndclass.style      = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra  = 0 ;
    wndclass.cbWndExtra  = 0 ;
    wndclass.hInstance  = hInstance ;
    wndclass.hIcon       = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor     = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = szAppName ;

    if (!RegisterClass (&wndclass))
    {
        MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                    szAppName, MB_ICONERROR) ;
        return 0 ;
    }

    hwnd = CreateWindow (szAppName, TEXT ("Get System Metrics No. 4"),
                        WS_OVERLAPPEDWINDOW | WS_VSCROLL | WS_HSCROLL,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}

```

(continued)

Figure 6-2. *continued*

```

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static int    cxChar, cxCaps, cyChar, cxClient, cyClient, iMaxWidth ;
    HDC          hdc ;
    int          i, x, y, iVertPos, iHorzPos, iPaintBeg, iPaintEnd ;
    PAINTSTRUCT ps ;
    SCROLLINFO si ;
    TCHAR        szBuffer[10] ;
    TEXTMETRIC tm ;

    switch (message)
    {
    case WM_CREATE:
        hdc = GetDC (hwnd) ;

        GetTextMetrics (hdc, &tm) ;
        cxChar = tm.tmAveCharWidth ;
        cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2 ;
        cyChar = tm.tmHeight + tm.tmExternalLeading ;

        ReleaseDC (hwnd, hdc) ;

        // Save the width of the three columns

        iMaxWidth = 40 * cxChar + 22 * cxCaps ;
        return 0 ;

    case WM_SIZE:
        cxClient = LOWORD (lParam) ;
        cyClient = HIWORD (lParam) ;

        // Set vertical scroll bar range and page size

        si.cbSize = sizeof (si) ;
        si.fMask = SIF_RANGE | SIF_PAGE ;
        si.nMin = 0 ;
        si.nMax = NUMLINES - 1 ;
        si.nPage = cyClient / cyChar ;
        SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;

        // Set horizontal scroll bar range and page size

        si.cbSize = sizeof (si) ;
        si.fMask = SIF_RANGE | SIF_PAGE ;
        si.nMin = 0 ;
        si.nMax = 2 + iMaxWidth / cxChar ;
        si.nPage = cxClient / cxChar ;
    }
}

```

```

SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
return 0 ;

case WM_VSCROLL:
    // Get all the vertical scroll bar information

    si.cbSize = sizeof (si) ;
    si.fMask = SIF_ALL ;
    GetScrollInfo (hwnd, SB_VERT, &si) ;

    // Save the position for comparison later on

    iVertPos = si.nPos ;

    switch (LOWORD (wParam))
    {
    case SB_TOP:
        si.nPos = si.nMin ;
        break ;

    case SB_BOTTOM:
        si.nPos = si.nMax ;
        break ;

    case SB_LINEUP:
        si.nPos -= 1 ;
        break ;

    case SB_LINEDOWN:
        si.nPos += 1 ;
        break ;

    case SB_PAGEUP:
        si.nPos -= si.nPage ;
        break ;

    case SB_PAGEDOWN:
        si.nPos += si.nPage ;
        break ;

    case SB_THUMBTRACK:
        si.nPos = si.nTrackPos ;
        break ;

    default:
        break ;
    }
}

```

(continued)

Figure 6-2. *continued*

```

        // Set the position and then retrieve it. Due to adjustments
        // by Windows it might not be the same as the value set.

        si.fMask = SIF_POS ;
        SetScrollInfo (hwnd, SB_VERT, &si, TRUE) ;
        GetScrollInfo (hwnd, SB_VERT, &si) ;

        // If the position has changed, scroll the window and update it

        if (si.nPos != iVertPos)
        {
            ScrollWindow (hwnd, 0, cyChar * (iVertPos - si.nPos),
                        NULL, NULL) ;
            UpdateWindow (hwnd) ;
        }
        return 0 ;

case WM_HSCROLL:
    // Get all the vertical scroll bar information

    si.cbSize = sizeof (si) ;
    si.fMask = SIF_ALL ;

    // Save the position for comparison later on

    GetScrollInfo (hwnd, SB_HORZ, &si) ;
    iHorzPos = si.nPos ;

    switch (LOWORD (wParam))
    {
    case SB_LINELEFT:
        si.nPos -= 1 ;
        break ;

    case SB_LINERIGHT:
        si.nPos += 1 ;
        break ;

    case SB_PAGELEFT:
        si.nPos -= si.nPage ;
        break ;

    case SB_PAGERIGHT:
        si.nPos += si.nPage ;
        break ;
    }

```

```

case SB_THUMBPOSITION:
    si.nPos = si.nTrackPos ;
    break ;

default:
    break ;
}

// Set the position and then retrieve it. Due to adjustments
// by Windows it might not be the same as the value set.

si.fMask = SIF_POS ;
SetScrollInfo (hwnd, SB_HORZ, &si, TRUE) ;
GetScrollInfo (hwnd, SB_HORZ, &si) ;

// If the position has changed, scroll the window

if (si.nPos != iHorzPos)
{
    ScrollWindow (hwnd, cxChar * (iHorzPos - si.nPos), 0,
        NULL, NULL) ;
}
return 0 ;

case WM_KEYDOWN:
    switch (wParam)
    {
    case VK_HOME:
        SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;
        break ;

    case VK_END:
        SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;
        break ;

    case VK_PRIOR:
        SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0) ;
        break ;

    case VK_NEXT:
        SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN, 0) ;
        break ;

    case VK_UP:
        SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0) ;
        break ;
    }
}

```

*(continued)*

Figure 6-2. *continued*

```

        case VK_DOWN:
            SendMessage (hwnd, WM_VSCROLL, SB_LINEDOWN, 0) ;
            break ;

        case VK_LEFT:
            SendMessage (hwnd, WM_HSCROLL, SB_PAGEUP, 0) ;
            break ;

        case VK_RIGHT:
            SendMessage (hwnd, WM_HSCROLL, SB_PAGEDOWN, 0) ;
            break ;
    }
    return 0 ;

case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;

    // Get vertical scroll bar position

    si.cbSize = sizeof (si) ;
    si.fMask = SIF_POS ;
    GetScrollInfo (hwnd, SB_VERT, &si) ;
    iVertPos = si.nPos ;

    // Get horizontal scroll bar position

    GetScrollInfo (hwnd, SB_HORZ, &si) ;
    iHorzPos = si.nPos ;

    // Find painting limits

    iPaintBeg = max (0, iVertPos + ps.rcPaint.top / cyChar) ;
    iPaintEnd = min (NUMLINES - 1,
                     iVertPos + ps.rcPaint.bottom / cyChar) ;

    for (i = iPaintBeg ; i <= iPaintEnd ; i++)
    {
        x = cxChar * (1 - iHorzPos) ;
        y = cyChar * (i - iVertPos) ;

        TextOut (hdc, x, y,
                 sysmetrics[i].szLabel,
                 strlen (sysmetrics[i].szLabel)) ;

        TextOut (hdc, x + 22 * cxCaps, y,
                 sysmetrics[i].szDesc,
                 strlen (sysmetrics[i].szDesc)) ;
    }

```

```

        SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;

        TextOut (hdc, x + 22 * cxCaps + 40 * cxChar, y, szBuffer,
            wsprintf (szBuffer, TEXT ("%5d"),
                GetSystemMetrics (sysmetrics[i].iIndex))) ;

        SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
    }

    EndPaint (hwnd, &ps) ;
    return 0 ;

case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
}

```

## CHARACTER MESSAGES

Earlier in this chapter, I discussed the idea of translating keystroke messages into character messages by taking shift-state information into account. I warned you that shift-state information is not enough: you also need to know about country-dependent keyboard configurations. For this reason, you should not attempt to translate keystroke messages into character codes yourself. Instead, Windows does it for you. You've seen this code before:

```

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

```

This is a typical message loop that appears in *WinMain*. The *GetMessage* function fills in the *msg* structure fields with the next message from the queue. *DispatchMessage* calls the appropriate window procedure with this message.

Between these two functions is *TranslateMessage*, which takes on the responsibility of translating keystroke messages to character messages. If the keystroke message is *WM\_KEYDOWN* or *WM\_SYSKEYDOWN*, and if the keystroke in combination with the shift state produces a character, *TranslateMessage* places a character message in the message queue. This character message will be the next message that *GetMessage* retrieves from the queue after the keystroke message.

## The Four Character Messages

There are four character messages:

|                       | <i>Characters</i> | <i>Dead Characters</i> |
|-----------------------|-------------------|------------------------|
| Nonsystem Characters: | WM_CHAR           | WM_DEADCHAR            |
| System Characters:    | WM_SYSCHAR        | WM_SYSDEADCHAR         |

The WM\_CHAR and WM\_DEADCHAR messages are derived from WM\_KEYDOWN messages. The WM\_SYSCHAR and WM\_SYSDEADCHAR messages are derived from WM\_SYSKEYDOWN messages. (I'll discuss what a dead character is shortly.)

Here's the good news: In most cases, your Windows program can process the WM\_CHAR message while ignoring the other three character messages. The *lParam* parameter that accompanies the four character messages is the same as the *lParam* parameter for the keystroke message that generated the character code message. However, the *wParam* parameter is not a virtual key code. Instead, it is an ANSI or Unicode character code.

These character messages are the first messages we've encountered that deliver text to the window procedure. They're not the only ones. Other messages are accompanied by entire zero-terminated text strings. How does the window procedure know whether this character data is 8-bit ANSI or 16-bit Unicode? It's simple: Any window procedure associated with a window class that you register with *RegisterClassA* (the ANSI version of *RegisterClass*) gets messages that contain ANSI character codes. Messages to window procedures that were registered with *RegisterClassW* (the wide-character version of *RegisterClass*) come with Unicode character codes. If your program registers its window class using *RegisterClass*, that's really *RegisterClassW* if the UNICODE identifier was defined and *RegisterClassA* otherwise.

Unless you're explicitly doing mixed coding of ANSI and Unicode functions and window procedures, the character code delivered with the WM\_CHAR message (and the three other character messages) is

```
(TCHAR) wParam
```

The same window procedure might be used with two window classes, one registered with *RegisterClassA* and the other registered with *RegisterClassW*. This means that the window procedure might get some messages with ANSI character codes and some messages with Unicode character codes. If your window procedure needs help to sort things out, it can call

```
fUnicode = IsWindowUnicode (hwnd) ;
```

The *fUnicode* variable will be TRUE if the window procedure for *hwnd* gets Unicode messages, which means the window is based on a window class that was registered with *RegisterClassW*.

## Message Ordering

Because the character messages are generated by the *TranslateMessage* function from WM\_KEYDOWN and WM\_SYSKEYDOWN messages, the character messages are delivered to your window procedure sandwiched between keystroke messages. For instance, if Caps Lock is not toggled on and you press and release the A key, the window procedure receives the following three messages:

| <i>Message</i> | <i>Key or Code</i>              |
|----------------|---------------------------------|
| WM_KEYDOWN     | Virtual key code for 'A' (0x41) |
| WM_CHAR        | Character code for 'a' (0x61)   |
| WM_KEYUP       | Virtual key code for 'A' (0x41) |

If you type an uppercase A by pressing the Shift key, pressing the A key, releasing the A key, and then releasing the Shift key, the window procedure receives five messages:

| <i>Message</i> | <i>Key or Code</i>               |
|----------------|----------------------------------|
| WM_KEYDOWN     | Virtual key code VK_SHIFT (0x10) |
| WM_KEYDOWN     | Virtual key code for 'A' (0x41)  |
| WM_CHAR        | Character code for 'A' (0x41)    |
| WM_KEYUP       | Virtual key code for 'A' (0x41)  |
| WM_KEYUP       | Virtual key code VK_SHIFT (0x10) |

The Shift key by itself does not generate a character message.

If you hold down the A key so that the typematic action generates keystrokes, you'll get a character message for each WM\_KEYDOWN message:

| <i>Message</i> | <i>Key or Code</i>              |
|----------------|---------------------------------|
| WM_KEYDOWN     | Virtual key code for 'A' (0x41) |
| WM_CHAR        | Character code for 'a' (0x61)   |
| WM_KEYDOWN     | Virtual key code for 'A' (0x41) |
| WM_CHAR        | Character code for 'a' (0x61)   |
| WM_KEYDOWN     | Virtual key code for 'A' (0x41) |
| WM_CHAR        | Character code for 'a' (0x61)   |
| WM_KEYDOWN     | Virtual key code for 'A' (0x41) |
| WM_CHAR        | Character code for 'a' (0x61)   |
| WM_KEYUP       | Virtual key code for 'A' (0x41) |

If some of the WM\_KEYDOWN messages have a Repeat Count greater than 1, the corresponding WM\_CHAR message will have the same Repeat Count.

The Ctrl Key in combination with a letter key generates ASCII control characters from 0x01 (Ctrl-A) through 0x1A (Ctrl-Z). Several of these control codes are also generated by the keys shown in the following table:

| <i>Key</i> | <i>Character Code</i> | <i>Duplicated by</i> | <i>ANSI C Escape</i> |
|------------|-----------------------|----------------------|----------------------|
| Backspace  | 0x08                  | Ctrl-H               | \b                   |
| Tab        | 0x09                  | Ctrl-I               | \t                   |
| Ctrl-Enter | 0x0A                  | Ctrl-J               | \n                   |
| Enter      | 0x0D                  | Ctrl-M               | \r                   |
| Esc        | 0x1B                  | Ctrl-[               |                      |

The rightmost column shows the escape code defined in ANSI C to represent the character codes for these keys.

Windows programs sometimes use the Ctrl key in combination with letter keys for menu accelerators (which I'll discuss in Chapter 10). In this case, the letter keys are not translated into character messages.

## Control Character Processing

The basic rule for processing keystroke and character messages is this: If you need to read keyboard character input in your window, you process the WM\_CHAR message. If you need to read the cursor keys, function keys, Delete, Insert, Shift, Ctrl, and Alt, you process the WM\_KEYDOWN message.

But what about the Tab key? Or Enter or Backspace or Escape? Traditionally, these keys generate ASCII control characters, as shown in the preceding table. But in Windows they also generate virtual key codes. Should these keys be processed during WM\_CHAR processing or WM\_KEYDOWN processing?

After a decade of considering this issue (and looking back over Windows code I've written over the years), I seem to prefer treating the Tab, Enter, Backspace, and Escape keys as control characters rather than as virtual keys. My WM\_CHAR processing often looks something like this:

```
case WM_CHAR:
    [other program lines]
    switch (wParam)
    {
    case '\b':           // backspace
        [other program line]
        break ;
```

```

case '\t':           // tab
    [other program lines]
    break ;

case '\n':           // linefeed
    [other program lines]
    break ;

case '\r':           // carriage return
    [other program lines]
    break ;

default:             // character codes
    [other program lines]
    break ;
}
return 0 ;

```

## Dead-Character Messages

Windows programs can usually ignore WM\_DEADCHAR and WM\_SYSDEADCHAR messages, but you should definitely know what dead characters are and how they work.

On some non-U.S. English keyboards, certain keys are defined to add a diacritic to a letter. These are called “dead keys” because they don’t generate characters by themselves. For instance, when a German keyboard is installed, the key that is in the same position as the +/- key on a U.S. keyboard is a dead key for the grave accent (`) when shifted and the acute accent (´) when unshifted.

When a user presses this dead key, your window procedure receives a WM\_DEADCHAR message with *wParam* equal to ASCII or Unicode code for the diacritic by itself. When the user then presses a letter key that can be written with this diacritic (for instance, the A key), the window procedure receives a WM\_CHAR message where *wParam* is the ANSI code for the letter ‘a’ with the diacritic.

Thus, your program does not have to process the WM\_DEADCHAR message because the WM\_CHAR message gives the program all the information it needs. The Windows logic even has built-in error handling: If the dead key is followed by a letter that can’t take a diacritic (such as ‘s’), the window procedure receives two WM\_CHAR messages in a row—the first with *wParam* equal to the ASCII code for the diacritic by itself (the same *wParam* value delivered with the WM\_DEADCHAR message) and the second with *wParam* equal to the ASCII code for the letter ‘s’.

Of course, the best way to get a feel for this is to see it in action. You need to load a foreign keyboard that uses dead keys, such as the German keyboard that I described earlier. You do this in the Control Panel by selecting Keyboard and then the Language tab. Then you need an application that shows you the details of every keyboard message a program can receive. That’s the KEYVIEW1 program coming up next.

## KEYBOARD MESSAGES AND CHARACTER SETS

The remaining sample programs in this chapter have flaws. They will not always run correctly under all versions of Windows. Their flaws are not something I deliberately introduced into the code; indeed, you might never notice them. These problems—I hesitate to call them “bugs”—reveal themselves only when switching among certain different keyboard languages and layouts, and when running the programs under Far Eastern versions of Windows that use multibyte character sets.

However, the programs *will* work much better when compiled for Unicode and run under Windows NT. This is the promise I made in Chapter 2, and it demonstrates why Unicode is so important in simplifying the work involved in internationalization.

### The KEYVIEW1 Program

The first step in understanding keyboard internationalization issues is to examine the contents of the keyboard and character messages that Windows delivers to your window procedure. The KEYVIEW1 program shown in Figure 6-3 will help. This program displays in its client area all the information that Windows sends the window procedure for the eight different keyboard messages.

#### KEYVIEW1.C

```
/*-----
KEYVIEW1.C -- Displays Keyboard and Character Messages
              (c) Charles Petzold, 1998
-----*/

#include <windows.h>

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

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("KeyView1") ;
    HWND          hwnd ;
    MSG           msg ;
    WNDCLASS      wndclass ;

    wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc    = WndProc ;
    wndclass.cbClsExtra     = 0 ;
    wndclass.cbWndExtra     = 0 ;
    wndclass.hInstance      = hInstance ;
```

Figure 6-3. The KEYVIEW1 program.

```

wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
wndclass.lpszMenuName    = NULL ;
wndclass.lpszClassName  = szAppName ;

if (!RegisterClass (&wndclass))
{
    MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                szAppName, MB_ICONERROR) ;
    return 0 ;
}

hwnd = CreateWindow (szAppName, TEXT ("Keyboard Message Viewer #1"),
                    WS_OVERLAPPEDWINDOW,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    NULL, NULL, hInstance, NULL) ;

ShowWindow (hwnd, iCmdShow) ;
UpdateWindow (hwnd) ;

while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
}
return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static int    cxClientMax, cyClientMax, cxClient, cyClient, cxChar, cyChar ;
    static int    cLinesMax, cLines ;
    static PMSG   pmsg ;
    static RECT   rectScroll ;
    static TCHAR  szTop[] = TEXT ("Message      Key      Char      ")
                          TEXT ("Repeat Scan Ext ALT Prev Tran") ;
    static TCHAR  szUnd[] = TEXT ("_____")
                          TEXT ("_____") ;

    static TCHAR * szFormat[2] = {
        TEXT ("% -13s %3d % -15s%c%6u %4d %3s %3s %4s %4s"),
        TEXT ("% -13s          0x%04X%1s%c %6u %4d %3s %3s %4s %4s") } ;

```

(continued)

Figure 6-3. *continued*

```

static TCHAR * szYes = TEXT ("Yes") ;
static TCHAR * szNo  = TEXT ("No") ;
static TCHAR * szDown = TEXT ("Down") ;
static TCHAR * szUp  = TEXT ("Up") ;

static TCHAR * szMessage [] = {
    TEXT ("WM_KEYDOWN"),    TEXT ("WM_KEYUP"),
    TEXT ("WM_CHAR"),       TEXT ("WM_DEADCHAR"),
    TEXT ("WM_SYSKEYDOWN"), TEXT ("WM_SYSKEYUP"),
    TEXT ("WM_SYSCHAR"),    TEXT ("WM_SYSDEADCHAR") } ;

HDC          hdc ;
int          i, iType ;
PAINTSTRUCT  ps ;
TCHAR        szBuffer[128], szKeyName [32] ;
TEXTMETRIC   tm ;

switch (message)
{
case WM_CREATE:
case WM_DISPLAYCHANGE:

    // Get maximum size of client area

    cxClientMax = GetSystemMetrics (SM_CXMAXIMIZED) ;
    cyClientMax = GetSystemMetrics (SM_CYMAXIMIZED) ;

    // Get character size for fixed-pitch font

    hdc = GetDC (hwnd) ;

    SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
    GetTextMetrics (hdc, &tm) ;
    cxChar = tm.tmAveCharWidth ;
    cyChar = tm.tmHeight ;

    ReleaseDC (hwnd, hdc) ;

    // Allocate memory for display lines

    if (pmsg)
        free (pmsg) ;

    cLinesMax = cyClientMax / cyChar ;
    pmsg = malloc (cLinesMax * sizeof (MSG)) ;
    cLines = 0 ;

    // fall through

```

```

case WM_SIZE:
    if (message == WM_SIZE)
    {
        cxClient = LOWORD (lParam) ;
        cyClient = HIWORD (lParam) ;
    }
    // Calculate scrolling rectangle

    rectScroll.left   = 0 ;
    rectScroll.right  = cxClient ;
    rectScroll.top    = cyChar ;
    rectScroll.bottom = cyChar * (cyClient / cyChar) ;

    InvalidateRect (hwnd, NULL, TRUE) ;
    return 0 ;

case WM_KEYDOWN:
case WM_KEYUP:
case WM_CHAR:
case WM_DEADCHAR:
case WM_SYSKEYDOWN:
case WM_SYSKEYUP:
case WM_SYSCHAR:
case WM_SYSDEADCHAR:

    // Rearrange storage array

    for (i = cLinesMax - 1 ; i > 0 ; i--)
    {
        pmsg[i] = pmsg[i - 1] ;
    }
    // Store new message

    pmsg[0].hwnd = hwnd ;
    pmsg[0].message = message ;
    pmsg[0].wParam = wParam ;
    pmsg[0].lParam = lParam ;

    cLines = min (cLines + 1, cLinesMax) ;

    // Scroll up the display

    ScrollWindow (hwnd, 0, -cyChar, &rectScroll, &rectScroll) ;

    break ;           // i.e., call DefWindowProc so Sys messages work

```

*(continued)*

Figure 6-3. *continued*

```

case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;

    SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
    SetBkMode (hdc, TRANSPARENT) ;
    TextOut (hdc, 0, 0, szTop, lstrlen (szTop)) ;
    TextOut (hdc, 0, 0, szUnd, lstrlen (szUnd)) ;

    for (i = 0 ; i < min (cLines, cyClient / cyChar - 1) ; i++)
    {
        iType = pmsg[i].message == WM_CHAR ||
                pmsg[i].message == WM_SYSCHAR ||
                pmsg[i].message == WM_DEADCHAR ||
                pmsg[i].message == WM_SYSDEADCHAR ;

        GetKeyNameText (pmsg[i].lParam, szKeyName,
                        sizeof (szKeyName) / sizeof (TCHAR)) ;

        TextOut (hdc, 0, (cyClient / cyChar - 1 - i) * cyChar, szBuffer,
                wsprintf (szBuffer, szFormat [iType],
                szMessage [pmsg[i].message - WM_KEYFIRST],
                pmsg[i].wParam,
                (PTSTR) (iType ? TEXT (" ") : szKeyName),
                (TCHAR) (iType ? pmsg[i].wParam : ' '),
                LOWORD (pmsg[i].lParam),
                HIWORD (pmsg[i].lParam) & 0xFF,
                0x01000000 & pmsg[i].lParam ? szYes : szNo,
                0x20000000 & pmsg[i].lParam ? szYes : szNo,
                0x40000000 & pmsg[i].lParam ? szDown : szUp,
                0x80000000 & pmsg[i].lParam ? szUp : szDown)) ;
    }
    EndPaint (hwnd, &ps) ;
    return 0 ;

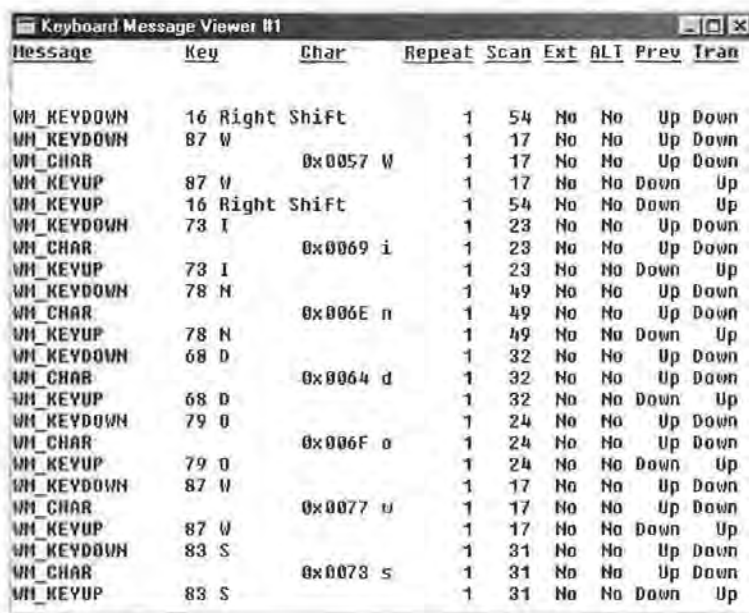
case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
}

```

KEYVIEW1 displays the contents of each keystroke and character message that it receives in its window procedure. It saves the messages in an array of MSG structures. The size of the array is based on the size of the maximized window size and the fixed-

pitch system font. If the user resizes the video display while the program is running (in which case KEYVIEW1 gets a WM\_DISPLAYCHANGE message), the array is reallocated. KEYVIEW1 uses the standard C *malloc* function to allocate memory for this array.

Figure 6-4 shows the KEYVIEW1 display after the word “Windows” has been typed. The first column shows the keyboard message. The second column shows the virtual key code for keystroke messages followed by the name of the key. This is obtained by using the *GetKeyNameText* function. The third column (labeled “Char”) shows the hexadecimal character code for character messages followed by the character itself. The remaining six columns display the status of the six fields in the *lParam* message parameter.



| <u>Message</u> | <u>Key</u>     | <u>Char</u> | <u>Repeat</u> | <u>Scan</u> | <u>Ext</u> | <u>Alt</u> | <u>Prev</u> | <u>Tran</u> |
|----------------|----------------|-------------|---------------|-------------|------------|------------|-------------|-------------|
| WM_KEYDOWN     | 16 Right Shift |             | 1             | 54          | No         | No         | Up          | Down        |
| WM_KEYDOWN     | 87 W           |             | 1             | 17          | No         | No         | Up          | Down        |
| WM_CHAR        |                | 0x0057 W    | 1             | 17          | No         | No         | Up          | Down        |
| WM_KEYUP       | 87 W           |             | 1             | 17          | No         | No         | Down        | Up          |
| WM_KEYUP       | 16 Right Shift |             | 1             | 54          | No         | No         | Down        | Up          |
| WM_KEYDOWN     | 73 I           |             | 1             | 23          | No         | No         | Up          | Down        |
| WM_CHAR        |                | 0x0069 i    | 1             | 23          | No         | No         | Up          | Down        |
| WM_KEYUP       | 73 I           |             | 1             | 23          | No         | No         | Down        | Up          |
| WM_KEYDOWN     | 78 N           |             | 1             | 49          | No         | No         | Up          | Down        |
| WM_CHAR        |                | 0x006E n    | 1             | 49          | No         | No         | Up          | Down        |
| WM_KEYUP       | 78 N           |             | 1             | 49          | No         | No         | Down        | Up          |
| WM_KEYDOWN     | 68 D           |             | 1             | 32          | No         | No         | Up          | Down        |
| WM_CHAR        |                | 0x0064 d    | 1             | 32          | No         | No         | Up          | Down        |
| WM_KEYUP       | 68 D           |             | 1             | 32          | No         | No         | Down        | Up          |
| WM_KEYDOWN     | 79 O           |             | 1             | 24          | No         | No         | Up          | Down        |
| WM_CHAR        |                | 0x006F o    | 1             | 24          | No         | No         | Up          | Down        |
| WM_KEYUP       | 79 O           |             | 1             | 24          | No         | No         | Down        | Up          |
| WM_KEYDOWN     | 87 U           |             | 1             | 17          | No         | No         | Up          | Down        |
| WM_CHAR        |                | 0x0077 u    | 1             | 17          | No         | No         | Up          | Down        |
| WM_KEYUP       | 87 U           |             | 1             | 17          | No         | No         | Down        | Up          |
| WM_KEYDOWN     | 83 S           |             | 1             | 31          | No         | No         | Up          | Down        |
| WM_CHAR        |                | 0x0073 s    | 1             | 31          | No         | No         | Up          | Down        |
| WM_KEYUP       | 83 S           |             | 1             | 31          | No         | No         | Down        | Up          |

Figure 6-4. The KEYVIEW1 display.

To ease the columnar display of this information, KEYVIEW1 uses a fixed-pitch font. As discussed in the last chapter, this requires calls to *GetStockObject* and *SelectObject*:

```
SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
```

KEYVIEW1 draws a header at the top of the client area identifying the nine columns. The text in this column is underlined. Although it's possible to create an underlined font, I took a different approach here. I defined two character string variables named *szTop* (which has the text) and *szUnd* (which has the underlining) and displayed both of them at the same position at the top of the window during the WM\_PAINT message. Normally, Windows displays text in an “opaque” mode, meaning that Windows erases the character background area while displaying a character. This would cause the second character string (*szUnd*) to erase the first (*szTop*). To prevent this, switch the device context into the “transparent” mode:

```
SetBkMode (hdc, TRANSPARENT) ;
```

This method of underlining is possible only when using a fixed-pitch font. Otherwise, the underline character wouldn't necessarily be the same width as the character the underline is to appear under.

## The Foreign-Language Keyboard Problem

If you're running the American English version of Windows, you can install different keyboard layouts and pretend that you're typing in a foreign language. You install foreign language keyboard layouts in the Keyboard applet in the Control Panel. Select the Language tab, and click Add. To see how dead keys work, you might want to install the German keyboard. I'll also be discussing the Russian and Greek keyboard layouts, so you might want to install those as well. If the Russian and Greek keyboard layouts are not available in the list that the Keyboard applet displays, you might need to install multilanguage support. Select the Add/Remove Programs applet from the Control Panel, and choose the Windows Setup tab. Make sure the Multilanguage Support box is checked. In any case, you'll need to have your original Windows CD-ROM handy for these changes.

After you install other keyboard layouts, you'll see a blue box with a two-letter code in the tray at the right side of the task bar. It'll be "EN" if the default is English. When you click on this icon, you get a list of all the installed keyboard layouts. You can change the keyboard for the currently active program by clicking on the one you want. This change affects only the currently active program.

Now we're ready to experiment. Compile the KEYVIEW1 program *without* the UNICODE identifier defined. (On this book's companion disc, the non-Unicode version of KEYVIEW1 is located in the RELEASE subdirectory.) Run the program under the American English version of Windows, and type the letters "abcde." The WM\_CHAR messages are exactly what you expect: the ASCII character codes 0x61, 0x62, 0x63, 0x64, and 0x65 and the characters a, b, c, d, and e.

Now, while still running KEYVIEW1, select the German keyboard layout. Press the = key and then a vowel (a, e, i, o, or u). The = key generates a WM\_DEADCHAR message, and the vowel generates a WM\_CHAR message with (respectively) the character codes 0xE1, 0xE9, 0xED, 0xF3, 0xFA, and the characters á, é, í, ó, and ú. This is how dead keys work.

Now select the Greek keyboard layout. Type "abcde" and what do you get? You get WM\_CHAR messages with the character codes 0xE1, 0xE2, 0xF8, 0xE4, 0xE5, and the characters á, â, ø, ä, and å. Something doesn't seem to be right here. Shouldn't you be getting letters in the Greek alphabet?

Now switch to the Russian keyboard and again type "abcde." Now you get WM\_CHAR messages with the character codes 0xF4, 0xE8, 0xF1, 0xE2, and 0xF3, and the characters ô, è, ñ, â, and ó. Again, something is wrong. You should be getting letters in the Cyrillic alphabet.

The problem is this: you have switched the keyboard to generate different character codes, but you haven't informed GDI of this switch so that GDI can interpret these character codes by displaying the proper symbols.

If you're very brave, and you have a spare PC to play with, and if you have a Professional or Universal Subscription to Microsoft Developer Network (MSDN), you might want to install (for example) the Greek version of Windows. You can also install the same four keyboard layouts (English, Greek, German, and Russian). Now run KEYLOOK1. Switch to the English keyboard layout, and type "abcde". You get the ASCII character codes 0x61, 0x62, 0x63, 0x64, and 0x65 and the characters a, b, c, d, and e. (And you can breathe a sigh of relief that ASCII still works, even in Greece.)

Under this Greek version of Windows, switch to the Greek keyboard layout and type "abcde." You get WM\_CHAR messages with the character codes 0xE1, 0xE2, 0xF8, 0xE4, and 0xE5. These are the same character codes you got under the English version of Windows with the Greek keyboard layout installed. But now the displayed characters are α, β, ψ, δ, and ε. These are indeed the lowercase Greek letters alpha, beta, psi, delta, and epsilon. (What happened to gamma? Well, if you were using the Greek version of Windows for real, you'd probably be using a keyboard with Greek letters on the keycaps. The key corresponding to the English c happens to be a psi. The gamma is generated by the key corresponding to the English g. You can see the complete Greek keyboard layout on page 587 of Nadine Kano's *Developing International Software for Windows 95 and Windows NT*.

Still running KEYVIEW1 under the Greek version of Windows, switch to the German keyboard layout. Type the = key followed by a, then e, then i, then o, and then u. You get WM\_CHAR messages with the character codes 0xE1, 0xE9, 0xED, 0xF3, and 0xFA. These are the same character codes as under the English version of Windows with the German keyboard installed. However, the displayed characters are α, ι, ν, σ, and ï, not the correct ā, é, î, ó, and ú.

Now switch to the Russian keyboard and type "abcde." You get the character codes 0xF4, 0xE8, 0xF1, 0xE2, and 0xF3, which are the same as under the English version of Windows with the Russian keyboard installed. However, the displayed characters are τ, θ, ρ, β, and σ, not letters in the Cyrillic alphabet.

You can also install the Russian version of Windows. As you may have guessed by now, the English and Russian keyboard layouts will work, but not the German or Greek.

Now, if you're really, really brave, you can install the Japanese version of Windows and run KEYVIEW1. If you type at your American keyboard, you can enter English text and everything will seem to work fine. However, if you switch to the German, Greek, or Russian keyboard layouts and try any of the exercises described above, you'll see the characters displayed as dots. If you type capital letters—either accented German letters, Greek letters, or Russian letters—you'll see the characters rendered as katakana, which is the Japanese alphabet generally used to spell words from other languages. You may have fun typing katakana, but it's not German, Greek, or Russian.

The Far East versions of Windows include a utility called the Input Method Editor (IME) that appears as a floating toolbar. This utility lets you use the normal keyboard for

entering ideographs, which are the complex characters used in Chinese, Japanese, and Korean. Basically, you type combinations of letters and the composed symbols appear in another floating window. You then press Enter and the resultant character codes are sent to the active window (that is, KEYVIEW1). KEYVIEW1 responds with almost total nonsense—the WM\_CHAR messages have character codes above 128, but the characters are meaningless. (Nadine Kano's book has much more information on using the IME.)

So, we've seen a couple examples of KEYLOOK1 displaying incorrect characters—when running the English version of Windows with the Russian or Greek keyboard layouts installed, when running the Greek version of Windows with the Russian or German keyboard layouts installed, and when running the Russian version of Windows with the German, Russian, or Greek keyboards installed. We've also seen errors when entering characters from the Input Method Editor in the Japanese version of Windows.

## Character Sets and Fonts

The problem with KEYLOOK1 is a font problem. The font that it's using to display characters on the screen is inconsistent with the character codes it's receiving from the keyboard. So, let's take a look at some fonts.

As I'll discuss in more detail in Chapter 17, Windows supports three types of fonts—bitmap fonts, vector fonts, and (beginning in Windows 3.1) TrueType fonts.

The vector fonts are virtually obsolete. The characters in these fonts were composed of simple lines, but these lines did not define filled areas. The vector fonts had the benefit of being scalable to any size, but the characters often looked anemic.

TrueType fonts are outline fonts with characters defined by filled areas. TrueType fonts are scalable; indeed the character definitions contain "hints" for avoiding rounding problems that could result in unsightly or unreadable text. It is with TrueType that Windows achieves a true WYSIWYG ("what you see is what you get") display of text on the video display that accurately matches printer output.

In bitmap fonts, each character is defined by an array of bits that correspond to the pixels of the video display. Bitmaps fonts can be scalable to larger sizes, but they look jagged as a result. Bitmap fonts are often tweaked by their designers to be more easily readable on the video display. Thus, Windows uses bitmap fonts for the text that appears in title bars, menus, buttons, and dialog boxes.

The bitmap font that you get in a default device context is known as the system font. You can obtain a handle to this font by calling the *GetStockObject* function with the identifier SYSTEM\_FONT. The KEYVIEW1 program elects to use a fixed-pitch version of the system font, denoted by SYSTEM\_FIXED\_FONT. Another alternative in the *GetStockObject* function is OEM\_FIXED\_FONT.

These three fonts have typeface names of (respectively) System, FixedSys, and Terminal. A program can use the typeface name to refer to the font in a *CreateFont* or *Create-*

*FontIndirect* function call. These three fonts are stored in two sets of three files in the FONTS subdirectory of the Windows directory. The particular set of files that Windows uses depends on whether you've elected to display "Small Fonts" or "Large Fonts" in the Display applet of the Control Panel (that is, whether you want Windows to assume that the video display has a 96 dpi resolution or a 120 dpi resolution). This is all summarized in the following table:

| <b><i>GetStockObject<br/>Identifier</i></b> | <b><i>Typeface<br/>Name</i></b> | <b><i>Small Font<br/>File</i></b> | <b><i>Large Font<br/>File</i></b> |
|---------------------------------------------|---------------------------------|-----------------------------------|-----------------------------------|
| SYSTEM_FONT                                 | System                          | VGASYS.FON                        | 8514SYS.FON                       |
| SYSTEM_FIXED_FONT                           | FixedSys                        | VGAFIX.FON                        | 8514FIX.FON                       |
| OEM_FIXED_FONT                              | Terminal                        | VGAOEM.FON                        | 8514OEM.FON                       |

In the file names, "VGA" refers to the Video Graphics Array, the video adapter that IBM introduced in 1987. It was IBM's first PC video adapter to have a pixel display size of 640 by 480. If you select Small Fonts from the Display applet in the Control Panel (meaning that you want Windows to assume that the video display has a resolution of 96 dpi), Windows uses the filenames beginning with "VGA" for these three fonts. If you select Large Fonts (meaning that you want a resolution of 120 dpi), Windows uses the filenames beginning with "8514." The 8514 was another video adapter that IBM introduced in 1987, and it had a maximum display size of 1024 by 768.

Windows does not want you to see these files. The files have the system and hidden file attributes set, and if you use the Windows Explorer to view the contents of your FONTS subdirectory, you won't see them at all, even if you've elected to view system and hidden files. Use the Find option from the Tools menu to search for files with a specification of \*.FON. From there, you can double-click the filename to see what the font characters look like.

For many standard controls and user interface items, Windows doesn't use the System font. Instead, it uses a font with the typeface name MS Sans Serif. (MS stands for Microsoft.) This is also a bitmap font. The file (named SSERIFE.FON) contains fonts based on a 96-dpi video display, with point sizes of 8, 10, 12, 14, 18, and 24. You can get this font by using the DEFAULT\_GUI\_FONT identifier in *GetStockObject*. The point size Windows uses will be based on the display resolution you've selected in the Display applet of the Control Panel.

So far, I've mentioned four of the identifiers you can use with *GetStockObject* to obtain a font for use in a device context. There are three others: ANSI\_FIXED\_FONT, ANSI\_VAR\_FONT, and DEVICE\_DEFAULT\_FONT. To begin approaching the problem of the keyboard and character displays, let's take a look at all the stock fonts in Windows. The program that displays the fonts is named STOKFONT and is shown in Figure 6-5 beginning on the following page.

**STOKFONT.C**

```

/*-----
STOKFONT.C -- Stock Font Objects
              (c) Charles Petzold, 1998
-----*/

#include <windows.h>

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

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("StokFont") ;
    HWND          hwnd ;
    MSG           msg ;
    WNDCLASS      wndclass ;

    wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc    = WndProc ;
    wndclass.cbClsExtra     = 0 ;
    wndclass.cbWndExtra     = 0 ;
    wndclass.hInstance      = hInstance ;
    wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName   = NULL ;
    wndclass.lpszClassName  = szAppName ;

    if (!RegisterClass (&wndclass))
    {
        MessageBox (NULL, TEXT ("Program requires Windows NT!"),
                    szAppName, MB_ICONERROR) ;
        return 0 ;
    }

    hwnd = CreateWindow (szAppName, TEXT ("Stock Fonts"),
                        WS_OVERLAPPEDWINDOW | WS_VSCROLL,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

```

**Figure 6-5.** *The STOKFONT program.*

```

while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
}
return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static struct
    {
        int      idStockFont ;
        TCHAR * szStockFont ;
    }
    stockfont [] = { OEM_FIXED_FONT,      "OEM_FIXED_FONT",
                    ANSI_FIXED_FONT,     "ANSI_FIXED_FONT",
                    ANSI_VAR_FONT,      "ANSI_VAR_FONT",
                    SYSTEM_FONT,        "SYSTEM_FONT",
                    DEVICE_DEFAULT_FONT, "DEVICE_DEFAULT_FONT",
                    SYSTEM_FIXED_FONT,   "SYSTEM_FIXED_FONT",
                    DEFAULT_GUI_FONT,    "DEFAULT_GUI_FONT" } ;

    static int  iFont, cFonts = sizeof stockfont / sizeof stockfont[0] ;
    HDC         hdc ;
    int         i, x, y, cxGrid, cyGrid ;
    PAINTSTRUCT ps ;
    TCHAR       szFaceName [LF_FACESIZE], szBuffer [LF_FACESIZE + 64] ;
    TEXTMETRIC  tm ;

    switch (message)
    {
    case WM_CREATE:
        SetScrollRange (hwnd, SB_VERT, 0, cFonts - 1, TRUE) ;
        return 0 ;

    case WM_DISPLAYCHANGE:
        InvalidateRect (hwnd, NULL, TRUE) ;
        return 0 ;

    case WM_VSCROLL:
        switch (LOWORD (wParam))
        {
        case SB_TOP:          iFont = 0 ;          break ;
        case SB_BOTTOM:       iFont = cFonts - 1 ; break ;
        case SB_LINEUP:

```

(continued)

Figure 6-5. *continued*

```

        case SB_PAGEUP:          iFont -= 1 ;          break ;
        case SB_LINEDOWN:
        case SB_PAGEDOWN:       iFont += 1 ;          break ;
        case SB_THUMBPOSITION:  iFont = HIWORD (wParam) ; break ;
    }
    iFont = max (0, min (cFonts - 1, iFont)) ;
    SetScrollPos (hwnd, SB_VERT, iFont, TRUE) ;
    InvalidateRect (hwnd, NULL, TRUE) ;
    return 0 ;

case WM_KEYDOWN:
    switch (wParam)
    {
        case VK_HOME: SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0) ;      break ;
        case VK_END:   SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0) ;   break ;
        case VK_PRIOR:
        case VK_LEFT:
        case VK_UP:     SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0) ;   break ;
        case VK_NEXT:
        case VK_RIGHT:
        case VK_DOWN:   SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN, 0) ; break ;
    }
    return 0 ;

case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;

    SelectObject (hdc, GetStockObject (stockfont[iFont].idStockFont)) ;
    GetTextFace (hdc, LF_FACESIZE, szFaceName) ;
    GetTextMetrics (hdc, &tm) ;
    cxGrid = max (3 * tm.tmAveCharWidth, 2 * tm.tmMaxCharWidth) ;
    cyGrid = tm.tmHeight + 3 ;

    TextOut (hdc, 0, 0, szBuffer,
        wsprintf (szBuffer, TEXT (" %s: Face Name = %s, CharSet = %i"),
            stockfont[iFont].szStockFont,
            szFaceName, tm.tmCharSet)) ;

    SetTextAlign (hdc, TA_TOP | TA_CENTER) ;

    // vertical and horizontal lines

```

```

for (i = 0 ; i < 17 ; i++)
{
    MoveToEx (hdc, (i + 2) * cxGrid, 2 * cyGrid, NULL) ;
    LineTo   (hdc, (i + 2) * cxGrid, 19 * cyGrid) ;

    MoveToEx (hdc,      cxGrid, (i + 3) * cyGrid, NULL) ;
    LineTo   (hdc, 18 * cxGrid, (i + 3) * cyGrid) ;
}

// vertical and horizontal headings

for (i = 0 ; i < 16 ; i++)
{
    TextOut (hdc, (2 * i + 5) * cxGrid / 2, 2 * cyGrid + 2, szBuffer,
        wsprintf (szBuffer, TEXT ("%X-"), i)) ;

    TextOut (hdc, 3 * cxGrid / 2, (i + 3) * cyGrid + 2, szBuffer,
        wsprintf (szBuffer, TEXT ("-%X"), i)) ;
}

// characters

for (y = 0 ; y < 16 ; y++)
for (x = 0 ; x < 16 ; x++)
{
    TextOut (hdc, (2 * x + 5) * cxGrid / 2,
        (y + 3) * cyGrid + 2, szBuffer,
        wsprintf (szBuffer, TEXT ("%c"), 16 * x + y)) ;
}

EndPaint (hwnd, &ps) ;
return 0 ;

case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
}

return DefWindowProc (hwnd, message, wParam, lParam) ;
}

```

This program is fairly simple. It uses the scroll bar and cursor movement keys to let you select one of the seven stock fonts to display. The program displays the 256 characters of the font in a grid. The headings at the top and left of the grid show the hexadecimal values of the character codes.

At the top of the client area, STOKFONT shows the identifier it uses to select the font using the *GetStockObject* function. It also displays the typeface name of the font obtained from the *GetTextFace* function and the *tmCharSet* field of the TEXTMETRIC structure. This “character set identifier” turns out to be crucial in understanding how Windows deals with foreign-language versions of Windows.

If you run STOKFONT under the American English version of Windows, the first screen you’ll see shows you the font obtained by using the OEM\_FIXED\_FONT identifier with the *GetStockObject* function. This is shown in Figure 6-6.

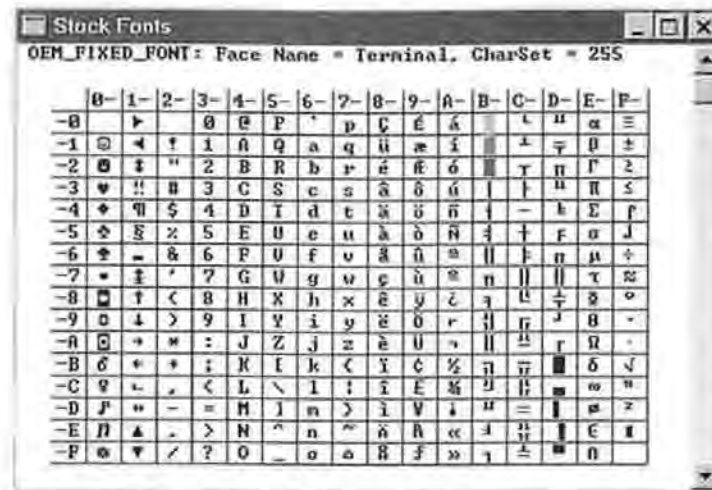


**Figure 6-6.** The OEM\_FIXED\_FONT in the U.S. version of Windows.

In this character set (as in all the others in this chapter), you’ll see some ASCII. But remember that ASCII is a 7-bit code that defines displayable characters for codes 0x20 through 0x7E. By the time IBM developed the original IBM PC the 8-bit byte had been firmly established, so a full 8 bits could be used for character codes. IBM decided to extend the ASCII character set with a bunch of line- and block-drawing characters, accented letters, Greek letters, math symbols, and some miscellany. Many character-mode MS-DOS programs used the line-drawing characters in their on-screen displays, and many MS-DOS programs used some of the extended characters in their files.

This particular character set posed a problem for the original developers of Windows. On the one hand, the line- and block-drawing characters are not needed in Windows because Windows has a complete graphics programming language. The 48 codes used for these characters could better be used for additional accented letters required by many Western European languages. On the other hand, the IBM character set was definitely a standard that couldn’t be ignored completely.

So, the original developers of Windows decided to support the IBM character set but to relegate it to secondary importance—mostly for old MS-DOS applications that ran in a

window and for Windows programs that needed to use files created by MS-DOS applications. Windows applications do not use the IBM character set, and over the years it has faded in importance. Still, however, if you need it you can use it. In this context, “OEM” means “IBM.”

(Be aware that foreign-language versions of Windows do not necessarily support the same OEM character set as the American English version does. Other countries had their own MS-DOS character sets. That’s a whole subject in itself, but not one for this book.)

Because the IBM character set was deemed inappropriate for Windows, a different extended character set was selected. This is called the “ANSI character set,” referring to the American National Standards Institute, but it’s actually an ISO (International Standards Organization) standard, namely standard 8859. It’s also known as Latin 1, Western European, or code page 1252. Figure 6-7 shows one version of the ANSI character set—the system font in the American English version of Windows.

|    | 0- | 1- | 2- | 3- | 4- | 5- | 6- | 7- | 8-  | 9- | A- | B- | C- | D- | E- | F- |
|----|----|----|----|----|----|----|----|----|-----|----|----|----|----|----|----|----|
| -0 |    |    |    | 0  | @  | P  | ^  | p  |     |    |    | *  | À  | Ð  | à  | ð  |
| -1 |    |    | !  | 1  | A  | Q  | a  | q  |     | '  | i  | ±  | Á  | Ñ  | á  | ñ  |
| -2 |    |    | "  | 2  | B  | R  | b  | r  | ,   | '  | ¢  | ²  | Â  | Ô  | â  | ô  |
| -3 |    |    | #  | 3  | C  | S  | c  | s  | f   | "  | £  | ³  | Ã  | Õ  | ã  | ó  |
| -4 |    |    | \$ | 4  | D  | T  | d  | t  | "   | "  | ¤  | ⁴  | Ä  | Ö  | ä  | ö  |
| -5 |    |    | %  | 5  | E  | U  | e  | u  | ... | •  | ¥  | µ  | Å  | Ø  | å  | ø  |
| -6 |    |    | &  | 6  | F  | V  | f  | v  | †   | -  |    | ¶  | Æ  | Ö  | æ  | ö  |
| -7 |    |    | '  | 7  | G  | W  | g  | w  | ‡   | —  | §  | ·  | Ç  | ×  | ç  | ÷  |
| -8 |    |    | (  | 8  | H  | X  | h  | x  | -   | -  | -  | ¸  | È  | Ø  | è  | ø  |
| -9 |    |    | )  | 9  | I  | Y  | i  | y  | ‰   | ™  | ©  | ¹  | É  | Ù  | é  | ù  |
| -A |    |    | *  | :  | J  | Z  | j  | z  | Š   | š  | ª  | º  | Ê  | Ú  | ê  | ú  |
| -B |    |    | +  | ;  | K  | [  | k  | {  | <   | >  | «  | »  | Ë  | Û  | ë  | û  |
| -C |    |    | ,  | <  | L  | \  |    |    | œ   | œ  | ¬  | ¼  | Ì  | Ü  | ì  | ü  |
| -D |    |    | -  | =  | M  | ]  | m  | }  |     |    | -  | ½  | Í  | Ý  | í  | ý  |
| -E |    |    | .  | >  | N  | ^  | n  | ~  |     |    | @  | ¾  | Î  | Þ  | î  | þ  |
| -F |    |    | /  | ?  | O  | _  | o  |    |     | ÿ  | -  | ¿  | Ï  | ß  | ï  | ÿ  |

**Figure 6-7.** The `SYSTEM_FONT` in the U.S. version of Windows.

The thick vertical bars indicate codes for which characters are not defined. Notice that codes 0x20 through 0x7E are once again ASCII. Also, the ASCII control characters (0x00 through 0x1F, and 0x7F) are not associated with displayable characters. This is as it should be.

The codes 0xC0 through 0xFF make the ANSI character set important to foreign-language versions of Windows. These codes provide 64 characters commonly found in Western European languages. The character 0xA0, which looks like a space, is actually defined as a nonbreaking space, such as the space in “WW II.”

I say this is “one version” of the ANSI character set because of the presence of the characters for codes 0x80 through 0x9F. The fixed-pitch system font includes only two of these characters, as shown in Figure 6-8.

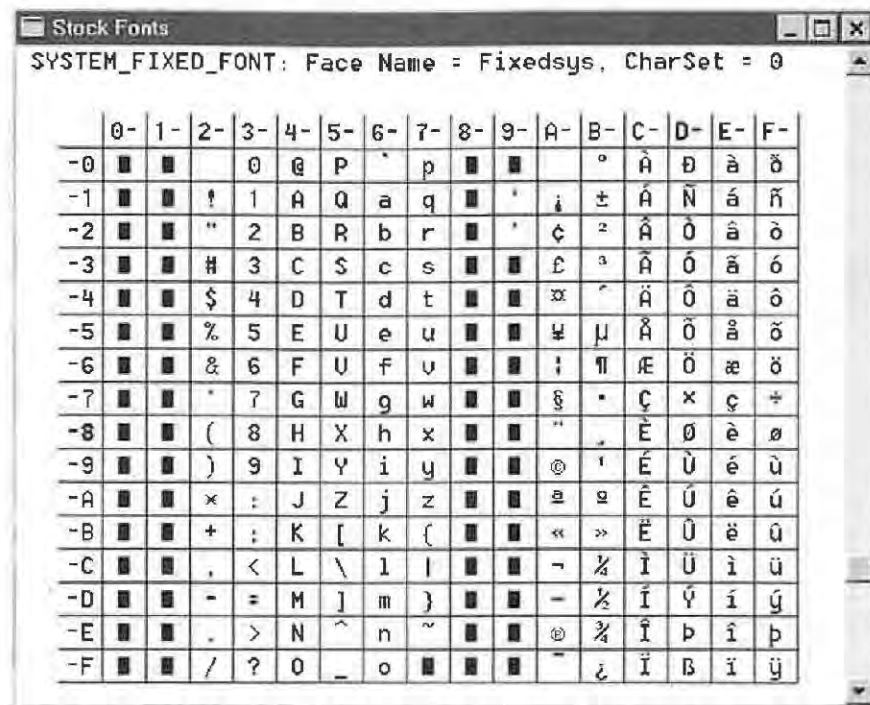


**Figure 6-8.** The `SYSTEM_FIXED_FONT` in the U.S. version of Windows.

In Unicode, codes 0x0000 through 0x007F are the same as ASCII, codes 0x0080 through 0x009F duplicate control characters 0x0000 through 0x001F, and codes 0x00A0 through 0x00FF are the same as the ANSI character set used in Windows.

If you run the German version of Windows, you'll get the same ANSI character sets when you call `GetStockObject` with the `SYSTEM_FONT` or `SYSTEM_FIXED_FONT` identifiers. This is true of other Western European versions of Windows as well. The ANSI character set was designed to have all the characters that are required in these languages.

However, when you run the Greek version of Windows, the default character set is *not* the same. Instead, the `SYSTEM_FONT` is that shown in Figure 6-9.

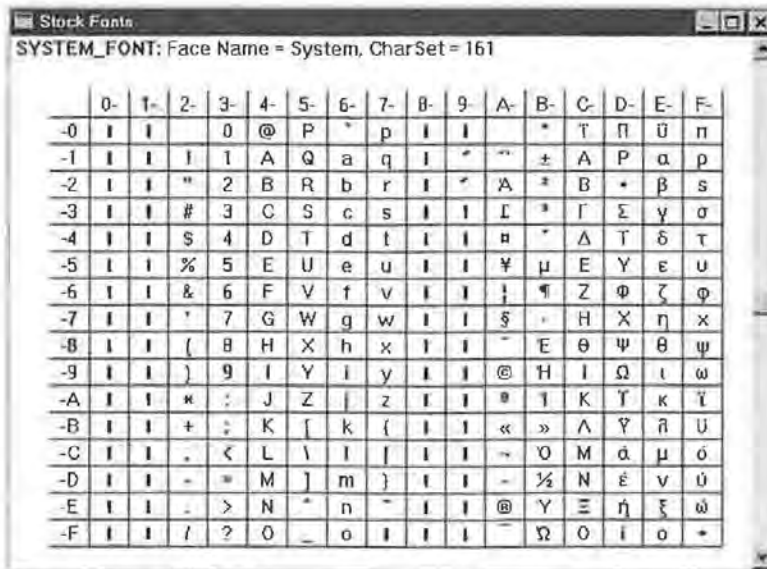


**Figure 6-9.** The `SYSTEM_FONT` in the Greek version of Windows.

The `SYSTEM_FIXED_FONT` has the same characters. Notice the codes from 0xC0 through 0xFF. These codes contain uppercase and lowercase letters from the Greek alphabet. When you're running the Russian version of Windows, the default character set is shown in Figure 6-10.

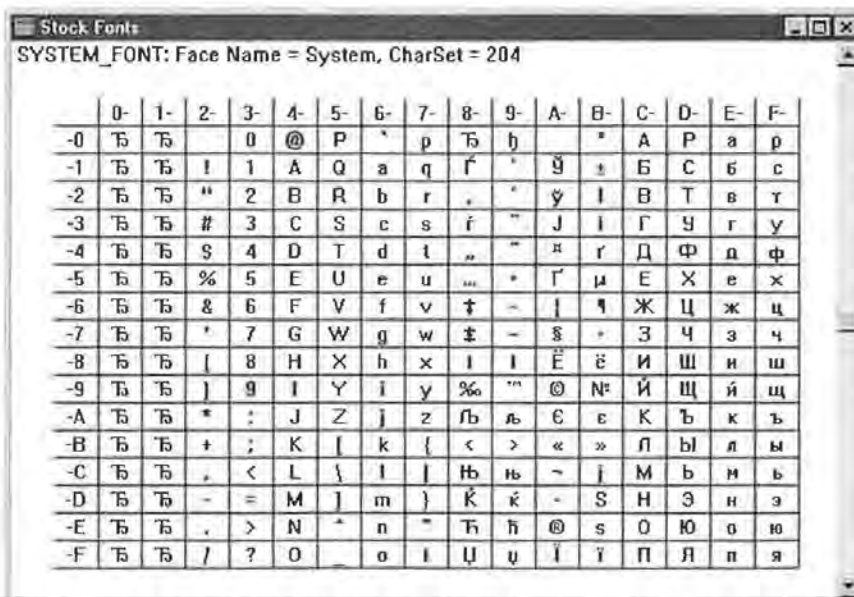


**Figure 6-10.** The `SYSTEM_FONT` in the Russian version of Windows.

Again, notice that uppercase and lowercase letters of the Cyrillic alphabet occupy codes 0xC0 and 0xFF.

Figure 6-11 on the following page shows the `SYSTEM_FONT` from the Japanese version of Windows. The characters from 0xA5 through 0xDF are all part of the katakana alphabet.

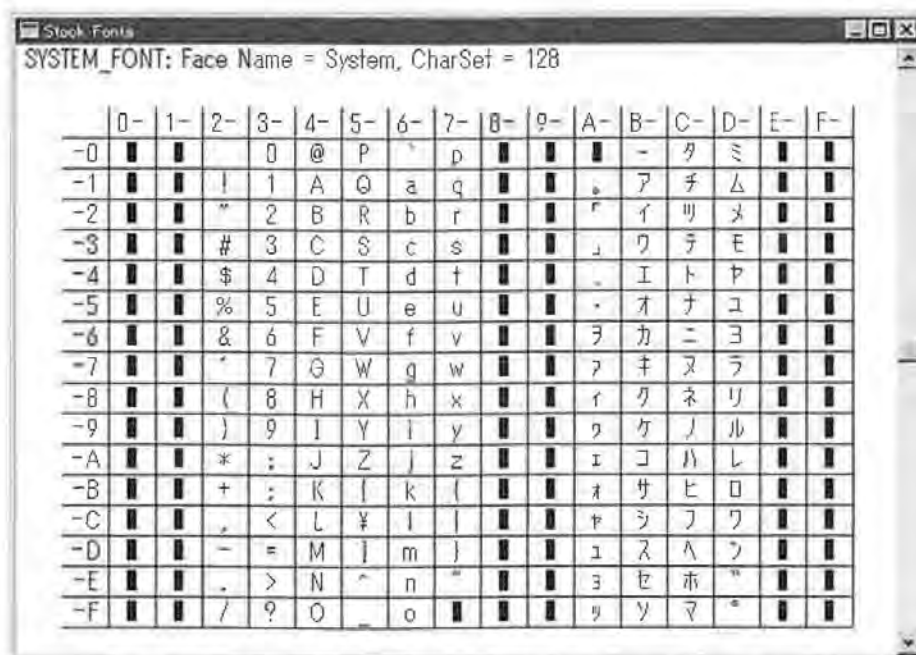


**Figure 6-11.** The `SYSTEM_FONT` in the Japanese version of Windows.

The Japanese system font shown in Figure 6-11 is different from those shown previously because it is actually a double-byte character set (DBCS) called Shift-JIS. (JIS stands for Japanese Industrial Standard.) Most of the character codes from 0x81 through 0x9F and from 0xE0 through 0xFF are really just the first byte of a 2-byte code. The second byte is usually in the range 0x40 through 0xFC. (See Appendix G in Nadine Kano's book for a complete table of these codes.)

So now we can see where the problem is in `KEYVIEW1`: If you have the Greek keyboard layout installed and you type "abcde," *regardless of the version of Windows you're running*, Windows generates `WM_CHAR` messages with the character codes 0xE1, 0xE2, 0xF8, 0xE4, and 0xE5. But these character codes will correspond to the characters  $\alpha$ ,  $\beta$ ,  $\psi$ ,  $\delta$ , and  $\epsilon$  only if you're running the Greek version of Windows with the Greek system font.

If you have the Russian keyboard layout installed and you type "abcde," *regardless of the version of Windows you're running*, Windows generates `WM_CHAR` messages with the character codes 0xF4, 0xE8, 0xF1, 0xE2, and 0xF3. But these character codes will correspond to the characters  $\phi$ ,  $\pi$ ,  $\varsigma$ ,  $\nu$ , and  $\gamma$  only if you're running the Russian version of Windows or another language that uses the Cyrillic alphabet, and you're using the Cyrillic system font.

If you have the German keyboard layout installed and you type the = key (or the key in that same position) followed by the a, e, i, o, or u key, *regardless of the version of Windows you're running*, Windows generates `WM_CHAR` messages with the character codes 0xE1, 0xE9, 0xED, 0xF3, and 0xFA. Only if you're running a Western European or American version of Windows, which means that you have the Western European system font, will these character codes correspond to the characters á, é, í, ó, or ú.

If you have the American English keyboard layout installed, you can type anything on your keyboard and Windows will generate WM\_CHAR messages with character codes that correctly match to the proper characters.

## What About Unicode?

I claimed in Chapter 2 that Unicode support in Windows NT helps out in writing programs for an international market. Let's try compiling KEYVIEW1 with the UNICODE identifier defined and running it under various versions of Windows NT. (On this book's companion disc, the Unicode version of KEYVIEW1 is located in the DEBUG directory.)

If the UNICODE identifier is defined when the program is compiled, the "KeyView1" window class is registered with the *RegisterClassW* rather than the *RegisterClassA* function. This means that any message delivered to *WndProc* that has character or text data will use 16-bit characters rather than 8-bit characters. In particular, the WM\_CHAR message will deliver a 16-bit character code rather than an 8-bit character code.

Run the Unicode version of KEYVIEW1 under the American English version of Windows NT. I'll assume you've installed at least the other three keyboard layouts we've been experimenting with—that is, German, Greek, and Russian.

With the American English version of Windows NT and either the English or German keyboard layout installed, the Unicode version of KEYVIEW1 will appear to work the same as the non-Unicode version. It will receive the same character codes (all of which will be 0xFF or lower in value) and display the same correct characters. This is because the first 256 characters of Unicode are the same as the ANSI character set used in Windows.

Now switch to the Greek keyboard layout, and type "abcde." The WM\_CHAR messages will have the Unicode character codes 0x03B1, 0x03B2, 0x03C8, 0x03B4, and 0x03B5. Note that for the first time we're seeing character codes with values higher than 0xFF. These Unicode character codes correspond to the Greek letters α, β, ψ, δ, and ε. However, all five characters are displayed as solid blocks! This is because the SYSTEM\_FIXED\_FONT only has 256 characters.

Now switch to the Russian keyboard layout, and type "abcde." KEYVIEW1 displays WM\_CHAR messages with the Unicode character codes 0x0444, 0x0438, 0x0441, 0x0432, and 0x0443, corresponding to the Cyrillic characters ф, и, с, в, and y. Once again, however, all five characters are displayed as solid blocks.

In short, where the non-Unicode version of KEYVIEW1 displayed incorrect characters, the Unicode version of KEYVIEW1 displays solid blocks, indicating that the current font does not have that particular character. I hesitate to say that the Unicode version of KEYVIEW1 represents an "improvement" over the non-Unicode version, but it does. The non-Unicode version displays characters that are not correct. The Unicode version does not.

The differences between the Unicode and non-Unicode versions of KEYVIEW1 are mostly in two areas.

First, the WM\_CHAR message is accompanied by a 16-bit character code rather than an 8-bit character code. The 8-bit character code in the non-Unicode version of KEYVIEW1

could have different meanings depending what keyboard layout is active. A code of 0xE1 could mean á if it came from the German keyboard, α if it came from the Greek keyboard, and ъ if it came from the Russian keyboard. In the Unicode version of the program, the 16-bit character code is totally unambiguous. The á character is 0x00E1, the α character is 0x03B1, and the ъ character is 0x0431.

Second, the Unicode *TextOutW* function displays characters based on 16-bit character codes rather than on the 8-bit character codes of the non-Unicode *TextOutA* function. Because these 16-bit character codes are totally unambiguous, GDI can determine whether the font currently selected in the device context is capable of displaying each character.

Running the Unicode version of KEYVIEW1 under the American version of Windows NT is somewhat deceptive, because it appears as if GDI is simply displaying character codes in the range 0x0000 through 0x00FF and not those above 0x00FF. That is, it appears as if there's a simple one-to-one mapping between the character codes and the 256 characters of the system font.

However, if you install the Greek or Russian versions of Windows NT, you'll discover that this is not the case. For example, if you install the Greek version of Windows NT, the American English, German, Greek, and Russian keyboards will generate the same Unicode character codes as the American version of Windows NT. However, the Greek version of Windows NT will not display German-accented characters or Russian characters because these characters are not in the Greek system font. Similarly, the Russian version of Windows NT will not display the German-accented characters or Greek characters because these characters are not in the Russian system font.

Where the Unicode version of KEYVIEW1 makes the most dramatic difference is under the Japanese version of Windows NT. You enter Japanese characters from the IME and they display correctly. The only problem is formatting: because the Japanese characters are often visually complex, they are displayed twice as wide as other characters.

## TrueType and Big Fonts

The bitmap fonts that we've been using (with the exception of the fonts in the Japanese version of Windows) contain a maximum of 256 characters. This is to be expected, because the format of the bitmap font file goes back to the early days of Windows when character codes were assumed to be mere 8-bit values. That's why when we use the `SYSTEM_FONT` or the `SYSTEM_FIXED_FONT`, there are always some characters from some languages that we can't display properly. (The Japanese system font is a bit different because it's a double-byte character set; most of the characters are actually stored in TrueType Collection files with a filename extension of .TCC.)

TrueType fonts can contain more than 256 characters. Not all TrueType fonts have more than 256 characters, but the ones shipped with Windows 98 and Windows NT do. Or rather, they do if you've installed multilanguage support. In the Add/Remove Programs applet of the Control Panel, click the Windows Setup tab and make sure Multilanguage

Support is checked. This multilanguage support involves five character sets: Baltic, Central European, Cyrillic, Greek, and Turkish. The Baltic character set is used for Estonian, Latvian, and Lithuanian. The Central European character set is used for Albanian, Czech, Croatian, Hungarian, Polish, Romanian, Slovak, and Slovenian. The Cyrillic character set is used for Bulgarian, Belarusian, Russian, Serbian, and Ukrainian.

The TrueType fonts shipped with Windows 98 support those five character sets, plus the Western European (ANSI) character set that is used for virtually all other languages except those in the Far East (Chinese, Japanese, and Korean). TrueType fonts that support multiple character sets are sometimes referred to as “big fonts.” The word “big” in this context does not refer to the size of the characters, but to their quantity.

You can take advantage of big fonts even in a non-Unicode program, which means that you can use big fonts to display characters in several different alphabets. However, you need to go beyond the *GetStockObject* function in obtaining a font to select into a device context.

The functions *CreateFont* and *CreateFontIndirect* create a logical font, similar to the way *CreatePen* creates a logical pen and *CreateBrush* creates a logical brush. *CreateFont* has 14 arguments that describe the font you want to create. *CreateFontIndirect* has one argument, but that argument is a pointer to a LOGFONT structure, which has 14 fields that correspond to the arguments of the *CreateFont* function. I’ll discuss these functions in more detail in Chapter 17. For now, we’ll look at the *CreateFont* function, but we’ll focus on only a couple arguments. All the other arguments can be set to zero.

If you need a fixed-pitch font (as we’ve been using for the KEYVIEW1 program), set the thirteenth argument to *CreateFont* to FIXED\_PITCH. If you need a font of a nondefault character set (as we will be needing), set the ninth argument to *CreateFont* to something called the “character set ID.” This character set ID will be one of the following values defined in WINGDI.H. I’ve added comments that indicate the code pages associated with these character sets:

```
#define ANSI_CHARSET          0          // 1252 Latin 1 (ANSI)
#define DEFAULT_CHARSET      1
#define SYMBOL_CHARSET       2
#define MAC_CHARSET          77
#define SHIFTJIS_CHARSET     128         // 932 (DBCS, Japanese)
#define HANGEUL_CHARSET      129         // 949 (DBCS, Korean)
#define HANGUL_CHARSET       129         // " "
#define JOHAB_CHARSET        130         // 1361 (DBCS, Korean)
#define GB2312_CHARSET        134         // 936 (DBCS, Simplified Chinese)
#define CHINESEBIG5_CHARSET   136         // 950 (DBCS, Traditional Chinese)
#define GREEK_CHARSET         161         // 1253 Greek
#define TURKISH_CHARSET       162         // 1254 Latin 5 (Turkish)
#define VIETNAMESE_CHARSET    163         // 1258 Vietnamese
#define HEBREW_CHARSET        177         // 1255 Hebrew
#define ARABIC_CHARSET        178         // 1256 Arabic
```

(continued)

```

#define BALTIC_CHARSET      186    // 1257 Baltic Rim
#define RUSSIAN_CHARSET     204    // 1251 Cyrillic (Slavic)
#define THAI_CHARSET        222    // 874 Thai
#define EASTEUROPE_CHARSET  238    // 1250 Latin 2 (Central Europe)
#define OEM_CHARSET         255    // Depends on country

```

Why does Windows have two different numbers—a character set ID and a code page ID—to refer to the same character sets? It's just one of the confusing quirks in Windows. Notice that the character set ID requires only 1 byte of storage, which is the size of the character set field in the LOGFONT structure. (Back in the Windows 1.0 days, memory and storage space were limited and every byte counted.) Notice that many different MS-DOS code pages are used in other countries, but only one character set ID—OEM\_CHARSET—is used to refer to the MS-DOS character set.

You'll also notice that these character set values agree with the "CharSet" value shown on the top line of the STOKFONT program. In the American English version of Windows, we saw stock fonts that had character set IDs of 0 (ANSI\_CHARSET) and 255 (OEM\_CHARSET). We saw 161 (GREEK\_CHARSET) in the Greek version of Windows, 204 (RUSSIAN\_CHARSET) in the Russian version, and 128 (SHIFTJIS\_CHARSET) in the Japanese version.

In the code above, DBCS stands for double-byte character set, which is used in the Far East versions of Windows. Other versions of Windows do not support DBCS fonts, so you can't use those character set IDs.

*CreateFont* returns an HFONT value—a handle to a logical font. You can select this font into a device context using *SelectObject*. You must eventually delete every logical font you create by calling *DeleteObject*.

The other part of the big font solution is the WM\_INPUTLANGCHANGE message. Whenever you change the keyboard layout using the popup menu in the desktop tray, Windows sends your window procedure the WM\_INPUTLANGCHANGE message. The *wParam* message parameter is the character set ID of the new keyboard layout.

The KEYVIEW2 program shown in Figure 6-12 implements logic to change the font whenever the keyboard layout changes.

## KEYVIEW2.C

```

/*-----
KEYVIEW2.C -- Displays Keyboard and Character Messages
              (c) Charles Petzold, 1998
-----*/

#include <windows.h>

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

```

**Figure 6-12.** *The KEYVIEW2 program.*

```

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("KeyView2") ;
    HWND          hwnd ;
    MSG           msg ;
    WNDCLASS      wndclass ;

    wndclass.style      = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc = WndProc ;
    wndclass.cbClsExtra  = 0 ;
    wndclass.cbWndExtra  = 0 ;
    wndclass.hInstance   = hInstance ;
    wndclass.hIcon        = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor      = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName = NULL ;
    wndclass.lpszClassName = szAppName ;

    if (!RegisterClass (&wndclass))
    {
        MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                    szAppName, MB_ICONERROR) ;
        return 0 ;
    }

    hwnd = CreateWindow (szAppName, TEXT ("Keyboard Message Viewer #2"),
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)

```

(continued)

Figure 6-12. *continued*

```

{
    static DWORD dwCharSet = DEFAULT_CHARSET ;
    static int   cxClientMax, cyClientMax, cxClient, cyClient, cxChar, cyChar ;
    static int   cLinesMax, cLines ;
    static PMSG  pmsg ;
    static RECT  rectScroll ;
    static TCHAR szTop[] = TEXT ("Message      Key      Char      ")
                          TEXT ("Repeat Scan Ext ALT Prev Tran") ;
    static TCHAR szUnd[] = TEXT ("_____      _____      ")
                          TEXT ("_____      _____      ") ;

    static TCHAR * szFormat[2] = {
        TEXT ("% -13s %3d %-15s%c%6u %4d %3s %3s %4s %4s"),
        TEXT ("% -13s                0x%04X%1s%c %6u %4d %3s %3s %4s %4s") } ;

    static TCHAR * szYes  = TEXT ("Yes") ;
    static TCHAR * szNo   = TEXT ("No") ;
    static TCHAR * szDown = TEXT ("Down") ;
    static TCHAR * szUp   = TEXT ("Up") ;

    static TCHAR * szMessage [] = {
        TEXT ("WM_KEYDOWN"),    TEXT ("WM_KEYUP"),
        TEXT ("WM_CHAR"),       TEXT ("WM_DEADCHAR"),
        TEXT ("WM_SYSKEYDOWN"), TEXT ("WM_SYSKEYUP"),
        TEXT ("WM_SYSCHAR"),    TEXT ("WM_SYSDEADCHAR") } ;

    HDC          hdc ;
    int           i, iType ;
    PAINTSTRUCT  ps ;
    TCHAR         szBuffer[128], szKeyName [32] ;
    TEXTMETRIC   tm ;

    switch (message)
    {
    case WM_INPUTLANGCHANGE:
        dwCharSet = wParam ;

        // fall through

    case WM_CREATE:
    case WM_DISPLAYCHANGE:

        // Get maximum size of client area

        cxClientMax = GetSystemMetrics (SM_CXMAXIMIZED) ;
        cyClientMax = GetSystemMetrics (SM_CYMAXIMIZED) ;

        // Get character size for fixed-pitch font

```

```

hdc = GetDC (hwnd) ;

SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                               dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

GetTextMetrics (hdc, &tm) ;
cxChar = tm.tmAveCharWidth ;
cyChar = tm.tmHeight ;

DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
ReleaseDC (hwnd, hdc) ;

    // Allocate memory for display lines

if (pmsg)
    free (pmsg) ;

cLinesMax = cyClientMax / cyChar ;
pmsg = malloc (cLinesMax * sizeof (MSG)) ;
cLines = 0 ;

                                // fall through
case WM_SIZE:
    if (message == WM_SIZE)
    {
        cxClient = LOWORD (lParam) ;
        cyClient = HIWORD (lParam) ;
    }

    // Calculate scrolling rectangle

rectScroll.left   = 0 ;
rectScroll.right  = cxClient ;
rectScroll.top    = cyChar ;
rectScroll.bottom = cyChar * (cyClient / cyChar) ;

InvalidateRect (hwnd, NULL, TRUE) ;

if (message == WM_INPUTLANGCHANGE)
    return TRUE ;

return 0 ;

case WM_KEYDOWN:
case WM_KEYUP:
case WM_CHAR:
case WM_DEADCHAR:
case WM_SYSKEYDOWN:

```

(continued)

Figure 6-12. *continued*

```

case WM_SYSKEYUP:
case WM_SYSCHAR:
case WM_SYSDEADCHAR:

    // Rearrange storage array

    for (i = cLinesMax - 1 ; i > 0 ; i--)
    {
        pmsg[i] = pmsg[i - 1] ;
    }

    // Store new message

    pmsg[0].hwnd = hwnd ;
    pmsg[0].message = message ;
    pmsg[0].wParam = wParam ;
    pmsg[0].lParam = lParam ;

    cLines = min (cLines + 1, cLinesMax) ;

    // Scroll up the display

    ScrollWindow (hwnd, 0, -cyChar, &rectScroll, &rectScroll) ;

    break ;          // ie, call DefWindowProc so Sys messages work

case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;

    SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                   dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

    SetBkMode (hdc, TRANSPARENT) ;
    TextOut (hdc, 0, 0, szTop, strlen (szTop)) ;
    TextOut (hdc, 0, 0, szUnd, strlen (szUnd)) ;

    for (i = 0 ; i < min (cLines, cyClient / cyChar - 1) ; i++)
    {
        iType = pmsg[i].message == WM_CHAR ||
                pmsg[i].message == WM_SYSCHAR ||
                pmsg[i].message == WM_DEADCHAR ||
                pmsg[i].message == WM_SYSDEADCHAR ;

        GetKeyNameText (pmsg[i].lParam, szKeyName,
                        sizeof (szKeyName) / sizeof (TCHAR)) ;

        TextOut (hdc, 0, (cyClient / cyChar - 1 - i) * cyChar, szBuffer,
                wprintf (szBuffer, szFormat [iType],

```

```

WM_KEYFIRST],

        szMessage [pmsg[i].message -
        pmsg[i].wParam,
        (PTSTR) (iType ? TEXT (" ") : szKeyName),
        (TCHAR) (iType ? pmsg[i].wParam : ' '),
        LOWORD (pmsg[i].lParam),
        HIWORD (pmsg[i].lParam) & 0xFF,
        0x01000000 & pmsg[i].lParam ? szYes : szNo,
        0x20000000 & pmsg[i].lParam ? szYes : szNo,
        0x40000000 & pmsg[i].lParam ? szDown : szUp,
        0x80000000 & pmsg[i].lParam ? szUp : szDown)) ;
    }
    DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
    EndPaint (hwnd, &ps) ;
    return 0 ;

case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
}

```

Notice that KEYVIEW2 clears the screen and reallocates its storage space whenever the keyboard input language changes. There are two reasons for this: First, because KEYVIEW2 isn't being specific about the font it wants, the size of the font characters can change when the input language changes. The program needs to recalculate some variables based on the new character size. Second, KEYVIEW2 doesn't retain the character set ID in effect at the time it receives each character message. Thus, if the keyboard input language changed and KEYVIEW2 needed to redraw its client area, all the characters would be displayed with the new font.

I'll discuss fonts and character sets more in Chapter 17. If you'd like to research internationalization issues more, you can find documentation at */Platform SDK/Windows Base Services/International Features*, but much essential information is also located in */Platform SDK/Windows Base Services/General Library/String Manipulation*.

## THE CARET (NOT THE CURSOR)

When you type text into a program, generally a little underline, vertical bar, or box shows you where the next character you type will appear on the screen. You may know this as a "cursor," but you'll have to get out of that habit when programming for Windows. In Windows, it's called the "caret." The word "cursor" is reserved for the little bitmap image that represents the mouse position.

## The Caret Functions

There are five essential caret functions:

- *CreateCaret* Creates a caret associated with a window.
- *SetCaretPos* Sets the position of the caret within the window.
- *ShowCaret* Shows the caret.
- *HideCaret* Hides the caret.
- *DestroyCaret* Destroys the caret.

There are also functions to get the current caret position (*GetCaretPos*) and to get and set the caret blink time (*GetCaretBlinkTime* and *SetCaretBlinkTime*).

In Windows, the caret is customarily a horizontal line or box that is the size of a character, or a vertical line that is the height of a character. The vertical line caret is recommended when you use a proportional font such as the Windows default system font. Because the characters in a proportional font are not of a fixed size, the horizontal line or box can't be set to the size of a character.

If you need a caret in your program, you should not simply create it during the WM\_CREATE message of your window procedure and destroy it during the WM\_DESTROY message. The reason this is not advised is that a message queue can support only one caret. Thus, if your program has more than one window, the windows must effectively share the same caret.

This is not as restrictive as it sounds. When you think about it, the display of a caret in a window makes sense only when the window has the input focus. Indeed, the existence of a blinking caret is one of the visual cues that allows a user to recognize that he or she may type text into a program. Since only one window has the input focus at any time, it doesn't make sense for multiple windows to have carets blinking all at the same time.

A program can determine if it has the input focus by processing the WM\_SETFOCUS and WM\_KILLFOCUS messages. As the names imply, a window procedure receives a WM\_SETFOCUS message when it receives the input focus and a WM\_KILLFOCUS message when it loses the input focus. These messages occur in pairs: A window procedure will always receive a WM\_SETFOCUS message before it receives a WM\_KILLFOCUS message, and it always receives an equal number of WM\_SETFOCUS and WM\_KILLFOCUS messages over the course of the window's lifetime.

The main rule for using the caret is simple: a window procedure calls *CreateCaret* during the WM\_SETFOCUS message and *DestroyCaret* during the WM\_KILLFOCUS message.

There are a few other rules: The caret is created hidden. After calling *CreateCaret*, the window procedure must call *ShowCaret* for the caret to be visible. In addition, the window procedure must hide the caret by calling *HideCaret* whenever it draws something on its window during a message other than WM\_PAINT. After it finishes drawing on the

window, the program calls *ShowCaret* to display the caret again. The effect of *HideCaret* is additive: if you call *HideCaret* several times without calling *ShowCaret*, you must call *ShowCaret* the same number of times before the caret becomes visible again.

## The TYPER Program

The TYPER program shown in Figure 6-13 brings together much of what we've learned in this chapter. You can think of TYPER as an extremely rudimentary text editor. You can type in the window, move the cursor (I mean caret) around with the cursor movement keys (or are they caret movement keys?), and erase the contents of the window by pressing Escape. The contents of the window are also erased when you resize the window or change the keyboard input language. There's no scrolling, no search and replace, no way to save files, no spelling checker, and no anthropomorphous paper clip, but it's a start.

### TYPER.C

```
/*-----
  TYPER.C -- Typing Program
             (c) Charles Petzold, 1998
  -----*/

#include <windows.h>

#define BUFFER(x,y) *(pBuffer + y * cxBuffer + x)

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

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT ("Typer") ;
    HWND          hwnd ;
    MSG           msg ;
    WNDCLASS      wndclass ;

    wndclass.style          = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc    = WndProc ;
    wndclass.cbClsExtra     = 0 ;
    wndclass.cbWndExtra     = 0 ;
    wndclass.hInstance      = hInstance ;
    wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground  = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName   = NULL ;
    wndclass.lpszClassName  = szAppName ;
```

Figure 6-13. The TYPER program.

(continued)

Figure 6-13. *continued*

```

    if (!RegisterClass (&wndclass))
    {
        MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                    szAppName, MB_ICONERROR) ;
        return 0 ;
    }

    hwnd = CreateWindow (szAppName, TEXT ("Typing Program"),
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg) ;
        DispatchMessage (&msg) ;
    }
    return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static DWORD    dwCharSet = DEFAULT_CHARSET ;
    static int      cxChar, cyChar, cxClient, cyClient, cxBuffer, cyBuffer,
                    xCaret, yCaret ;
    static TCHAR *  pBuffer = NULL ;
    HDC             hdc ;
    int             x, y, i ;
    PAINTSTRUCT     ps ;
    TEXTMETRIC      tm ;

    switch (message)
    {
    case WM_INPUTLANGCHANGE:
        dwCharSet = wParam ;
        // fall through

    case WM_CREATE:
        hdc = GetDC (hwnd) ;
        SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                       dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;
    }
}

```

```

    GetTextMetrics (hdc, &tm) ;
    cxChar = tm.tmAveCharWidth ;
    cyChar = tm.tmHeight ;

    DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
    ReleaseDC (hwnd, hdc) ;
                                     // fall through
case WM_SIZE:
    // obtain window size in pixels

    if (message == WM_SIZE)
    {
        cxClient = LOWORD (lParam) ;
        cyClient = HIWORD (lParam) ;
    }
    // calculate window size in characters

    cxBuffer = max (1, cxClient / cxChar) ;
    cyBuffer = max (1, cyClient / cyChar) ;

    // allocate memory for buffer and clear it

    if (pBuffer != NULL)
        free (pBuffer) ;

    pBuffer = (TCHAR *) malloc (cxBuffer * cyBuffer * sizeof (TCHAR)) ;

    for (y = 0 ; y < cyBuffer ; y++)
        for (x = 0 ; x < cxBuffer ; x++)
            BUFFER(x,y) = ' ' ;

    // set caret to upper left corner

    xCaret = 0 ;
    yCaret = 0 ;

    if (hwnd == GetFocus ())
        SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;

    InvalidateRect (hwnd, NULL, TRUE) ;
    return 0 ;

case WM_SETFOCUS:
    // create and show the caret

```

*(continued)*

Figure 6-13. *continued*

```

        CreateCaret (hwnd, NULL, cxChar, cyChar) ;
        SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
        ShowCaret (hwnd) ;
        return 0 ;

    case WM_KILLFOCUS:
        // hide and destroy the caret

        HideCaret (hwnd) ;
        DestroyCaret () ;
        return 0 ;

    case WM_KEYDOWN:
        switch (wParam)
        {
            case VK_HOME:
                xCaret = 0 ;
                break ;

            case VK_END:
                xCaret = cxBuffer - 1 ;
                break ;

            case VK_PRIOR:
                yCaret = 0 ;
                break ;

            case VK_NEXT:
                yCaret = cyBuffer - 1 ;
                break ;

            case VK_LEFT:
                xCaret = max (xCaret - 1, 0) ;
                break ;

            case VK_RIGHT:
                xCaret = min (xCaret + 1, cxBuffer - 1) ;
                break ;

            case VK_UP:
                yCaret = max (yCaret - 1, 0) ;
                break ;

            case VK_DOWN:
                yCaret = min (yCaret + 1, cyBuffer - 1) ;
                break ;
        }

```

```

case VK_DELETE:
    for (x = xCaret ; x < cxBuffer - 1 ; x++)
        BUFFER (x, yCaret) = BUFFER (x + 1, yCaret) ;

    BUFFER (cxBuffer - 1, yCaret) = ' ' ;

    HideCaret (hwnd) ;
    hdc = GetDC (hwnd) ;

    SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                   dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

    TextOut (hdc, xCaret * cxChar, yCaret * cyChar,
             & BUFFER (xCaret, yCaret),
             cxBuffer - xCaret) ;

    DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
    ReleaseDC (hwnd, hdc) ;
    ShowCaret (hwnd) ;
    break ;
}
SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
return 0 ;

case WM_CHAR:
    for (i = 0 ; i < (int) LOWORD (lParam) ; i++)
    {
        switch (wParam)
        {
            case '\b':                // backspace
                if (xCaret > 0)
                {
                    xCaret-- ;
                    SendMessage (hwnd, WM_KEYDOWN, VK_DELETE, 1) ;
                }
                break ;

            case '\t':                // tab
                do
                {
                    SendMessage (hwnd, WM_CHAR, ' ', 1) ;
                }
                while (xCaret % 8 != 0) ;
                break ;
        }
    }
}

```

*(continued)*

Figure 6-13. *continued*

```

        case '\n':                                // line feed
            if (++yCaret == cyBuffer)
                yCaret = 0 ;
            break ;

        case '\r':                                // carriage return
            xCaret = 0 ;

            if (++yCaret == cyBuffer)
                yCaret = 0 ;
            break ;

        case '\x1B':                              // escape
            for (y = 0 ; y < cyBuffer ; y++)
                for (x = 0 ; x < cxBuffer ; x++)
                    BUFFER (x, y) = ' ' ;

            xCaret = 0 ;
            yCaret = 0 ;

            InvalidateRect (hwnd, NULL, FALSE) ;
            break ;

        default:                                  // character codes
            BUFFER (xCaret, yCaret) = (TCHAR) wParam ;

            HideCaret (hwnd) ;
            hdc = GetDC (hwnd) ;

            SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                           dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

            TextOut (hdc, xCaret * cxChar, yCaret * cyChar,
                    & BUFFER (xCaret, yCaret), 1) ;

            DeleteObject (
                SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
            ReleaseDC (hwnd, hdc) ;
            ShowCaret (hwnd) ;

            if (++xCaret == cxBuffer)
            {
                xCaret = 0 ;
            }

```

```

        if (++yCaret == cyBuffer)
            yCaret = 0 ;
    }
    break ;
}

SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
return 0 ;

case WM_PAINT:
    hdc = BeginPaint (hwnd, &ps) ;

    SelectObject (hdc, CreateFont (0, 0, 0, 0, 0, 0, 0, 0,
                                   dwCharSet, 0, 0, 0, FIXED_PITCH, NULL)) ;

    for (y = 0 ; y < cyBuffer ; y++)
        TextOut (hdc, 0, y * cyChar, & BUFFER(0,y), cxBuffer) ;

    DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
    EndPaint (hwnd, &ps) ;
    return 0 ;

case WM_DESTROY:
    PostQuitMessage (0) ;
    return 0 ;
}
return DefWindowProc (hwnd, message, wParam, lParam) ;
}

```

To keep things reasonably simple, TYPED uses a fixed-pitch font. Writing a text editor for a proportional font is, as you might imagine, much more difficult. The program obtains a device context in several places: during the WM\_CREATE message, the WM\_KEYDOWN message, the WM\_CHAR message, and the WM\_PAINT message. Each time, calls to *GetStockObject* and *SelectObject* select a fixed-pitch font with the current character set.

During the WM\_SIZE message, TYPED calculates the character width and height of the window and saves these values in the variables *cxBuffer* and *cyBuffer*. It then uses *malloc* to allocate a buffer to hold all the characters that can be typed in the window. Notice that the size of this buffer in bytes is the product of *cxBuffer*, *cyBuffer*, and *sizeof (TCHAR)*, which can be 1 or 2 depending on whether the program is compiled for 8-bit character processing or Unicode.

The *xCaret* and *yCaret* variables store the character position of the caret. During the WM\_SETFOCUS message, TYPED calls *CreateCaret* to create a caret that is the width and

height of a character. It then calls *SetCaretPos* to set the caret position and *ShowCaret* to make the caret visible. During the WM\_KILLFOCUS message, TYPED calls *HideCaret* and *DestroyCaret*.

The WM\_KEYDOWN processing mostly involves the cursor movement keys. Home and End send the caret to the beginning and end of a line, and Page Up and Page Down send the caret to the top and bottom of the window. The arrow keys work as you would expect. For the Delete key, TYPED must move everything remaining in the buffer from the next caret position to the end of the line and then display a blank space at the end of the line.

The WM\_CHAR processing handles the Backspace, Tab, Linefeed (Ctrl-Enter), Enter, Escape, and character keys. Notice that I've used Repeat Count in *lParam* when processing the WM\_CHAR message (under the assumption that every character the user types is important) but not during the WM\_KEYDOWN message (to prevent inadvertent over-scrolling). The Backspace and Tab processing is simplified somewhat by the use of the *SendMessage* function. Backspace is emulated by the Delete logic, and Tab is emulated by a series of spaces.

As I mentioned earlier, a program should hide the caret when drawing on the window during messages other than WM\_PAINT. TYPED does this when processing the WM\_KEYDOWN message for the Delete key and the WM\_CHAR message for character keys. In both these cases, TYPED alters the contents of the buffer and then draws the new character or characters on the window.

Although TYPED uses the same logic as KEYVIEW2 to switch between character sets as the user switches keyboard layouts, it does not work quite right for Far Eastern versions of Windows. TYPED does not make any allowance for the double-width characters. This raises issues that are better covered in Chapter 17, which explores fonts and text output in more detail.