

CAUTION
Computer Disk Inside
Do NOT Demagnetize

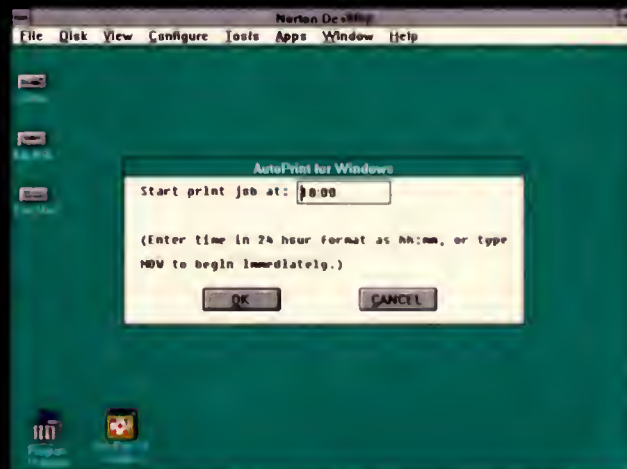
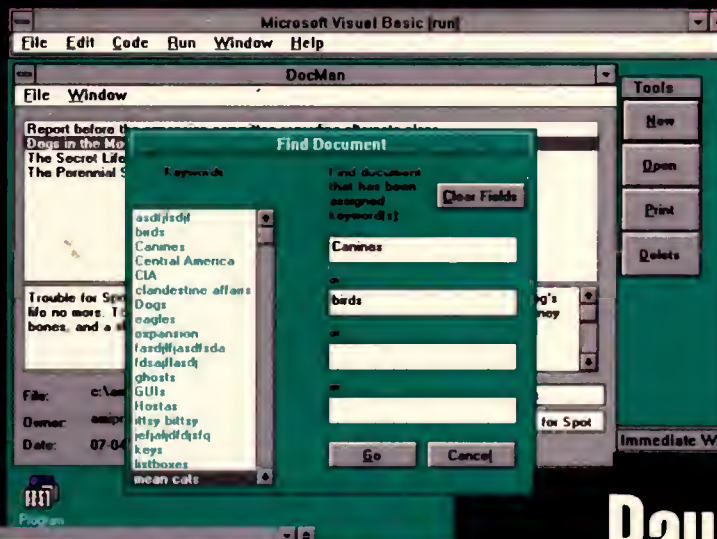
DISK 1



PC Computing

Customizing Windows 3.1

Take full control of Windows 3.1 with this unprecedented collection of techniques and insights from *PC/Computing's* senior editor and Windows expert Paul Bonner. You will quickly learn to customize Windows 3.1, customize existing Windows applications, and write your own programs in the Windows 3.1 environment. Includes a disk with source code and compiled code for use with leading applications like Microsoft Excel, Ami Pro, Norton Desktop for Windows, Visual BASIC, DynaComm, and more.



Paul Bonner

Handwritten text at the top of the page, possibly a title or header, which is mostly illegible due to blurring and fading. Some faint characters are visible, including what appears to be "18" and "19".

WAY

WITHDRAWN

3 1833 02118 5688

005.43 W72bo
Bonner, Paul.
PC/Computing customizing
Windows 3.1

~~JUN 26 1992~~

CAUTION
Computer Disk Inside
Do NOT Demagnetize

DO NOT REMOVE
CARDS FROM POCKET

ALLEN COUNTY PUBLIC LIBRARY
FORT WAYNE, INDIANA 46802

You may return this book to any agency, branch,
or bookmobile of the Allen County Public Library.

**NOTICE:
Warning of Copyright
Restrictions**

The Copyright law of the United States (Title 17, United States Code) governs the reproduction, distribution, adaptation, public performance, and public display of copyrighted material.

Under certain conditions of the law, non-profit libraries are authorized to lend, lease, or rent copies of computer programs to patrons on a nonprofit basis and for non-profit purposes. Any person who makes an unauthorized copy or adaptation of the computer program, or redistributes the loan copy, or publicly performs or displays the computer program, except as permitted by Title 17 of the United States Code, may be liable for copyright infringement.

This institution reserves the right to refuse to fulfill a loan request if, in its judgment, fulfillment of the request would lead to violation of the copyright law.

PC/Computing Customizing Windows 3.1

PLEASE NOTE—USE OF THE DISK(S) AND THE PROGRAMS INCLUDED ON THE DISK(S) PACKAGED WITH THIS BOOK AND THE PROGRAM LISTINGS INCLUDED IN THIS BOOK IS SUBJECT TO AN END-USER LICENSE AGREEMENT (THE "AGREEMENT") FOUND AT THE BACK OF THE BOOK. PLEASE READ THE AGREEMENT CAREFULLY BEFORE MAKING YOUR PURCHASE DECISION. PURCHASE OF THE BOOK AND USE OF THE DISKS, PROGRAMS, AND PROGRAM LISTINGS WILL CONSTITUTE ACCEPTANCE OF THE AGREEMENT.



Digitized by the Internet Archive
in 2012

<http://www.archive.org/details/pccomputingcusto00bonn>

PC Computing[®]

PC/Computing
Customizing
Windows 3.1

Paul Bonner



Ziff-Davis Press
Emeryville, California

**Allen County Public Library
Ft. Wayne, Indiana**

Editor	Leslie Tilley
Technical Reviewer	Neil Rubenking
Project Coordinator	Ami Knox
Proofreader	Aidan Wylde
Cover Design	Mike Yapp, Ken Roberts
Book Design	Laura Lamar/MAX, San Francisco; Stephen Bradshaw
Technical Illustration	Cherie Plumlee Computer Graphics & Illustration
Word Processing	Howard Blechman, Cat Haglund, Kim Haglund
Page Layout	Adrian Severynen and Anna L. Marks
Indexer	Valerie Robbins

This book was produced on a Macintosh IIx, with the following applications: FrameMaker[®], Microsoft[®] Word, MacLink[®] Plus, Aldus[®] FreeHand[™], Adobe Photoshop[™], and Collage Plus[™].

Ziff-Davis Press
5903 Christie Avenue
Emeryville, CA 94608

Copyright © 1992 by Paul Bonner. All rights reserved.

PC/Computing is a registered trademark of Ziff Communications Company. Ziff-Davis Press and ZD Press are trademarks of Ziff Communications Company.

All other product names and services identified throughout this book are trademarks or registered trademarks of their respective companies. They are used throughout this book in editorial fashion only and for the benefit of such companies. No such uses, or the use of any trade name, is intended to convey endorsement or other affiliation with the book.

No part of this publication may be reproduced in any form, or stored in a database or retrieval system, or transmitted or distributed in any form by any means, electronic, mechanical photocopying, recording, or otherwise, without the prior written permission of Ziff-Davis Press, except as permitted by the Copyright Act of 1976 and the End-User License Agreement at the back of this book and except that program listings may be entered, stored, and executed in a computer system.

EXCEPT FOR THE LIMITED WARRANTY COVERING THE PHYSICAL DISK(S) PACKAGED WITH THIS BOOK AS PROVIDED IN THE END-USER LICENSE AGREEMENT AT THE BACK OF THIS BOOK, THE INFORMATION AND MATERIAL CONTAINED IN THIS BOOK ARE PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION ANY WARRANTY CONCERNING THE ACCURACY, ADEQUACY, OR COMPLETENESS OF SUCH INFORMATION OR MATERIAL OR THE RESULTS TO BE OBTAINED FROM USING SUCH INFORMATION OR MATERIAL. NEITHER ZIFF-DAVIS PRESS NOR THE AUTHOR SHALL BE RESPONSIBLE FOR ANY CLAIMS ATTRIBUTABLE TO ERRORS, OMISSIONS, OR OTHER INACCURACIES IN THE INFORMATION OR MATERIAL CONTAINED IN THIS BOOK, AND IN NO EVENT SHALL ZIFF-DAVIS PRESS OR THE AUTHOR BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF SUCH INFORMATION OR MATERIAL.

ISBN 1-56276-018-1
Manufactured in the United States of America
10987654321



This book is dedicated to my wife Betsy Woldman and my friend Orlan Cannon.

Betsy's patience and good humor throughout the long months that I worked on *PC/Computing Customizing Windows 3.1* made this project possible, and her love made it worth doing.

As for Orlan, since he was responsible over a four- or five-year span for introducing me to Betsy, professional writing, coffee, and computers, there is no doubt that this book could not have been written without him.

C O N T E N T S A T A G L A N C E

Introduction xix

Part 1: Application-Development Concepts and Tools

Chapter 1: Building a Great Windows Application 3

Chapter 2: Programming Basics 13

Chapter 3: Cut-and-Paste Programming 41

Chapter 4: Choosing Your Tools 55

Part 2: The Application-Development Process

Chapter 5: Principles of Application Design 97

Chapter 6: The Nuts and Bolts of Application Design 111

Chapter 7: Implementing a Windows Interface 137

Chapter 8: Prototypes, Testing, and Documentation 165

Part 3: The Projects

Chapter 9: Customizing Applications—The Ultimate Notepad 181

Chapter 10: Presenting Data—Who's Who at PC/Computing 209

Chapter 11: Automating Existing Applications—AutoPrint for
Windows 239

Chapter 12: Making Use of Libraries—Recycler 267

Chapter 13: Linking Applications through DDE—Windows Broker 295

Chapter 14: Enhancing Applications—DocMan 337

Chapter 15: Communicating with Host Systems—M.M.M.: the MCI Mail
Manager 403

Appendix A: Commercial DLLs and Custom Controls 521

Appendix B: Companion Disk Instructions 536

Index 545

TABLE OF CONTENTS

Introduction: xix

Swimming Lessons xix

How to Avoid the Windows SDK xx

An Overview of the Chapters xxi

Part 1: Application-Development Concepts and Tools xxii

Part 2: The Application-Development Process xxii

Part 3: The Projects xxiii

What's on the Disk xxiv

Part 1: Application-Development Concepts and Tools

Chapter 1: Building a Great Windows Application 3

Multitasking under Windows 5

Interrupt Handling and Finger Knowledge 6

Multiple-Application Processes 7

Designing for Multitasking 8

Shared Processing 8

Shared Facilities 9

Shared Interface 10

Chapter 2: Programming Basics 13

The Elements of Programming 13

Commands 14

Variables 16

Functions 20

Expressions 23

Comments 27

Arrays 28

Program Control Elements 29

Decision Making 29

Loops 30

Input/Output 33

Named Subroutines 34

Error Handling 36

User Errors 37

Learning to Program 38

Chapter 3: Cut-and-Paste Programming 41

- Customizing Applications 43
 - Customizing with Macro Languages 44
 - Customizing with Batch Languages 46
- Linking Existing Applications 48
 - Linking with Macro versus Batch Languages 48
- Using Third Party Code 50
 - Incorporating Dynamic Link Libraries 51
- Making Cut-and-Paste Programming Work for You 53

Chapter 4: Choosing Your Tools 55

- Evaluating Tools 58
 - Multitasking Facilities 58
 - User Interface Facilities 63
 - File Formats 69
- Windows Development Tools 69
 - Application Macro Languages 70
 - Windows Batch Languages 80
 - Windows BASICS 85
 - Pascal 90
 - Graphical Hypertext Products 90

Part 2: The Application-Development Process**Chapter 5: Principles of Application Design 97**

- Designing for the Designer 97
- Designing for the User 98
- Bonner's Usability Guidelines 99
 - Rule 1: Fit Applications into the Current Work Flow 100
 - Rule 2: Improve On Existing Methods 101
 - Rule 3: Don't Surprise the User 102
 - Rule 4: Try to Delight the User 104
 - Rule 5: Finish the Job 105
 - Rule 6: Make Applications Open-ended 105
 - Rule 7: Design for Reliability 106
 - Rule 8: Don't Overwhelm New Users 106

-
- Rule 9: Don't Delay Experienced Users 107
 - Rule 10: Design for the User's Convenience—Not Your Own 107

Chapter 6: The Nuts and Bolts of Application Design 111

- Identifying the Application's Purpose 111
 - Determining Application Requirements 112
- Picking a Development Tool 113
- Drawing a Flowchart 115
 - Flowchart Iterations 116
 - Neater Flowcharts 118
- Defining Input Requirements 120
 - Links to Other Applications 121
 - AutoPrint's Data Requirements 123
- Planning Data Structures 125
 - Defining Variables 125
 - Disk-Based Data Formats 129
 - AutoPrint's Data Structures 132
- Defining Internal Processing 133
- Putting It All Together 135

Chapter 7: Implementing a Windows Interface 137

- Application Window Features 139
 - Title Bar 139
 - Minimize and Maximize Buttons 140
 - Control Menu Box 140
 - Menu Bar 140
 - Scroll Bars 143
 - Application Workspace 143
 - Document Windows 143
- Standard User-Interface Controls 144
 - Command Buttons 144
 - Check Boxes 145
 - Radio Buttons 146
 - Group Boxes 147
 - Icons 148
 - Static Text 150
 - Edit Boxes 151

List Boxes	152
Combo Boxes and Drop-Down Lists	155
The SAA Standard	156
Common Extensions to the Standard	157
Icon Bars	157
Ribbons	157
Active Status Lines	158
The Keyboard Interface	159
Keyboard Navigation	159
Keyboard Shortcuts	160
Dialog Box Design	161
Using Common Dialog Boxes	162
Putting It All Together	163

Chapter 8: Prototypes, Testing, and Documentation 165

Prototyping	165
Iterative Prototypes	166
On-the-Job Training	167
Testing and Debugging	169
Developer Testing	169
Covering Every Base	170
User Testing	174
Documentation	175
On-Line Help	176
Shrink-Wrap Time	176

Part 3: The Projects

Chapter 9: Customizing Applications—The Ultimate Notepad 181

The Birth of a Notion	181
Selecting the Development Tool	181
Setting Objectives	182
Exploring NOTEPAD.WDF	183
Exploring NOTEPAD.WBT	185
Introductory Lines	185
The Subroutine Macros	186

The Dialog Box Routines 204

Wrapping Up The Ultimate Notepad 207

Chapter 10: Presenting Data—Who's Who at PC/Computing 209

In the Beginning Was Confusion 209

Choosing the Tool 210

Application-Design Issues in Plus 211

Stacks 211

Drawing the Interface 212

Exploring the Application 215

The Opening Screen 216

The Search Button Script 220

The Floor Plan Screen 225

The Personnel Card Screen 230

The Organization Chart Screen 234

Wrapping Up the Who's Who Application 236

Chapter 11: Automating Existing Applications—AutoPrint for Windows 239

The Impetus behind AutoPrint 239

Designing the Application 240

The Right Tool for the Job 241

How AutoPrint Works 241

The GETFILE.WBT Batch File 243

GETFILE Dissected 244

The AUTOPRN.WBT Batch File 248

AUTOPRN Dissected 249

Putting the Batch Files to Work 252

The WinBatch Version 254

Macro Files 255

The COPYMAC.WBT Batch File 255

COPYMAC Dissected 256

The AUTOP.WBT Batch File 258

AUTOP Dissected 260

Chapter 12: Making Use of Libraries—Recycler 267

Basic Operations 267

Capabilities and Limitations	269
Defining Functional Requirements	271
Selecting the Development Tool	271
Message Loops	272
DOS Attribute Control	274
The Application Framework	274
Exploring GLOBAL.BAS	275
Type Definitions	275
Function and Subroutine Definitions	276
Global Constant and Variable Definitions	279
Exploring FORM1.FRM	280
General Procedures	282
Event Procedures	282
Exploring DRAGDROP.BAS	288
Wrapping Up Recycler	293

Chapter 13: Linking Applications through DDE—Windows Broker 295

Broker's Origin and Structure	295
Choosing the Tools	295
Application Framework	296
Exploring BROKER1.XLS	296
Worksheet Mechanics	298
Hidden Data	301
Exploring LOTS.XLS and IBM.XLS	302
Exploring the BROKER1.XLM File	304
Navigational Macros	304
Communication Macros	307
Transaction-Recording Macros	313
Exploring BROKER1.DCP	317
The Introductory Routines	318
General-Purpose Subroutines	321
Task-Specific Routines	323
Wrapping Up Windows Broker	334

Chapter 14: Enhancing Applications—DocMan 337

Opening Moves	337
---------------	-----

Functional Requirements	338
Selecting Development Tools	338
A Tour of DocMan	339
The OpenDM Screen	339
The FindDlg Screen	340
The Ami Pro Document Description Dialog Box	342
DocMan's Skeleton	343
Exploring DCGLOBAL.BAS	343
Constant Declarations	343
External-Function Declarations	344
Data-Storage Declarations	345
Variable Declarations	346
Exploring FORM1.FRM	346
Exploring DOCMAN2.FRM	348
General and Loading Routines	348
User-Action Routines	350
Menu Item Routines	360
Exploring ACTIONS.FRM	362
Button Routines	363
Exploring FINDDL.G.FRM	365
General and Loading Routines	365
Event Procedures	368
Exploring ABOUTDLG.FRM	371
Exploring GLOBCODE.BAS	372
Inside Ami Pro	393
Ami Pro Macros	393
Wrapping Up DocMan	399

Chapter 15: Communicating with Host Systems—M.M.M.: the MCI Mail Manager 403

Improving on an Existing Model	403
Selecting the Development Tool	404
Other Possible Approaches	404
M.M.M.'s Capabilities	405
Creating and Editing Messages	405
Transmitting and Receiving Messages	406

Organizing and Managing Messages	406
How M.M.M. Works	407
The Mailboxes Screen	408
Online Options	410
Offline Options	410
Message-Handling Options	411
The Set Up Menu	412
The Phonebook Management Routine	412
Message-Composition Routines	415
Exploring the AUTOMCI.DCP Script	416
Initializing Global Settings and Variables	416
Analyzing the Main Routine	421
Mailboxes Screen Support Routines	427
The Set Up Menu Routine	429
Menu-Support Routines	433
Table-Handling Routines	442
Mailboxes Screen Action Routines	446
Message-Handling Routines	456
Welcome-Message Routines	461
Account Setup Routines	462
The Code Routine	467
Exploring TM.DCP	470
Calling the TM Module	470
Message-Creation Routines	471
Utility Routines	488
Exploring EMAIL.DCP	489
Initializing Global Variables	489
Message-Transmission Routines	490
Message-Reception Routines	501
Standard Library Routines	510
Exploring PM.DCP	511
Exploring ONLINE.DCP	515
The Terminal Routines	516
Wrapping Up M.M.M.	518

Appendix A: Commercial DLLs and Custom Controls 521

Appendix B: Companion Disk Instructions 536

Index 545

ACKNOWLEDGMENTS

I OWE A DEBT TO THE MANY FINE MAGAZINE EDITORS WHO HAVE SHAPED my work through the years by virtue of their skill, knowledge, care, and professionalism, especially Nora Seymour, Jim Seymour, Jean Atelsek, Fred Paul, Preston Gralla, Carol Day, Ernie Baxter, and Mike Edelhart. Mike deserves a special note of thanks here for giving me the green light to develop the Windows Project series of feature stories in *PC/Computing*, which acted as a catalyst for this book. I'm also beholden to Dylan Tweney who assisted me in the research for Appendix A as my final deadlines approached.

I'd also like to express my sincere appreciation to all the folks at Ziff-Davis Press who have worked on this book, especially my publisher, Cindy Hudson, for her inexhaustible enthusiasm and only slightly exhaustible patience; Leslie Tilley, the development editor who managed to turn much of the book into English; and Neil J. Rubenking, for his always-brilliant and often-combustible technical edit. Thanks are also due to Ami Knox and the production group, to Simon Tonner and the marketing department, and to everyone else at the press who had a hand in shaping the book.

Finally, I'd like to thank my agent, Claudette Moore, for helping to ensure that this project was profitable as well as enjoyable.

NOW THAT THE WRITING IS NEARLY DONE, I'VE STOPPED WORRYING that half the people to whom I described this book told me that it was impossible. "It's a guide to Windows programming for nonprogrammers" I would tell them, and they would smile and roll their eyes and mutter something like "Good luck."

Of course, they imagined a different sort of book than the one you're holding in your hands. I'm sure that they expected it would start off with a long tutorial on the basics of programming in C, with a particular emphasis on pointers and data types and the respective benefits of various memory models.

Then too, they undoubtedly expected that at around page 1200 I would have to lead the reader to the heart of darkness itself: the Microsoft Windows Software Development Kit. Then, over the course of the middle chapters (pages 2,500 to 4,800 or so), they likely envisioned me describing Windows messages and default Windows procedures, the secrets of the GDI, and the trickier aspects of registering new clipboard formats. And, positively without a doubt, they foresaw me wrapping the whole thing up on page 6,000 or 6,500 by formally presenting you, the reader, with a bona fide certificate awarding you status as a professional Windows developer.

My skeptical friends imagined this book taking that form because they are professional Windows developers themselves, and that was the course that *they* followed to learn to write Windows programs. They had studied database theory and knew five versions of the Quicksort algorithm by heart and could recite every word of the *Systems Application Architecture Common User Access Advanced Interface Design Guide*, so naturally they expected that any book about Windows programming would have to follow the same path.

But that's not the book I wrote, because that's not the path I followed to become a Windows programmer.

I don't intend any disrespect to the traditional method of turning oneself into a Windows programmer, but even those who have followed it have to admit that it has little appeal for anyone who doesn't intend to make writing Windows applications his or her life's work. For someone who sees Windows programming as a means to an end—as a way to simplify or improve the efficiency of other tasks—rather than as an end unto itself, the time required to learn Windows programming that way is simply unacceptable.

Swimming Lessons

Think of it as learning to swim. If you've got the time, you can do it the traditional way, starting off at the shallow end of the pool and practicing your kicks and learning to breath between strokes, and only gradually, over the course of many weeks, working your way toward the deep water.

But let's say you don't have the time. Your Sunny Sails cruise ship has just run down an iceberg, and the lifeboats are nowhere in sight. With the waters rising fast around you, the last thing you want to do is hop in the shallow end of the pool and ask, "Now what was that about the crawl?"

On the other hand, an accelerated course of training that simply tells you to "Jump in, move your arms a lot, kick, and try not to forget to breath," though somewhat more appropriate given the circumstances, probably won't save your life either.

But what if, as everyone around you went wild trying to hire swimming consultants or looking for lumber to build boats, someone called you over and said, "You realize that there are some great life jackets in this closet. And how about a wet suit to protect you against hypothermia? And check out this—one of those cool jet skis like James Bond uses. They're all here for the taking. Just help yourself."

Well, I'm the help yourself guy, and this book is about all the things I've found that will let you become a Windows programmer without ever learning the difference between GlobalAlloc and GetDC.

This book is also sort of autobiographical, because the methods it promotes, the advice it offers, and the programs it describes are all the result of my own experience learning to write Windows applications. I *know* that you don't need to learn to write C code or ever crack the shrink wrap on the Windows SDK in order to write custom Windows applications, because I've never felt the need to do either one.

Of course, I've never tried to write Microsoft Excel or Aldus PageMaker or Micrografx Designer. If those are the kinds of applications that you want to write—big commercial applications that come in 20-pound boxes—put this book back on the shelf and pick up Charlie Petzold's guide to programming with the SDK—you're going to need it.

If you're like me, however, you're probably more interested in customizing the word processor or spreadsheet you already have to work exactly the way that you want it to than in building a new one from scratch. Life is short, and you've got other things to do than spend the next five years devising table formatting algorithms.

Maybe you want to extend the capabilities of the applications that you already have, or create links between those applications, or build custom applications that will automate or expedite some portion of the things that you already do with your PC or, even better, that you would like to be able to do. If so, then you've found the right book.

How to Avoid the Windows SDK

Before you get the wrong idea, let me say that the Windows SDK is a marvelous tool. It provides professional developers with total access to the

extraordinary range of functions and routines that make Windows 3.1 and Windows 3.1 applications the wonders that they are.

In fact, the SDK is so marvelous that there is almost no reason for anyone to use it anymore. At least, not for anything short of full-scale commercial development efforts. For any other kind of project, there are dozens of better, faster, easier alternatives that have been built with the SDK so that you don't have to use the SDK. These tools, which range from application macro languages to standalone development tools like Visual BASIC, don't require programming experience to use them, just the desire to create custom applications and the willingness to learn some new skills.

It's easy to develop custom Windows applications using these tools, and it's fun. But don't be misled. It also takes a good deal of effort, so it's not for everyone. If the off-the-shelf applications that you already have entirely fit your needs, or if there are off-the-shelf applications available that you think will do so, there's really no need for you to learn to program your own custom applications.

In my experience, however, off-the-shelf applications aren't enough. They're like an off-the-rack suit. You can squeeze into one, but it probably won't be a perfect fit. Fortunately, however, you can put the high-level development tools discussed in this book to work like a crew of tailors to alter off-the-shelf applications until they fit like a custom tailored suit. And, once you've got them working for you, your crew of tailors will be only too glad to throw in an extra pair of pants, or even make an entirely new suit for you from scratch.

I started writing Windows applications precisely because off-the-shelf applications no longer satisfied me. For instance, I rely daily on the MCI Mail electronic mail service, and I grew frustrated at having to manually instruct my modem to dial the MCI access number, navigate through its menus, create messages online, and capture messages that were waiting for me to a disk file. So I used the macro language capabilities of FutureSoft Engineering's DynaComm to automate all those processes and more, resulting in the MCI Mail Manager application described in Chapter 15.

Since then, I've learned how to customize many other Windows applications—ranging from Notepad to Excel—and how to build custom applications from scratch using these high-level development tools. *PC/Computing Customizing Windows 3.1* tells you how you can do the same thing.

An Overview of the Chapters

PC/Computing Customizing Windows 3.1's fifteen chapters are organized in three parts.

Part 1: Application-Development Concepts and Tools

Part 1 consists of Chapters 1 through 4. These chapters provide an overview of basic programming concepts, an in-depth examination of the dozens of high-level development tools available for Windows 3.1, and some advice about how you should approach your Windows development projects.

Chapter 1, “Building a Great Windows Application,” discusses the qualities which separate great Windows applications from mediocre ones: effective use of Windows multitasking facilities, smart interactions with other Windows applications, and, most of all, an overriding concern with usability.

Chapter 2, “Programming Basics,” is an overview of the basic programming concepts and terminology that you’ll need to understand before you can use any Windows development tool.

Chapter 3, “Cut-and-Paste Programming,” shows how you can save time and effort by using the capabilities of the applications that you already own or of commercially available code libraries rather than inventing them from scratch. By applying the concepts discussed in this chapter, you can produce a custom application in a matter of hours that might otherwise take weeks or months to complete.

Chapter 4, “Choosing Your Tools,” begins by discussing the process of selecting a development tool and the qualities that you should look for in evaluating a particular tool. Then it introduces the five general categories of program development tools covered in this book: application macro languages, Windows batch languages, Windows implementation of BASIC, Pascal for Windows, and graphical hypertext products. The chapter concludes with descriptions and recommendations about the best available products in each of those categories.

Part 2: The Application-Development Process

Part 2 includes Chapters 5 through 8, which discuss the key steps involved in designing, writing, testing, and documenting custom applications.

Chapter 5, “Principles of Application Design,” discusses the importance of designing applications for the gratification of the user, not that of the designer, and presents Bonner’s Usability Guidelines, the ten keys to creating truly usable applications. While the distinction made in this chapter between the designer and the user is most valid for those readers who will be building applications for use by other people, the Usability Guidelines are valid even for applications that no one other than the programmer will ever see.

Chapter 6, “The Nuts and Bolts of Application Design,” is a step-by-step guide to the actual application development process: identifying the application’s purpose, determining its functional requirements, selecting a development tool, drawing a flowchart, determining the application’s data

requirements, planning any links to other applications that it might need, creating its data structures, and finally writing the code that puts it all together.

Chapter 7, “Implementing a Windows Interface,” elaborates on the process of planning the user interface for your application and making effective use of Windows 3.1’s powerful interface elements and conventions.

Chapter 8, “Prototypes, Testing, and Documentation,” discusses the process of building iterative prototypes of your application, testing their usability and reliability until you arrive at a solid finished program, and providing documentation for the programs you produce.

Part 3: The Projects

Part 3 puts all the advice and theory from Parts 1 and 2 to work by presenting seven complete programming projects, ranging from simple customization of existing applications to complete standalone programs. In each project chapter, I discuss how the project originated, how I determined what features it needed, how I selected a development tool with which to build it, and then examine the complete source code for the project. (The source code is also available on the accompanying disk.)

Chapter 9, “Customizing Applications: The Ultimate Notepad,” uses WinBatch, a shareware batch language, to customize the Windows 3.1 Notepad application. You’ll create a variety of new features, including a search-and-replace function, the ability to save selected text and to merge a text file into the open file, and a word-count facility.

Chapter 10, “Presenting Data: Who’s Who at PC/Computing?,” uses a graphical hypertext product called Plus to build a custom employee information system.

Chapter 11, “AutoPrint for Windows,” presents two versions of a print queue utility that automatically prints files created by nearly any Windows application at a time that you specify. One version of AutoPrint uses the batch language and other facilities of Norton Desktop for Windows, and the other uses WinBatch.

Chapter 12, “Making Use of Libraries: Recycler,” is a recycling bin for files. Files dragged from the Windows 3.1 File Manager onto this Visual BASIC application seem to disappear, but they can be restored at any time.

Chapter 13, “Linking Applications through DDE: Windows Broker,” is a portfolio analysis and stock-trading program built using Microsoft Excel and FutureSoft Engineer’s asynchronous communications program DynaComm.

Chapter 14, “Enhancing Applications—DocMan,” is a document manager built in Visual BASIC that provides Lotus Development Corporation’s Ami Pro word processor with the ability to access files using long file names or through keyword searches.

Chapter 15, “Communicating with Host Systems—M.M.M.: the MCI Mail Manager,” is a complete electronic mail access and management system for MCI Mail built using DynaComm.

PC/Computing Customizing Windows 3.1 also includes two appendices. The first presents a guide to third-party code libraries and custom controls that can be incorporated into your applications, and the second presents instructions for using the accompanying disk.

What's on the Disk

The disk that accompanies this book contains the source code for the projects presented in the text, as well as other programs that will be useful as you write your applications. Included are

- CW Install, a Windows-based installation program
- Compiled EXE versions of the DocMan and Recycler projects
- A set of Visual BASIC files to use with the functions in the Recycler project
- WinBatch 3.1, for use with the AutoPrint and The Ultimate Notepad projects

You can exchange the 5 1/4-inch disk for a 3 1/2-inch disk; see the disk exchange offer at the back of the book for details.

Learning to customize or create Windows applications isn't effortless, but it may be the best way to increase the benefits you gain from Windows. Once you've learned to use one or more of the development tools discussed in *PC/Computing Customizing Windows 3.1*, and have started to apply the advice it offers, the sky is the limit. Nothing will stand in the way of your making your Windows 3.1 environment and applications work exactly the way you want them to.

Building a Great Windows Application

Programming Basics

Cut-and-Paste Programming

Choosing Your Tools

PART

1

**Application-
Development
Concepts and Tools**

C H A P T E R

1

**Building a Great
Windows
Application**

*Multitasking under
Windows*

*Designing for
Multitasking*

WAY BACK IN 1983, DAN BRICKLIN TAUGHT ME—AND A CONFERENCE room full of other people—how to build great Windows applications. Of course, at the time we weren't learning about Windows, *per se*.

The setting was Applefest Boston, a then-annual computer conference. If you remember back that far, you'll recall that the Apple II was still king of the hill as far as personal computers for business users went. The IBM PC was the new kid on the block, selling fast but not yet the certified leader, and Windows was but a glint in Bill Gates's eye.

Dan was the featured speaker at the show, and I was a kid reporter attending my first computer conference. A guy named Mitch Kapor was also at the conference, showing off a new application called 1-2-3 that his company, Lotus Development Corporation, had recently introduced for the IBM PC. But Dan was the star of the show. He'd achieved legendary status in personal computing circles because he was the first person to actually find a use for personal computers that would capture the hearts and minds (and wallets) of American business: Dan was the principal designer of VisiCalc, the first electronic spreadsheet—and the first certified hit software.

In those days, the people who came to computer conferences weren't interested in debating the relative merits of Windows and OS/2, nor in hearing about GUIs, pen interfaces, connectivity, or RISC architecture, as are audiences today. Instead, they were there to find out what these new things called personal computers were all about and how they differed from the Goliath mainframes that until then were the only computers many of them had ever seen.

Two Boxes

Dan answered those questions by drawing two sets of boxes on a whiteboard.

The first set consisted of a tiny box labeled "User Interface" and a giant box labeled "Processing Power." That set of boxes defined the mainframe: a hugely powerful, hugely unfriendly computing device. As Dan explained, mainframe processor cycles were simply too expensive to waste on the job of supporting a friendly user interface. Instead, they relied on simple dumb-terminal interfaces, which provided access to a great many users, but comfort to none.

Dan's second set of boxes consisted of a giant box labeled "User Interface," and a tiny box labeled "Processing Power." That set defined the nascent personal computer: pitifully weak compared to the mainframe in terms of processing power, but vastly superior in terms of user friendliness.

His point was this: Even though mainframes could run circles around personal computers at crunching numbers and sorting databases, personal computers were inherently superior at the job of interacting with human beings. When the user presses a key on a personal computer, the PC's software reacts, guiding him or her on to the next step in the process. The entire

system—processor, screen, and software—is there to serve the user. In contrast, on mainframe computers it's the user who serves the system—supplying the raw data the system needs to do its job, and distracting the system from that task as little as possible.

The State of the Art

A lot has changed in the nearly ten years since that conference. The IBM PC took over the world, Mitch Kapor became a much better draw than Dan Bricklin, and most importantly, personal computers became a lot more powerful. As I write this, Intel is about to release a new version of the 80486 that performs 40 million instructions per second. Within a year, you'll probably be able to buy a PC built around that chip for about the same \$4,000 that would buy you a complete Apple II system in 1983.

Moreover, in retrospect the “enormously friendly” user interfaces that Dan spoke about that day look pretty darned curmudgeonly. VisiCalc's menu system, for instance, consisted of a single line of one-letter command abbreviations, which popped up at the bottom of the screen when you pressed the slash key. Put that original version of VisiCalc next to a graphical spreadsheet such as Microsoft Excel for Windows, and you'll be apt to ask “What user interface?”

Nevertheless, Dan Bricklin's speech that day captured the essence of what had already made personal computers stand out in 1982, and of what makes state-of-the-art Windows applications stand out today: an overriding concern with the human-computer interaction.

Keeping the User in Mind

In any but the smallest Windows development projects, there are many things you'll have to think about. You have to be concerned with how your application will interact with Windows' multitasking environment. And you've got to consider how dynamic data exchange (DDE) and other methods of interaction between applications can benefit the application you're building.

You also have to outline the process you're trying to automate within Windows, and translate that outline to program or macro code in an efficient and accurate manner. You've got to think about learning new programming languages and face the challenge of writing, testing, and debugging your applications.

With all that to worry about, it would be easy to overlook the simple fact that your program will be used by human beings. The routines, algorithms, and functions you'll build into your application are enormously important to the system: They are what make Windows and your application perform the tasks you want performed. Because of that it would be very easy to scrimp on user-interface design.

Don't make that mistake.

Windows has many wonderful things going for it: multitasking, communications between applications, advanced memory management functions. But the most immediate and important aspect of every Windows application is its user interface, because Windows and Windows applications are there to support the user. If you don't invest the time, energy, and processor cycles necessary for a great user interface, no one will ever find out what wonderful things your application can do. So, of course, you should try to make that box labeled "Processing Power" as big as possible in the Windows applications you build. But you should *absolutely* make sure the box labeled "User Interface" stays even bigger.

Thanks, Dan.

Multitasking under Windows

First-generation PC programs, such as VisiCalc, and second-generation programs, like Lotus 1-2-3, were universes unto themselves. From the time the user typed the command to start those applications, the application owned the PC. It could send data to the screen with impunity because there was no possibility that another application might be using the screen at the same time. It could use the disk with the same nonchalance. It could also assume that data would be input only from the keyboard or from its own files, and it could insist that the user do things in exactly the manner it specified.

That kind of application was fine as long as PC users only needed to use a single application program. Millions of PCs were purchased to run Lotus 1-2-3 or WordPerfect or dBASE, and by and large they did a great job of doing so.

Eventually, however, PC users grew more ambitious. The spreadsheet user wanted to be able to dial into a remote electronic mail (e-mail) service. The word processing user wanted to include charts and graphs in documents. The database designer wanted to use a project management program. And so on.

At that point, the fundamental weakness of those first- and second-generation applications, and of DOS itself, began to grow apparent: Intended to own the PC and all of its resources, they proved poor at the art of sharing.

You couldn't use more than one application at a time, because DOS was a single-tasking operating system. You couldn't keep two applications in memory and switch back and forth easily, because DOS couldn't address enough memory. You could try to supplement your primary application with terminate-and-stay-resident (TSR) utilities, but all too often they proved both unreliable and functionally shallow.

For a long time the best solution to these problems was to use a multi-tasking shell for DOS, such as Quarterdeck's DESQview, which would move programs in and out of memory as needed and, given sufficient RAM,

would even keep several applications running at once on an 80386-based PC. But in solving those problems, DESQview and its competitors exposed another: the incompatibility in both structure and user interface of standard DOS applications.

For example, let's say you were a whiz at Lotus 1-2-3. You could write 1-2-3 macros, build an amortization schedule, and play what-if with your company's finances with the best of them. Then one bright day, somebody handed you a copy of WordPerfect or dBASE or Crosstalk Mk.4. How much did all your 1-2-3 knowledge benefit you then?

Not one iota. No matter how good you were at any of those programs, you'd be lucky even to be able to find a help screen in another program without investing significant time in learning the new application.

Obviously the fault there was not yours. But—and this is a critical point—it really wasn't the fault of the applications either. Each had been designed to provide as direct an interface as possible between the task it set out to do and the user. One might argue quite convincingly that Lotus 1-2-3 was a near-perfect model for the PC user who was interested in spreadsheets and spreadsheets alone. But it, like any DOS application, was completely ignorant of every other DOS application, and hence inadequate for any user who wanted to do more than just run numbers all day long.

Obviously the situation had to change. We needed to start over from scratch with a new computing environment that would address the limitations of both DOS and standard DOS applications.

Windows is that environment.

Interrupt Handling and Finger Knowledge

When you first look at Windows 3.1, it's easy to be dazzled by its colorful icons, impressed by its TrueType WYSIWYG (what-you-see-is-what-you-get) fonts, and amused by its Accessory applications. Those are certainly all important factors in its success, but the most important aspect of Windows 3.1 is that both the Windows environment itself and the applications that run under it are designed for multitasking.

The term *multitasking* means a lot of different things. There's the classic example of a user recalculating a spreadsheet in the background while reading her e-mail and writing a letter. Windows can help you do that if you really want to, but in practice that capability doesn't prove to be all that valuable. People don't switch contexts as fast as computers. Usually when people hit the "recalc" key it's because they want to know the results of the recalculation as fast as possible, so they stick around to see it happen.

That's not to say that background processes aren't at all useful. They're ideal for some tasks, such as an application that checks your electronic mail box regularly and alerts you when incoming messages are retrieved. In fact, background mailbox checks are one of the key features of the application

project called M.M.M.: The MCI Mail Manager, which is presented in Chapter 15.

For most users, however, the most important benefits of using Windows applications designed for multitasking come in three areas: interrupt handling, finger knowledge, and multiple-application processes.

By *interrupt handling* I don't mean the process by which the PC BIOS interacts with the keyboard and communications ports. Rather, I mean the way human beings handle interruptions. Say you're in the middle of writing a memo in your word processor and your boss calls demanding the latest numbers from your sales report spreadsheet. No problem. A few clicks of the mouse will bring your spreadsheet to the fore, while your memo waits in the background for you to finish handling the interruption.

Finger knowledge refers to the knowledge one acquires about the operations of an application. In Lotus 1-2-3 your fingers know that typing **/FR** will present you with a list of available worksheet files. In Excel the same thing can be achieved by pressing Alt-F O. So far the two programs seem about even—three keys each. But load WordPerfect for DOS and type **/FR** and you'll see “/FR” appear in your document. In contrast, load Word for Windows, or almost any other Windows application, and pressing ALT-F O causes the File Open dialog box to appear, just as it does in Excel.

In Windows, finger knowledge is transportable. The user who learns to use one Windows application well should at least be able to navigate through any other Windows application. He or she will already know how to start and quit the program; open, save, and close files; move the cursor around the screen; copy and paste data; and how to get help when it is needed. This standardization of basic processes—standardization of finger knowledge—drastically simplifies the process of learning to use more than one major application. You'll find attempts to take advantage of this finger knowledge throughout all the application-building projects presented in Chapters 9 through 15.

Multiple-Application Processes

Finally, multitasking under Windows allows you to build and use real-world applications that cut across and incorporate the functions of several application programs. A time-billing system for a law firm, for instance, might incorporate a diverse set of functions, some of which are found in a word processor, some in a spreadsheet, some in a calendar program, and some in a database. If you so desire, you can build that application from scratch, figuring out how to program the various spreadsheet, database, word processing, and calendar functions that you need. If you're lucky, and start young enough, you might even live long enough to finish the job.

Under Windows, however, you can build the same application in a matter of days or weeks simply by writing a modicum of code that ties together

existing spreadsheet, database, calendar, and word processing programs into an integrated system—and it will function every bit as well as the mammoth programming project you’ve chosen to avoid. The DocMan project presented in Chapter 13 and the Windows Broker project in Chapter 14 are examples of linking existing programs to create new applications.

Designing for Multitasking

To design a great Windows application, you have to teach it to work and play well with others. The world of Windows is one of sharing: shared processing, shared facilities, shared interface. If you don’t want your application to have to share, don’t design it for Windows.

The high-level development tools used as examples in this book make it fairly easy to write a well-behaved Windows application that cooperates with others and reflects a concern for Windows’ standard user-interface protocols. But they also make it possible to ignore those concerns when necessary, because there are times when it is necessary to break the rules. The trick in designing good Windows applications is to break the rules as seldom as possible.

Shared Processing

Consider the following example of shared processing. In the old days, when programmers were designing a standard DOS application that included a time-consuming routine which would occupy the PC for several seconds or minutes, they could simply put a “Please Wait” message on the screen. Then, when the routine finished, everything unaffected by that routine would be just as it was when it began since nothing else could occur in the interim.

That’s not so in Windows. Even while an application is in the foreground receiving keystrokes and filling the entire screen, Windows and other Windows applications might be very active in the background: printing files, monitoring the keyboard and screen for commands regarding the application’s window, sending data back and forth with DDE, or communicating with a network or a remote device.

By the time an application finishes a lengthy routine, another application entirely might control the screen. For instance, the user might be busy typing data into a spreadsheet or word processor.

Because of the way Windows is structured, this functionality is not completely automatic. Instead, Windows hands control of the system in turn to each application that is running, and then waits for the application to return control. This process can occur many times per second—usually while an application is waiting for keystrokes or other user input. As application A waits for a keystroke, Windows hands control to B, which executes whatever

instructions it has waiting and then hands control back to Windows, which then repeats the process with application C or D, checking each time control is passed to ensure that the foreground application isn't waiting for control. This all takes place so fast that, in most cases, neither the applications nor the user is aware of it.

There are times, however, when you need to ensure that your application has sole control over the PC. For instance, some communications routines might be very time-critical—if your application doesn't respond to a command from the device with which it is communicating fast enough, the routine will fail. So you can't take the chance that application D in the background will hog the processor for a few too many milliseconds. For that reason, many of the development tools described in this book allow you to essentially put everything else on hold while a routine executes, by not handing control back to Windows.

In building Windows applications, you might be tempted to use this capability frequently. After all, the more your application has sole control of the system, the faster it will execute.

This is a temptation that you should avoid, simply because in most cases there is no justification for it. Usually applications in the background are not doing anything particularly processor-consuming. Instead, they're waiting for keystrokes too, or sending characters to the printer or sending data over a modem—processes which, in terms of a computer's sense of time, are incredibly slow and easy to maintain while still providing your application with what appears to be instantaneous response. So, at best, your application will generally only perform a few percentage points faster by hogging the system.

Meanwhile, you're losing most of the benefits of developing for Windows. Any background processes that are under way will be halted, and your users will be unable to switch to another application. The more proficient your users are with Windows, the more they'll resent your application as a result. That's too high a price to pay for a few percentage points of performance, except in the most critical situations.

Shared Facilities

Sharing the processor is an “under-the-covers” function—one that isn't immediately apparent to the user except when you don't do it. Your application's ability to share basic Windows facilities, however, is something that users will notice and appreciate immediately.

Windows 3.1 provides a set of standardized facilities that can be utilized by any Windows application. These include the Clipboard, which allows the user to cut, copy, and paste data; high-quality screen output; and facilities for creating links between applications using DDE or object linking and embedding.

In planning and building Windows applications you should look for ways to exploit these capabilities. Don't assume, for instance, that the only way that the user will want to enter data into your application is from the keyboard. Instead, put a Paste command on your application's menu so that the user can move data easily from another application into it. And include a Copy command there too, even if you don't perceive an immediate use for it. Your users will find uses for it.

Similarly, don't just dump ASCII text to the printer in the default font when you're building a facility for creating printed reports. There's no need for every application to have the page layout capabilities of PageMaker, but you should at least give the user some control over the appearance of the report by providing a choice of typefaces and sizes.

The benefits of providing Clipboard support and high-quality printing are immediately obvious. Support for DDE and object linking and embedding require a little more thought. These are complex protocols unlike anything found in the world of standard DOS applications. Their use can provide your application with a functional richness far beyond its apparent means. What appears to be a simple little forms-entry program, for instance, might through the use of DDE have all the analytic and functional resources of a product like Microsoft Excel at its beck and call. In planning your Windows applications, you should think long and hard about ways in which your application might benefit from these facilities. (Chapter 3, "Cut-and-Paste Programming," features an extended discussion of these considerations.)

Shared Interface

The most important form of sharing your application can offer the user is the shared Windows interface. Every time you implement a function in your application according to Windows' standard interface guidelines, you're making life easier for those users who know other Windows-based programs. And every time you choose to ignore those guidelines and get overly creative (or lazy) with your interface design, you're making life more difficult for those people.

Using the standardized Windows interface elements will also make your life easier as a developer of Windows applications. For instance, Windows 3.1 includes a dynamic link library that contains standard dialog boxes for functions such as opening and saving files. You can access those facilities from any of the many high-level languages that support dynamic link libraries—thus both eliminating the need for you to write dozens of lines of code to build your own dialog boxes and ensuring that the user will recognize and know how to use those dialog boxes whenever they appear.

There are other obvious ways to take advantage of the user's knowledge of the environment. For instance, say you're building the routine from which the user will exit your application. You could add a menu item to

your application and label it “Quit” or “Leave” or any number of other alternatives, and you could conceivably incorporate that menu item in any of the menus your application includes, but the standard way of leaving an application in Windows is by selecting the item labeled “Exit” from the File menu. Nearly every Windows application uses that convention, so nearly every Windows user knows that’s how one quits an application. Unless you’ve got a very good reason—and personal preference really isn’t good enough—follow the convention. It’s one less thing for your user to learn and one less decision that you have to make during the design process.

Originality Versus Standards

A lot of developers want to resist interface standardization because they fear it will make their application indistinguishable from any other Windows application. “Where’s the room for originality?” they ask, “Where’s the opportunity to make my application do something that others don’t?”

That’s a valid concern. Or rather, it would be if the kind of standardization we’re talking about actually affected any functions that would enable your application to distinguish itself. But it doesn’t. Instead, the brief guidelines discussed here, and the more extensive ones discussed in Chapter 7, simply govern those functions that are common to almost all applications.

Every application has to open, save, and close files and provide a way to exit the application. There’s nothing original about these functions, so it is to your benefit, and that of the users of your application, to provide standard ways of handling them. That enables you to devote your creativity to the original things that your application does—its core functions and the tasks you hope to achieve through it—rather than reinventing the wheel for something as simple as closing a file. Standardizing these common functions simply saves you time as a developer, and helps make sure that the big user-interface box that Dan Bricklin talked about is as friendly as possible.

C H A P T E R

2

Programming Basics

*The Elements of
Programming*

*Program Control
Elements*

Error Handling

Learning to Program

BEFORE GOING ANY FURTHER DOWN THE PATH OF WINDOWS APPLICATION development, let's examine some of the basic concepts and terminology intrinsic to any programming language. Since this material will be introductory in nature, experienced programmers may wish to skim this chapter, or skip it entirely.

It isn't possible in a single book, let alone a single chapter, to cover every aspect of every programming language used in Windows development tools—nor is it particularly necessary. Once you actually start to work with a high-level application-development tool, the manuals and tutorials that accompany it should be your first source of information about the language.

The purpose of this chapter is to provide enough general background information that the most absolute newcomer to programming can understand and gain from the material in subsequent chapters and to illuminate some of the processes that go into any application-development project. With the help of the information in this chapter, any reader should be able to understand the discussions of programming techniques in the application-building chapters (9 through 15), and understand how each of the commands in those projects is used.

To that purpose, this chapter will examine the basic concepts common to all programming languages and examine how the various components of a programming language are used to build applications.

In addition, this chapter will look at the process of *learning* to program, examining how one makes the leap from merely following another programmer's code to generating original code, and suggesting ways to speed that progression.

The Elements of Programming

Like a language spoken by human beings, a programming language is composed of a series of elements, each of which plays a particular role in the composition of a program. For instance, human languages are composed of nouns, verbs, and prepositions, along with other parts of speech. Similarly, computer programming languages are composed of commands, functions, and variables, along with other such *expressions*.

Moreover, just as in a human language you must construct sentences from multiple parts of speech in order to encapsulate a complete thought, so too in a programming language you must organize into *statements* the various commands and functions that you want the computer to perform in order for the computer to understand them. Some statements, like some sentences, consist of only a single word, whereas others may consist of dozens of words. The statement, composed of commands, functions, variables, and other programming language elements, is thus the basic building block of a computer program.

Statements can be very concise. For instance, the line

```
PRINT 52
```

is a complete statement in the BASIC language. It would instruct the computer to print the value 52 on the screen (or the currently selected output device).

However, computer language statements can also grow considerably more complex, as in the following example:

```
If ASC(RIGHT$(Editfile, 1)) < 33 Or ASC(RIGHT$(editfile,  
1)) > 122 THEN SET Editfile = LEFT$(Editfile, 1)
```

This statement, taken from a Visual BASIC program, is more complex, mixing a variety of language elements, including commands, functions, variables, constants, and expressions. You'll need to understand how each of these elements is used before the meaning of even a moderately complex statement like this one is clear.

Note that most of the programming examples and descriptions of commands and functions throughout this chapter are generic: They are intended primarily to explain the general concepts common to most high-level Windows development tools. Consequently, they don't reflect the exact syntax of any one programming language, and would therefore require some translation before they could be used in actual programs.

Commands

A command is just what you would think it is: a way of ordering the computer to perform a specified action.

Each executable statement in a computer program will generally include one command. For instance, the first example statement above contains the command Print. The second example contains the command Set, which is used to redefine the contents of variables (see the following section).

Most programming language commands accept *parameters*. These are used to make the point of the command more specific. When a mother wants her child to buy some milk, she doesn't just say "Go." Instead, she tells the child, "Go to the store and get some milk." Similarly, a program can't simply tell the computer to print. Unless the intent is to print a blank line, the program must include a parameter with the command that tells the computer what it is to print, as in:

```
PRINT 5
```

Depending on the command being used, the parameter may consist of a number, a text string, a variable, a subroutine name, or several other possibilities, all of which will be discussed in the following pages.

Most computer languages include a wide variety of types of commands. Some of these are common to many languages. Their syntax and implementation may vary from language to language, but their general purpose is achievable in a similar fashion in nearly every language. Other commands are very specific to the needs and purposes of a particular language. Thus it may be difficult to find equivalents to them in any other language.

Nearly every computer language includes a variety of commands used to control the flow of *program execution*, that is, which program statement's command will be executed next by the computer. Execution-control commands are needed because it is a rare situation indeed when you'll want a program to always execute its commands in exactly the same order every time the program is run. Certainly, any program that interacts with a user must be able to adapt the order in which it executes commands to the user's wishes.

Consider, for instance, the simple text editor called Notepad that accompanies Windows 3.1. When you run Notepad from the Program Manager, the program loads into memory and executes a series of instructions that first open up a blank file for editing, then draw Notepad's menus on the screen and put a flashing cursor at the point in the file where text will be inserted if you start typing. It then pauses to see what you'll do next. If you start typing, Notepad displays the characters you type and inserts them into the memory buffers it is maintaining for the unnamed text file. But, if you instead use the keyboard or mouse to activate Notepad's menus, it must carry out a different set of instructions. For example, if you select File Open it has to display a list of other files you can edit, whereas if you select Edit Search it has to open up a dialog box asking you to identify the text that you want to find. If you press F1 it has to display its help file.

Thus, once it has completed loading into memory and doing a little preliminary startup work, Notepad must have a block of code that employs logic on the order of the following:

```
Wait for something to happen
Then
If user is typing, execute the typing commands
If user opens File menu, execute the File menu commands
If user opens Edit menu, execute the Edit menu commands
If user presses F1, open the Help file
etc.
```

Execution control commands, therefore, are the traffic cops of programming. They ensure that the program's commands are executed in accordance with the wishes of the user and the programmer.

Most programming languages also include a set of *input/output* commands, which are used to read and write files, to display information on the

screen, and to send data to a printer. In Windows applications, some of these commands will make use of user-interface elements such as dialog boxes. For instance, most high-level Windows development tools have one-line commands for putting a simple yes-or-no message on screen, as in

```
MSGBOX ("Okay to delete all files on Drive C:? (Y/N) ")
```

Unfortunately, both the syntax of these commands and their specific capabilities differ from language to language. But their general capabilities are similar enough that, having learned how to read and write files or how to display information on the screen using one programming tool, the process of learning to do the same thing in another language will be much easier.

Finally, many languages have a set of commands that are wholly unique, without equivalents in other languages. This is most obvious in application macro languages, which offer commands geared to the specific capabilities of their underlying applications. For instance, Excel's macro language has a **RECALCULATE** command, reflecting the fact that Excel is at heart a spreadsheet. DynaComm's macro language, similarly, has a variety of commands that make sense only in a language designed for use with a communications program, including **WAIT QUIET**, which delays program execution until the communications line has been silent for a user-specified period, and **COPY BUFFER**, which copies the current terminal session buffer to the Windows Clipboard.

Variables

Variables are defined as temporary storage locations for information. By assigning a value to a variable such as **Delay%** or **Time\$**, you are telling the programming language to substitute that value whenever it encounters the variable name in the course of carrying out a command.

For instance, consider the following example of a simple two-line program:

```
X%=5  
PRINT X%
```

The first line tells the computer that the value 5 should be substituted for the variable named **X%** whenever it appears in a command statement. The second line tells the computer to print **X%**. Following the instruction from the first line, the computer will respond to line two by printing "5".

Most programming languages support the following basic types of variables, which are distinguished from one another by the kind of data they can contain:

- String variables
- Integer variables
- Real variables

String Variables

The first type, *string* variables, can be used to contain any assortment of alphanumeric characters, plus nonprintable control characters. For instance, if you were a *My Fair Lady* fan you could assign the sentence “The rain in Spain stays mainly in the plain” to a variable called Rain\$. Then, whenever your program encountered Rain\$ in the course of carrying out a command, it would substitute “The rain in Spain stays mainly in the plain.”

Thus, the command

```
PRINT Rain$
```

would print Eliza Doolittle’s famous tongue twister.

In many programming languages, a string variable is designated by a dollar sign, either at the beginning of its name (\$Rain) or at the end (Rain\$), depending upon the variable-naming convention used by the language.

Integer Variables

The second standard type of variable is an *integer* variable. As its name suggests, this variable is used for storing integers (nonfractional numbers). The numbers 6, 5, 302, and -256 are all valid integers. Frequently, integer variables are further limited to handling values ranging from -32,768 to 32,768, although some languages also offer a *long-integer* form of variable that extends the range of valid integers to several billion numbers (from -2,147,483,648 to 2,147,483,647).

Integer variables are often designated by a percentage sign, either at the beginning (%X) or the end (X%) of the variable name, again depending upon the language’s convention for naming variables.

The primary reasons for using integer variables are speed and economy of memory usage. Integer variables generally occupy only two to four bytes of memory (as opposed to the four to ten bytes occupied by noninteger numeric variables), making it very easy and fast for the computer to look up their value whenever it needs to, and helping to conserve memory.

Real Variables

The final standard variable type is the *real* variable, which can be used to store both nonfractional and fractional values. Thus, 3.1415927, 6.3, and -99.9394949 are all valid values for a real variable. Real variables occupy four to ten bytes of RAM. They should be used whenever fractional numbers will be assigned to the variable.

Some languages allow you to determine the precision and memory usage of a real variable. For instance, Visual BASIC offers both *single*- and *double-precision* real variables, which occupy four and eight bytes of memory, respectively. Single-precision variables can hold values ranging from $\pm 1.5E-45$ to $\pm 3.37E+38$ (with 7 to 8 digits of precision), whereas

double-precision variables can hold values ranging from $\pm 5.0E-324$ to $\pm 1.67E+308$ (with 15- to 16-digit precision).

There is no standard naming convention for real variables. Some languages use a number sign (#) preceding or following the variable name, others an exclamation point (!), and others no indicator at all.

Initializing Variables

Different high-level programming languages handle the *initialization* of variables (the setting of initial values for them) in different ways. Some require you to list all the variables your program will use at the beginning of the program, stating the type of each variable, as follows:

```
DEF Midwords AS STRING
DEF Wordlength AS INTEGER
```

and so on. The exact syntax for making these declarations differs from language to language.

Other languages allow you to make implicit declarations of variables, simply by using them and assigning them the correct suffix or prefix. For instance, you could issue a simple command such as

```
Midwords$="Early one morning"
```

to initialize a new string variable called Midwords\$, and assign the string "Early one morning" to it.

You can even use the contents of other variables to define a new variable, for instance

```
X%=4
Y%=5
Z%=X%+Y%
```

would define Z% as the sum of X% and Y%, or 9. Once defined in this manner, Z% would remain equal to 9 no matter how the values of X% and Y% change, until you explicitly redefine Z% again.

So, if the next lines in the program read

```
X%=8
Y%=14
```

Z% would remain equal to 9. However, if you followed those with the line

```
Z%=X%+Y%
```

Z% would be redefined to equal the sum of the current values of X% and Y%, or 22.

Regardless of how your programming language initializes variables, it is important that you always assign the correct form of data to them. Never assign string data to a real variable, or a real number to an integer variable. Doing so will cause an error that will probably halt the compiling or execution of your program. This kind of error is commonly referred to as a *type mismatch* error.

Why Use Variables?

There are two basic reasons for using a variable in writing a program rather than the actual contents of the variable. The first is to conserve space. A string variable, for instance, might be hundreds of characters long. If you're going to refer to that string dozens of times in your program, it makes much more sense to use a variable name—which is stored in the program as a four-byte address—than it does to use the entire string each time.

The other, and actually more frequent reason for using variables in lieu of actual values, is that at the time you write the program, you often don't know what the actual value will be. For instance, let's say you have a simple program that uses a dialog box to ask the user to enter an integer between 1 and 10. Then, once the user has entered the number, the program multiplies it by 10 and prints the result.

That's a simple enough program, but at the time you write it you have no idea which of ten possible values the user will enter. So how do you tell your program to multiply the number that is entered and then display the result? With variables, it's easy. You simply assign the number the user entered to an integer variable—call it `User%`—and then manipulate it with a couple of lines of code, something like this:

```
New_num%=User%*10  
Print New_num%
```

If you didn't have access to variables in which to store the number the user entered, however, it would be difficult, if not impossible, to carry out these operations on it.

Often, too, your program will contain commands that set variables to values that you could not know at the time that you wrote the program. For instance, a program that repeatedly prints the current time and compares it to a predefined starting time would continually be assigning new values to the variable in which the current time is stored.

In addition, the use of variables allows you to make use of a variety of program control devices, one of which is called a *loop*. A loop is a way of using the same programming code repeatedly, rather than repeating the code in your program. For instance, if you wanted to print a list of the numbers 1 to 1,000, you could go about the problem in two ways. You could either issue

the Print command 1,000 times, followed each time by the actual number to be printed:

```
PRINT 1
PRINT 2
PRINT 3
```

and so on, or you could use a variable to represent the number to be printed and simply increment the variable's value each time the Print statement is executed. To do so, you'll use a FOR-NEXT loop, as follows:

```
FOR X%=1 TO 1000
PRINT X%
NEXT X%
```

This simple device, (which saves you 997 lines of program code!) works like this: The statement FOR X%=1 TO 1000 tells the program to repeat the lines between that statement and the statement NEXT X% once for each integer, starting with 1 and ending with 1,000. Moreover, each time through the loop the value of X% is reset to the number of the current repetition, so the first time through X%=1, the second time X%=2, the third time X%=3, and so on. (Loops are covered in more detail later in the chapter.)

Functions

Functions are executable statements that return a value to your program by modifying the contents of a variable. For instance, the statement

```
Sub$=MIDSTRING$("This morning", 6,4)
```

utilizes the MIDSTRING\$ function, a common function in BASIC and BASIC-like languages. MIDSTRING\$ accepts three parameters. These parameters, enclosed in parentheses and separated by commas, consist of: a string to analyze, a starting position, and a length, respectively. The first MIDSTRING\$ parameter must be in the form of either a string or a string variable. The second and third parameters must be in the form of either numbers or integer or real variables. Thus, the same command could be written as

```
Sub$=MIDSTRING$(Morning$,Start%,Length%)
```

assuming that you or your program had already assigned the string "This morning" to Morning\$ and the values 6 to Start% and 4 to Length%.

The MIDSTRING\$ function is used to extract a part of the original string, starting at the starting position you specify and continuing for the number of characters specified in the length parameter. Thus, MIDSTRING\$("This morning", 6,4) would start at the sixth character of "This

morning” and continue on for four characters. The result of the function, therefore, would be “morn”.

In the example above, the MIDSTRING\$ function returns the value “morn” to the program, which then assigns it to the specified variable. In this case, the variable to be modified is called Sub\$, so after this statement has been executed, the variable Sub\$ will contain the string “morn”.

Predefined Functions

Functions perform most of the actual data-crunching work that goes on in a program. When you want to obtain an average of five numbers, determine the current date, or the cosine of an angle, or convert a real number to an integer, you’ll use a function. As you might expect, every programming language offers a different set of functions. The number and power of the functions offered by a programming language goes a long way toward determining its power and usefulness.

Like the functions in a spreadsheet program, those in a programming language can be classified into a number of broad categories. These include *mathematical* functions, *string* functions, and *type-conversion* functions.

Math Functions

Common mathematical functions include functions that perform binary arithmetic (BAND, BOR, BNOT, for example), and those that round off real numbers to a specified number of places, such as ROUND. For instance,

```
X=ROUND(3.25233,2)
PRINT X
```

would round off 3.25233 to 3.25, assign that value to *X*, and print it.

String Functions

The string functions you’ll encounter in many programming languages include functions such as MIDSTRING\$, which is used to extract part of an existing string; LEN, used to determine the length of a string; and UPPER\$, which converts all the characters in a string to uppercase. Thus

```
$First="The early bird gets the worm."
$Second=MIDSTRING$($First, 4,5)
Length_var=LEN($Second)
$Third=UPPER$($Second)
PRINT $Second
PRINT Length_var
PRINT $Third
```


would print the following:

```
early      The characters extracted from $first with the MIDSTRING$(  
           function  
5          The length of the extracted character string  
EARLY     The extracted character string after conversion to uppercase
```

Type-Conversion Functions

These functions are used to convert one type of variable to another. Among the more common variants of these are `STR$`, which converts a number to a string; `INT`, which converts a real number to an integer; and `VAL`, which converts a string to a number. Thus,

```
A$=STR$(4-2) would assign the character 2 to A$.  
B%=INT(2.535353) would assign the value 2 to B%.  
C=VAL("6.2") would assign the value 6.2 to C.
```

The exact syntax and usage of these commands will, of course, differ from language to language.

Other Function Types

In many languages you'll also encounter functions that perform statistical manipulations on groups of numbers (finding the average, maximum, or minimum value in a group, for instance) or that perform file-related duties (such as finding the length of a file or determining whether a file exists).

User-Defined Functions

The presence of a function geared specifically to the task you're trying to accomplish certainly speeds the development process, but the lack of one doesn't mean that you can't achieve the results that you want. For instance, imagine that you want your application to determine whether the file `MYDATA.DAT` exists in the `\DATA` subdirectory. If the language you're using includes a function to determine a file's existence, the task is simple. You'll simply include a line such as

```
YesNo%=EXISTS("\DATA\MYDATA.DAT")
```

If the `EXISTS` function returns a 1 when it finds a file and a 0 when it doesn't, the value of `YesNo%` would be 1 if the file existed and 0 if it wasn't found.

But what if the language with which you're working, like many, lacks a function that performs this task? You can build up the function yourself out of the functions that are available in the language.

For instance, many languages include an `ERROR` function, which can be used to determine if an error occurs while executing the program. The implementation of this function varies from language to language, but for

the purposes of this discussion let's take a case where the ERROR function reports whether an error occurred in executing the previous statement in the program. With that in mind, one could duplicate the EXISTS function with a couple of lines of code like these:

```
FILEOPEN( "\DATA\MYDATA.DAT" )  
%YesNo=ERROR( )
```

If an error occurred, indicating that the file wasn't found, %YesNo would be equal to a number between 1 and 255 that would correspond to the BASIC error code for that error. Otherwise it would be equal to 0.

Some languages allow you to define new functions and assign names to them, so that these *user-defined functions* can then be used later in your program simply by referring to the names you've assigned them. For instance, you might define an EXISTS function for a language that lacks one, and then call it simply by using its name and passing it the name of the file to check, as in:

```
YesNo%=EXISTS( "\DATA\MYDATA\DAT" )
```

Windows-Specific Functions

High-Level Windows programming languages frequently include an additional set of functions related to the Windows environment. These may include functions for manipulating the Clipboard or an application's window and for interacting with other applications.

For instance, WinBatch includes a number of clipboard-oriented functions, including CLIPGET, CLIPPUT, and CLIPAPPEND. CLIPGET is used to obtain the current contents of the Clipboard and assign them to a string. CLIPPUT copies a string to the Clipboard. CLIPAPPEND is used to *append* the contents of a string to the current contents of the Clipboard.

Similarly, various high-level Windows programming languages include commands to determine the size of an application's window and whether it has been iconized, to reposition or resize any window, and to send key-strokes to another application.

Expressions

Expressions are essentially formulas that use functions and operators to manipulate the contents of variables. For instance,

```
5+2
```

is a simple expression that adds the values 5 and 2 using the addition operator, +.

There are three basic varieties of expressions you'll find in most programming languages:

- Numeric expressions
- String expressions
- Boolean expressions

Numeric Expressions

Numeric expressions, as their name implies, operate on numeric variables—often using *numeric operators*. Table 2.1 demonstrates the usage of common numeric operators.

Table 2.1 Common Numeric Operators

Operator	Usage	Description
+	5+2	Adds two numbers
-	5-2	Subtracts the second number from the first
*	5*2	Multiplies the numbers on each side
/	5/2	Divides the first number by the second
MOD	5 MOD 2	Rounds the two numbers to integers, then divides the first number by the second and returns the remainder

String Expressions

String expressions operate on string variables, chiefly by applying functions such as those described in “String Functions,” above. For instance,

```
MIDSTRING$(B$,12,18)
```

is a string expression.

In addition to providing functions for converting strings to upper- and lowercase, obtaining a specified number of characters from a specified position within a string, and for converting strings into values, most languages also allow you to combine strings through a process known as *concatenation*.

In Visual BASIC, for instance, you could use code such as the following to concatenate two strings:

```
A$="It's raining "
```

```
B$="cats and dogs."  
C$=A$+B$
```

When this code is executed, C\$ would be set to read “It’s raining cats and dogs.”

The operator used for string concatenation differs from language to language. In DynaComm’s macro language, for instance, the vertical bar (|) is used in place of the + sign to concatenate strings, as follows:

```
C$=A$|B$
```

The result of the operation is the same, however. After execution of that statement, C\$ would contain the string “It’s raining cats and dogs.”

Boolean Expressions

Since string expressions are used to manipulate strings, and numeric expressions are used to manipulate numbers, you might think that Boolean expressions are used to manipulate “Booleans”—whatever those are. Actually, however, Boolean expressions are those that apply Boolean logic, or Boolean algebra, to strings and values. Named for the English mathematician George Boole, Boolean algebra is used to determine the truth or falseness of an expression, rather than to calculate values.

The Boolean operators are shown in Table 2.2. All Boolean expressions evaluate to a result of either TRUE (which may be represented by a value of TRUE, 1, or -1, depending on the language being used) or FALSE (which may be represented by a value of either FALSE or 0).

For instance, executing the Boolean expression

```
PRINT 5=2
```

would print either 0 or FALSE, whereas executing

```
PRINT 2=(4/2)
```

would print either -1, 1, or TRUE depending on the language being used.

Boolean expressions are generally used to compare variables. The most common Boolean operators are =, <>, <, and >. These operators can be applied to both numeric variables and string variables. For instance, the expression

```
PRINT "cat"<>"dog"
```

would evaluate as true since the two strings are not identical, and would thus print either -1, 1, or TRUE.

It’s evident how you might use greater than and less than comparisons with numeric variables, but you can also do so with string variables. In that case, the program compares the ASCII value of the first character in each

Table 2.2 Boolean Operators

Operator	Usage	Description
=	IF X=Y	Returns TRUE if the two values are equal; otherwise returns FALSE
>	IF X>Y	Returns TRUE if the first value is greater than the second; otherwise returns FALSE
<	IF X<Y	Returns TRUE if the first value is less than the second; otherwise returns FALSE
<>	IF X<>Y	Returns TRUE if the two values are not equal; otherwise returns FALSE
AND	X<Y AND Y>Z	Returns TRUE if both statements are true; otherwise returns FALSE
OR	X<Y OR Z<Y	Returns TRUE if either statement is true; otherwise returns FALSE
NOT	Y=Z	Returns the reverse of what would normally be returned: if Y were equal to Z, NOT would return FALSE; if Y didn't equal Z, NOT would return TRUE

string to determine which is “greater” or “less” than the other (and the ASCII values of the second characters if the first characters are equal, and so on). Thus, the expression

```
PRINT "cat">"dog"
```

would evaluate as FALSE, since the ASCII value of the letter “c” (99) is less than that of the letter “d” (100). (Cat lovers, however, may disagree with this result.)

There are three additional Boolean operators of note: AND, OR, and NOT. The first two are used to derive a single Boolean result from multiple Boolean evaluations, whereas the third is used to reverse the truth or falsity of an expression.

The AND operator requires that both the expressions it links be true in order for the Boolean result to be TRUE. By contrast, the OR operator requires that only one expression be true. Thus if X% has the value 6

```
X% <30 AND X% >3
```

would evaluate to TRUE, since both expressions are true. But

```
X% <30 AND X% <3
```


would evaluate to FALSE, since one of the expressions is false. However, if you changed the AND in the second statement to an OR

```
X% <30 OR X% <3
```

the result would be TRUE, since one of the two expressions is true.

```
X% >30 OR X% <3
```

would, however, evaluate to FALSE, since neither expression is true.

The NOT operator is simply used to reverse the result of a BOOLEAN expression. Therefore, since the expression

```
25>100
```

evaluates to FALSE, the expression

```
NOT (25>100)
```

would evaluate to TRUE.

Comments

Comments are nonexecutable statements that are placed in a program to explain the purpose or logic of the surrounding executable statements. For instance, in the following example, the line preceded by a single quotation mark is a comment:

```
'Print original value of string, then convert string  
'to uppercase and print it again  
PRINT X$  
X$=UPPER$(X$)  
PRINT X$
```

Different languages use different mechanisms to identify comments. In some languages, comments are preceded by a single quotation mark, as above. In others, a semicolon (;) or the abbreviation REM is used to identify a comment.

The presence or absence of comments in program code does not alter the performance of your programs at all, but it does greatly add to their readability and decipherability. As you write a complex program, the logic of the program's various routines may seem self-evident. However, if you go back to that program six months later, without the benefit of explanatory comments, you may find it all but impossible to determine even the general purpose of a routine, let alone to understand all of its elements. And anyone else who examines your program code will almost assuredly find it incomprehensible.

Liberal use of comments can greatly alleviate this problem. It doesn't take long to comment a program, and you'll thank yourself later for doing so.

Arrays

Arrays provide a way to refer to a series of variables with a single name, using an index number to differentiate between members of the series. This enables you to group variables in an organized fashion.

For instance, consider a series of variables containing the names of various animals. You could define the series with a set of statements, such as

```
Dog$="Dog"  
Cat$="Cat"  
Elephant$="Elephant"  
Monkey$="Monkey"
```

and so on. But this method is inefficient because it requires you to reenter each variable name every time you want to deal with the entire series of animals.

Using an array, however, you could deal with the same variables as follows:

```
Animal$(1)="Dog"  
Animal$(2)="Cat"  
Animal$(3)="Elephant"  
Animal$(4)="Monkey"
```

This approach enables you to deal with the elements of the series either individually or as a group. Thus, to print just "Dog", you would simply issue the command

```
PRINT Animal$(1)
```

But if you wanted to print the name of every animal in the array, you could do so quite easily by using a simple loop:

```
FOR X=1 TO 4  
PRINT Animal$(X)  
NEXT X
```

Multidimensional Arrays

The preceding example was of an array with only one dimension—that is, only one set of index numbers. But you can also construct arrays with two or more dimensions. For instance, imagine you wanted to expand your animal array to include the names of different breeds of each animal. You could do so with a two-dimensional array. The first dimension would hold

the species name, and the second would hold the name of the individual breed, for example,

```
Animal$(1,1)="Basset hound"  
Animal$(1,2)="Poodle"  
Animal$(1,3)="Great Dane"  
Animal$(2,1)="Siamese"  
Animal$(2,2)="Persian"
```

and so on. All dogs would have 1 as their first index number, whereas all cats would have 2 in the first index location.

Program Control Elements

In addition to the fundamental building blocks discussed above, such as variables and functions, computer programs incorporate a variety of control elements, including the aforementioned loops, and numerous ways to determine the order and conditions under which the program statements are executed. The most fundamental of these are decision-making structures such as the IF-THEN statement.

Decision Making

The strength of most good computer programs is their ability to make decisions about what to do next, based on user input or other data such as messages from other applications or from a modem.

The primary method that programs use to make decisions is the IF-THEN statement. This takes the form of

```
IF it is raining THEN wear your galoshes
```

or, in “computerese”

```
IF Weather%=Rain% THEN Galoshes%=1
```

The ELSE command is an adjunct to the standard IF-THEN statement. It is used to provide an alternate choice:

```
IF Weather%=Rain% THEN Galoshes%=1, ELSE Sneakers%=1
```

Some high-level languages also allow you to *chain* a series of IF-THEN-ELSE statements together, as follows:

```
IF Weather%=Rain%  
  THEN Galoshes%=1  
ELSEIF Weather%=Sunny%  
  THEN Sneakers%=1
```

```
ELSEIF Weather%=Snow%
  THEN Snowshoes%=1
ELSE Barefoot%=1
END IF
```

The statement `ELSE Barefoot%=1` tells the program that if none of the other conditions are true, the user should go barefoot. The final `END IF` statement tells the program that the block of `IF` statements has come to an end.

Select Case Statement

Another way of evaluating variables in order to make a decision is the `SELECT CASE` statement, which is supported by some, but not all, high-level programming languages.

A `SELECT CASE` statement tests an expression and selects a command to perform based on the result. This process is more easily shown than described. For instance, the `IF-THEN-ELSE` routine described above could be rewritten as the following `SELECT CASE` statement:

```
SELECT CASE Weather%
  CASE Rain%
    Galoshes%=1
  CASE Snow%
    Snowshoes%=1
  CASE Sunny%
    Sneakers%=1
  CASE ELSE
    Barefoot%=1
END SELECT
```

This routine would yield exactly the same results as the more complex `IF-THEN-ELSE` routine. One advantage this has is that the `SELECT CASE` statement is easier to follow because it is immediately apparent that all the decision possibilities in the loop depend on the contents of the variable `Weather%`. In addition, depending on how the `SELECT CASE` function has been implemented in a language, this version of the routine may execute more quickly than the complex `IF-THEN-ELSE` version.

Loops

The concept of loops was briefly touched on earlier. There the most simple form of this device, the `FOR-NEXT` loop, was described. As stated earlier, loops enable you to repeat a series of instructions without having to rewrite their code for every repetition.

One advantage of loops is that they enable you to use the computer's ability to simply grind away at a problem until it arrives at the correct answer. For instance, say you wanted to determine the square root of 467,856, and all that you knew was that the result was an integer between 1 and 1,000. If you were trying to find the answer on paper, you'd know right off that you could skip low numbers such as 1, 10, and even 100, because without even multiplying them out you know that their squares are not anywhere close to 467,856.

Computers don't have that kind of common sense, but they compensate for it by being blindingly fast. You could try to program the computer to follow the same reasoning that you would apply to solve the problem, but doing so would be a complex programming process because the computer doesn't know 1×1 doesn't equal 467,856 until it does the calculation. The alternative is to use a loop that simply muscled its way through, squaring every number between 1 and 1,000 until it gets the correct answer. This might add a second or two to the processing time for the loop, but it could save you hours of programming time.

Here's the loop:

```
FOR X=1 TO 1000
IF X*X=467856 THEN Answer=X
NEXT X
```

The problem with this method is that it is doubly inefficient. In addition to testing values that are absurdly low, it also continues to test even after it has arrived at the correct answer (684), because there is no mechanism for ending the loop once it has arrived at the correct answer.

Some languages provide a back door to FOR-NEXT loops in the form of a command called EXIT FOR. Using that command, you'd rewrite the loop as follows:

```
FOR X=1 TO 1000
IF X*X=467856 THEN EXIT FOR
NEXT X
Answer=X
```

WHILE-WEND Loops

There are also a variety of other looping methods that can be used to solve the above problem. One is the WHILE-WEND loop, which continues to execute a loop as long as the specified condition is true. Using WHILE-WEND, the loop above could be rewritten as follows:

```
X=1
WHILE X*X<467856
```



```

INCREMENT X
WEND

```

This loop uses the INCREMENT command to add 1 to the current value of X each time the WHILE loop is executed, and continues to execute the loop until X^2 is equal to the target number. It automatically stops execution of the loop when the target figure is reached, without the awkwardness of an EXIT FOR command.

As stated, a WHILE-WEND loop will continue to execute as long as its initial condition is true. If the condition is false the first time it is tested, none of the instructions in the loop will be executed.

In general, FOR-NEXT loops are more useful when you know in advance how many times you want the loop to repeat, whereas WHILE-WEND loops are more useful when the number of repetitions is not known in advance.

DO-LOOP

Another variety of loop is the DO-LOOP, which has many of the advantages of the WHILE-WEND loop, but is even more flexible.

The DO-LOOP can be set to either repeat itself as long as its initial condition is true (like a WHILE-WEND loop) or until that condition becomes true. Moreover, the condition upon which the loop operates can, in many languages, be specified at either the beginning or the end of the loop. Placing the condition at the end of the loop ensures that the loop will execute at least once, no matter whether the condition is true or false.

The square root problem would be rewritten using a DO-LOOP as follows:

```

DO UNTIL X*X=467856
INCREMENT X
LOOP

```

As written, this will continue to increment X until X squared is equal to 467,856. But if the initial value of X is 684, the condition will be satisfied immediately, and thus the X will never be incremented.

The loop could also be written like this:

```

DO
INCREMENT X
LOOP UNTIL X*X=467856

```

Since the condition is placed at the end of the loop, this ensures that X will be incremented at least once.

Endless Loops

Did you ever work with a buggy program that would start flashing a busy signal at you and never stop? Of course you have, every PC user has had that misfortune. You just sit there watching the busy indicator flashing and wondering why the program is acting the way it is.

One strong possibility is that the program has gone into an endless loop. That is, it is executing a loop that has a condition for exiting that can never be reached. How is such a thing possible? Consider the DO-LOOP just described:

```
DO
  INCREMENT X
LOOP UNTIL X*X=467856
```

This loop will work well under most conditions. However, it rests upon one big assumption: that the initial value of X will be less than 684. If it isn't, however, X*X will always be bigger than 467,856, no matter how many times the program increments X. As long as the program is written in such a way as to ensure that X is always less than or equal to 684 at the beginning of this loop, it will execute flawlessly. But if the value of X is ever greater than 684 at the beginning of the loop, the program will never find the answer it is seeking.

The bottom line: Loops are invaluable devices, but you should use them carefully in order to ensure that the conditions they are dependent on are realistic.

Input/Output

There is a tremendous variety in the file read/write capabilities of high-level Windows development tools. Some, such as application macro languages, have a native file format that they rely on. Some are also able to read and write files created by such popular programs as dBASE, Lotus 1-2-3, and Excel. But in addition, nearly every high-level language-development tool for Windows makes it possible for programs written using the tool to create, read, and write *text-format* files.

Text-format files are the simplest and most universal file type. In these files, data is stored in the form of plain ASCII characters, rather than being encoded into a binary file. After being displayed or printed, text files can be read by humans, and they can be exchanged with nearly any kind of computer system. Binary files, in contrast, can only be read by a computer and can only be deciphered by a program that knows the method that was used to encode them.

Text-format files can generally be categorized as either *sequential-access* or *random-access* files. With the former, data is accessed by reading through the entire file until the desired data is obtained, whereas with the latter, data

is organized into records—a program can read a specified record directly without having to search sequentially from the start of the file.

Sequential-access files are generally more useful for storing data records that will all be used at once—such as a stored list of setup options that is read as a program initializes. Random-access files, by contrast, are more useful for working with data that will be accessed on a record-by-record basis. Random access offers less flexibility in terms of individual record structure, but it allows individual records to be rewritten, updated, or deleted without rewriting the entire file.

Generally a file's type is specified at the time it is created. In Visual BASIC, for instance, you could open for random access a file called TESTFILE.DAT with 256-character records using the following command:

```
OPEN "TESTFILE.DAT" AS 1 FOR RANDOM LEN=256
```

The phrase "AS 1" is used to assign a file number to the file; it is required by Visual BASIC.

Sequential files in Visual BASIC must be opened for either input or output, not for both at once. To open the file for sequential output the command would be:

```
OPEN "TESTFILE.DAT" AS 1 FOR OUTPUT
```

Named Subroutines

Earlier in this chapter I talked about execution-control commands. These commands typically work in conjunction with *named subroutines*—groups of program statements that have been assigned a name.

The simplest form of named subroutine is the GOSUB statement, found in BASIC and some other languages. It is used to divert program execution to a named subroutine. The last statement in the named subroutine should be RETURN, which diverts program execution back to the original routine at the statement immediately following the GOSUB routine, for example:

```
X=4  
GOSUB PRINT_IT  
X=8  
GOSUB PRINT_IT  
...  
  
*PRINT_IT  
PRINT X  
RETURN
```


The asterisk preceding PRINT_IT indicates that PRINT_IT is the name of a subroutine. As the program executes, the PRINT_IT routine is executed each time a GOSUB PRINT_IT statement occurs.

A more sophisticated form of accessing named subroutines is the PERFORM statement, which, in many languages, allows you to pass a variable to the subroutine as you call it.

For example, imagine that you wanted the PRINT_IT routine to print not the value you passed to it, but rather its square. You could use the PERFORM command as follows to implement this:

```
X=4
PERFORM PRINT_IT(X)
X=8
PERFORM PRINT_IT(X)
...

*PRINT_IT (Y)
Y=Y*Y
PRINT Y
RETURN
```

The advantage of this construction is that although you're passing the contents of the variable X to the PRINT_IT subroutine, X itself is unchanged by the subroutine. Instead, it assigns the value that has been passed to it to a variable called Y. It then squares Y and prints the result. Variable X remains unchanged.

Variable Scoping Rules

This introduces a touchy subject called *variable scoping*—one that is made more difficult by the fact that the rules governing it vary greatly from one programming language to the next. In general, however, the point of variable scoping is that all variables are not created equal. Instead, each variable is assigned a *scope* at the time it is created.

Some variables are considered *global* in scope, in which case they can be used anywhere in a program, and their contents can be changed at any time. Other variables are considered *local*—they exist and are usable only in the subroutine that created them.

Other variables follow a downward hierarchy. They are valid in the routine that created them and any routines that routine calls, but are not valid in the routine that called the routine in which they were created.

Why do languages enforce variable scoping rules? There are two good reasons. One is that variables eat up a lot of memory, and if all variables were considered global (meaning that your program had to keep track of each variable that you create for as long as the program runs), you'd run out of memory much faster.

The second reason for having variable scoping rules is that they help make programs more simple and more understandable by eliminating the need to keep track of dozens of variables that may only have been used once. Consider, for example, the variable `Y` in the second `PRINT_IT` subroutine above. It was created only so that we could square it and then print it. Once it has been printed, the program has no further use for the variable, and hence it is of no consequence if the variable ceases to exist as soon as the `RETURN` statement in the `PRINT_IT` subroutine is executed.

In fact, you could improve the readability of the program by renaming the variable `Y` as `Temp`, and then using the name `Temp` whenever you created a local variable that was doomed to a quick extinction, for instance:

```
*PRINT_IT (Temp)
Temp=Temp*Temp
PRINT Temp
RETURN
```

```
*HALVE_IT (Temp)
Temp=Temp/2
PRINT Temp
RETURN
```

It would take only a few seconds of perusing a program containing several subroutines written this way to realize that “`Temp`” was always used to name a local variable that was to be discarded almost as soon as it was created.

Error Handling

Error handling might be better called the prevention and cure of *runtime errors* (errors that arise at the time your program is executed).

There are two basic causes of runtime errors: The first is that your program doesn't work the way you think it does; the second is that the user doesn't work with your program the way you think he or she will.

As an example of the first type of error, consider one of the `DO-LOOPS` listed above, where the test condition was placed at the beginning of the loop:

```
DO UNTIL X*X=467856
INCREMENT X
LOOP
```

Now let's suppose that for some reason you were really counting on `X` always being incremented—or perhaps that another command that you've added to the loop would always be executed—whenever the program came to that loop. In fact, you're counting on it so much that your program won't

work if it doesn't happen. Well, that's a problem because, as demonstrated above, if the initial value of X is 684, the loop will not execute, and thus X will not be incremented. In other words, your program won't work the way you thought it would.

This kind of problem is easy to overlook at design time and difficult to track down when it occurs. But if you want your program to be bulletproof, you have to anticipate it and find a solution to it—either by ensuring that X is never equal to 684 when the loop is called or by adding a statement somewhere else in your program that decrements X if it happens to be equal to 684 prior to calling the loop.

User Errors

The second major cause of runtime errors occurs when users don't respond the way you thought they would to your program. For instance, let's say you've got a routine that requests that the user input the current date, and then assigns the date the user enters to a string variable called Date\$. If the user enters **08/24/56**, you could *parse* (break up) the date into separate strings for the month, day, and year as follows:

```
Month$=SUBSTR$(Date$,1,2)
Day$=SUBSTR$(Date$,4,2)
Year$=SUBSTR$(Date$,7,2)
```

Those statements will work quite well, as long as the user responds exactly as you expect. But what if the user types **8/24/56**, **August 24, 1956**, or **24.8.56**? Any of those variants will result in confusion, at best, since your Month\$ variable would then be assigned to hold "8/", or "Au", or "24", and your Day\$ and Year\$ variables become equally mis-set.

The thing that makes user-introduced errors so difficult to prevent is that you might test that routine yourself a dozen times and never have it occur to you to enter anything at the prompt other than **08/24/56**. Similarly, you might test a dialog box hundreds of times, and never once hit the OK button before filling out the dialog boxes' edit fields (because it's obvious to you that no one would ever do that). But if you don't plan for it, perhaps by having a routine that immediately redraws the dialog box if the necessary edit fields aren't completed, sooner or later someone will do the unthinkable. He or she will hit the button that no one would ever hit, or enter a date in a format that no one would ever use, and crash your program in the process.

The only way to avoid these errors is to plan for every possible eventuality, and then when you think you've got every possible base covered, test your program on some real users. They're sure to find plenty of errors you would never uncover yourself, no matter how long you tested the program.

Learning to Program

That concludes this chapter's tour of the basic concepts of programming. The information covered here should enable you to understand the discussions of the programming techniques and the example programs in the chapters that follow, and it might even give you a head start when it's time for you to master one or more Windows high-level programming languages.

It's one thing to be able to follow along with the examples in this book, or to complete the tutorial chapters in some programming language's user manual. It's another thing entirely to know how to program, and that is something that can't be taught.

Oh, I know there are thousands of students sitting right now in programming courses learning the ins and outs of relational database theory and indexed sequential-access files and the fastest sort routines for all occasions—and certainly those are all valuable pieces of information. But just because someone is the master of those concepts doesn't mean that he or she is a real programmer.

Learning to program involves two tightly intertwined processes: You must learn how to see what is possible, and you must learn how to turn the possible into the real.

Before you can write a single line of code, you have to decide what it is that you want to achieve with your program. In order to do that, you've got to thoroughly understand both the capabilities of the Windows 3.1 environment and those of the development tools at your disposal. Then you have to be able to see where those capabilities and the possibilities they offer intersect with your needs as a computer user. In other words, the first part of programming, the conceptual part, is a creative process. You have to make a cognitive leap from the reality of the current capabilities of Windows and your Windows applications to the possibilities that you might achieve through careful wielding of high-level language-development tools.

Once you've made that leap, you have to use your knowledge of these development tools to actually write the program code that will turn the possible into the real. In the process of doing so, you'll undoubtedly encounter complications that need solving, and see new possibilities that need exploring. But if your fundamental premise for the application is sound, once you've gotten to the code-writing stage you should be able to turn your ideas into reality.

So how do you learn to recognize what is possible? And once that's achieved, how do you learn to use high-level language-development tools well enough to turn raw ideas—no matter how conceptually brilliant—into reliable, useful programs?

The answer is *experimentation*. You can study all the example programs in the world, and type every line of sample code you can find into your PC,

but unless you're willing to experiment with programming you'll never be more than a typist inputting someone else's ideas.

As you work through the examples in this book, or the examples in the manuals for any development tools you might happen to have, don't settle for simply typing in each example as it comes about. Change the values of variables, change the names assigned to buttons, change the size of other user-interface elements, rearrange the order of items on the screen. And more than anything else, just play a little bit with everything you do. That's the best way to discover the capabilities and limitations of your development tools, and the best way to catch a glimpse of what is really possible.

Of course, beyond the experimentation, learning to use a computer language is similar in many ways to learning a new spoken language. There are vocabularies to master and syntactical rules to learn. Whatever time you spend memorizing those vocabularies and rules will be well spent. But remember, that isn't the sum of learning to program. Vocabulary lessons will prepare you to ace a midterm exam, but it's experimentation with the language that will let you rave all night in a smoky cafe. Both experiences are necessary for the budding programmer.

This learning process can be both easier and more difficult than learning a human language. It's easier because you have a smaller vocabulary to work with, and because there are absolute rules upon which you can rely. But learning a programming language can also be more difficult than a human language, because computers are less forgiving than people. You have to say precisely what you mean, or your program won't perform in the way that you expect. There are no irregular verbs in BASIC, but there isn't any equivalent for "Uh, you know what I mean," either.

C H A P T E R

3

Cut-and-Paste Programming

*Customizing
Applications*

*Linking Existing
Applications*

Using Third Party Code

*Making Cut-and-Paste
Programming
Work for You*

AS YOU SET OUT ON ANY WINDOWS DEVELOPMENT PROJECT YOU HAVE two choices: You can write the entire application from scratch, or you can stitch together existing applications and commercial code libraries to build your application, using as little custom code as possible. Although both options have their merits, I've found that most non-commercial Windows development projects proceed more quickly, more efficiently, and more reliably using the second method, which I call *cut-and-paste programming*.

This method obviously isn't appropriate to all development projects. For instance, if you're intent on building an application that rethinks the entire process of word processing or that revolves around an entirely new species of multidimensional databases—and you plan on selling five million copies of it—then yes, perhaps you had better sit down and write that application from scratch.

But if, as is more likely, the applications you want to build are a little less extraordinary, and slightly more modest in scope than that, then cut-and-paste programming could save you hundreds or even thousands of hours of development time.

Don't Reinvent the Wheel

The idea of cut-and-paste programming is simple: rather than wasting countless hours recreating and debugging routines that other developers have already perfected, you should look for ways to incorporate those existing routines into your custom applications.

For example, most business applications tend to revolve around a fairly small set of basic activities. Perhaps you want your application to gather a little data and maybe perform a little analytical magic on it or compare it to data found in a departmental database. Maybe you'd like the application to be able to print a good-looking report or create some interesting analytical graphics. Perhaps the application has to be able to communicate with a remote mainframe.

Sound familiar? It should. This broad set of functions pretty much covers the basic ingredients of almost every business application. Of course, there are countless variations on each one of those ingredients—word processors, spreadsheets, and drawing programs all gather data, for instance, but they do so in markedly different ways. However, the point is that almost all business applications consist of combinations of those variations, and as a result, it is often possible to find existing program code or applications that perform the functions your custom application needs to perform.

The Windows environment makes it remarkably easy to integrate those functions. You can realize the benefits of cut-and-paste programming in the following ways:

- By taking advantage of such Windows 3.1 facilities as dynamic data exchange and object linking and embedding (OLE)
- By using application macro languages to customize existing applications
- By exploring the application-linking capabilities of Windows batch languages
- By incorporating third-party dynamic link libraries into your custom applications

Cost and Performance

It is only fair to point out, however, that there is some cost in terms of efficiency resulting from following the cut-and-paste programming method. A custom application that links two or three existing applications, for instance, will take up more disk space and operate more slowly than one written from scratch incorporating only the functions it actually needs. A cut-and-paste application will also require a larger out-of-pocket investment than one written from scratch.

But the expense and disk-space considerations can be at least partially mitigated if you're able to make use of applications you already own. And you might also consider the potentially huge time-savings that accrue from using the cut-and-paste method as justifying a relatively small additional out-of-pocket expense.

But what of the performance question? Well, although an application that relies on a macro language or DDE links will perform more slowly than one written from scratch (in a high-performance Windows development tool such as C, for example) it won't be as slow as you might think. Indeed, well-written applications that utilize macro languages (such as those found in Excel or DynaComm) for the tasks they're designed for, can certainly outperform poorly written C applications.

Of course, if you try to use Excel's macro language to build a word processor or DynaComm's to build a graphics program, your application's performance will be abysmal. However, if you stay within the limitations of the macro language and make use of those things that it does well (financial calculations in the case of Excel or telecommunications links in the case of DynaComm), you can achieve perfectly acceptable performance, as several of the example applications found in the later chapters illustrate.

Customizing Applications

The most basic form of Windows development project is one that customizes an existing Windows application. Customization of this sort is done for one of two reasons: to add new functions or capabilities to an existing application; or to transform a general-purpose tool, such as a spreadsheet, into a special-purpose one, such as a bond-ratings analysis program.

Several of the projects discussed later in this book are examples of customizing a single existing Windows application. These include the first example project, The Ultimate Notepad, and the book's final project, M.M.M.: The MCI Mail Manager.

Adding Functions—The Ultimate Notepad

When using Windows' Notepad feature, I was constantly frustrated by its lack of basic text-editing functions such as search-and-replace and the ability to merge text files. But I didn't have the time, or desire, to write a new text-editing program from scratch.

Instead, I used Wilson WindowWare's WinBatch to customize the existing Notepad program, using WinBatch macros to combine Notepad's core functions and WinBatch's own functions to give Notepad the capabilities I needed. For instance, I used a combination of Notepad's existing search function and some of the clipboard and string-handling functions of WinBatch to create a search-and-replace capability that functions as smoothly as Notepad's original search function.

The resulting application, The Ultimate Notepad, is described in more depth, and its complete source code is presented, in Chapter 9.

Creating a Special-Purpose Tool—M.M.M.: The MCI Mail Manager

The M.M.M. project had its origin in my desire to duplicate under Windows the functionality of a DOS application called Lotus Express. Express is a TSR program that logs on to the MCI Mail service regularly throughout the day, sending any messages in its "outbox" and retrieving any messages that are waiting on line. It also provides a series of user-definable "mailboxes" for storing and organizing mail messages, and a text editor for reading and composing messages.

I could have set out to create a Windows version of Express from scratch, but it would have been a Herculean task, including writing all the communications routines needed by the program (modem, serial port, terminal-emulation control functions, and so on), defining and creating a file management system for messages, and writing a text editor so that I could read and compose messages. In fact, I would have had to do so much work to write the program that it simply would not have been worth the time or effort.

As it turned out, of course, there was no need to do so. I was able to build the application to my exact specifications—even adding additional “bells and whistles” such as an off-line address directory—using DynaComm’s script language. In other words, I used a general-purpose tool, DynaComm, to create a special-purpose one, M.M.M.

Creating M.M.M. still wasn’t a trivial task—the resulting application consists of thousands of lines of code. But by using DynaComm’s native functions and the capabilities of its script language, I avoided having to write tens of thousands of additional lines. For example, incorporating DynaComm’s memo editor saved me the work of writing the code for a text editor; its structured record format saved me having to create database-search, -sort, and -indexing routines; and its communications functions saved me the effort of writing terminal-emulation and modem-control routines.

From the point of view of the user, M.M.M. is a stand-alone application. Every menu, dialog box, and icon displayed by M.M.M. is tailored specifically for use by M.M.M.—the user never has to navigate through a DynaComm dialog box or menu in order to use the script. In short, the user need know nothing about DynaComm, despite the fact that the application is built entirely in DynaComm macro code.

You’ll find a complete description of the M.M.M. project, as well as its complete source code, in Chapter 15.

Customizing with Macro Languages

Usually the simplest way to customize an existing Windows application is through use of the application’s own macro or script language. In the Windows applications with the most robust macro languages (Microsoft Word for Windows, Microsoft Excel, Ami Professional, and DynaComm are good examples), you can use the macro language to completely transform the application, replacing its standard menus, dialog boxes, and functions with customized versions tailored to your liking.

Of course, some Windows applications have less robust macro languages, and others have none at all. In those cases, you can use a batch language, or possibly a DDE link with another, more easily customized application, to customize the application.

The more robust application macro languages provide you with a range of capabilities extending from simple modifications of the application’s menus or appearance to the opportunity to create complete new programs that, like M.M.M., utilize the capability of the underlying application while completely hiding that application from the user.

As an example of a simple modification, let’s say you wanted to add a command to the File menu in Word for Windows. This command would load the DOS command processor so that you could issue commands at the DOS command line. In other words, it would perform the same function as

double-clicking on the DOS Prompt icon in the Program Manager. You could add this command by creating the following three-line WordBASIC program:

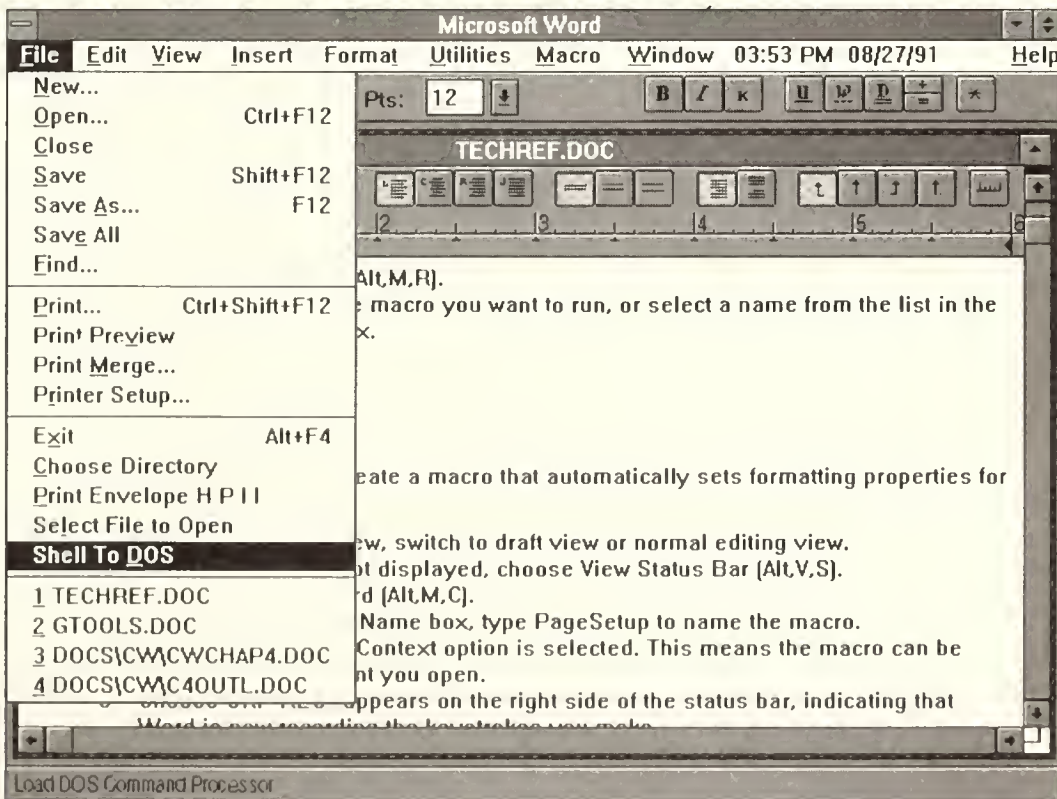
```
Sub MAIN
SHELL "C:\COMMAND.COM"
End Sub
```

This simple macro uses the WordBASIC SHELL command (used to launch another application) to load the DOS command processor, COMMAND.COM. (It assumes you're using the standard DOS command processor, located in the root directory of drive C.)

Once you've created the macro in Word for Windows' macro editor, and saved it with a name such as ShellToDOS, you can use the Assign to Menu option on the program's Macro menu to assign the macro to one of the standard WordBASIC menus. I've added this useful little macro to the File menu in my copy of Word for Windows, as shown in Figure 3.1. (Word for Windows automatically parses the macro name when suggesting a name for its menu item, so that "ShellToDOS" becomes "Shell To DOS".)

Figure 3.1

The ShellToDOS macro added to Word for Windows' standard File menu



From this kind of humble beginning it really isn't a very great leap to creating a completely customized "application within an application," à la M.M.M., using application macro languages. In fact, small beginnings like this often evolve into more intricate applications. You start with a small idea, and that sparks a bigger idea, which sparks a still bigger idea, and before you know it you've got a completely customized application.

One thing you should remember before you get too carried away with any application macro language is that such a language is only as good as its underlying application. Most macro languages add a varied set of window- and application-control features to the basic functions of their underlying applications, along with the ability to generate user-interface elements such as dialog boxes and messages. But beyond those extensions, application macro languages depend on the functional capabilities of the underlying application. So it is simply common sense that the performance of custom applications written in an application macro language will largely reflect the underlying application's ability to perform the tasks of the custom application.

The point is that you shouldn't try to build a spreadsheet application in WordBASIC, or a relational database application in DynaComm. Although each of these languages is strong enough that it has some capabilities beyond the core functions of its underlying application, you should use them with caution, knowing that you risk having your custom application's performance suffer every time you attempt to make it do something its underlying application wasn't designed for.

Does that mean you should never, for instance, attempt to have DynaComm perform any file-manipulation tasks or Word for Windows execute mathematical functions? Of course not. It simply means that you shouldn't attempt to make functions that are peripheral to the main task of the underlying application the central activity of your custom application—at least, not if its speed is important.

Customizing with Batch Languages

Windows batch languages, such as WinBatch and others (see Chapter 4), can be of great value when you want to customize the operation of a Windows application that lacks its own macro language.

The degree of customization that's possible through the use of a batch language is as much dependent on the capabilities of the original application as on those of the batch language. Batch languages bring a valuable set of tools to the fray, including the ability to interact with the user via dialog boxes and message boxes, and to send an application keystrokes and manipulate its windows. However, these languages are still limited by the capabilities of the applications with which they interact. In short, you can extend those capabilities with a batch language program, but you generally can't create new capabilities if the building blocks for them aren't already present in the application.

For instance, in The Ultimate Notepad I was able to add a search-and-replace function to Notepad by taking advantage of WinBatch's ability to create dialog boxes and Notepad's existing search function and support for Windows Clipboard functions such as copying and pasting. The search-and-replace batch program simply creates a dialog box into which the user enters the original and replacement strings. Then the program uses Notepad's search function to find the original string, and the Clipboard paste function inserts the replacement string in its place.

Thus, I was able to implement the search-and-replace function, because all the functional building blocks were present in Notepad. In contrast, no matter how long someone labored at it, it wouldn't be possible to add, for instance, the ability to display TIFF graphics files to Notepad using a batch language, because the program lacks the requisite building block—it can't display graphic images. No amount of batch language programming is going to change that.

But even working within the limitations of the least powerful Windows application, you can achieve pretty impressive results using a batch language. Even though Notepad is an extremely limited program, using WinBatch I was able to add a customized file-open routine, the abilities to insert a file into the current document and to save selected text to disk, a function for automatically indenting program code, routines to convert selected text to upper- or lowercase, and word- and character-count routines, in addition to the search-and-replace capability.

Of course, the more powerful the underlying application, the greater the possibility for customization with a batch language. Applications such as the Microsoft Powerpoint presentation graphics program or the Polaris PackRat personal information manager, which offer tremendous functional richness but lack macro languages, are great candidates for customization with a batch language. The ability to automate the operation of applications such as these, and to build interactive front ends for them, provides you with the raw material for building a thousand great custom applications.

In addition, there may be times when a batch language comes in handy even for customizing a Windows application that does have its own macro language. For instance, consider an application such as 1-2-3 for Windows, which has a macro language that can automate any spreadsheet operation, but lacks the ability to create standard Windows dialog boxes under macro language control. (The user-interface-modification functions in the 1-2-3 for Windows macro language are limited to modifying the 1-2-3 Classic menu bar.) You can build a powerful set of macros in 1-2-3 for Windows, but if you want to automate them through a standard Windows front end, you're out of luck. However, with a few lines of batch language code you could create a standard dialog box that would link your macros into a smooth custom application with a true Windows look and feel.

Linking Existing Applications

So far, this discussion has focused almost entirely on the process of automating or modifying a single Windows application to solve your custom application needs. But that capability—although unquestionably of great value—is hardly the answer to every Windows custom programming problem.

Say, for instance, that you want to build an application that will automatically log on to a remote database, download stock price data, store that data in a table, and produce graphs analyzing price trends, sales volume, and so forth. Unless you've got a single application that offers remote communications capabilities, spreadsheet-like tables, and graphic charting functions, you can't get by with simple customization. What you can do, however, is to link together two or more Windows applications that, in combination, provide the capabilities you need.

The Windows Broker project, presented in Chapter 14, is an example of linking two applications in this way. Designed to perform, among other things, the stock-price tracking and charting functions described above, Windows Broker consists of a series of Excel macros that are linked, via DDE, to custom DynaComm communications scripts.

From the point of view of the user, Windows Broker is a single application—one that is neither particularly Excel-like nor DynaComm-like. The application's main screen includes a series of buttons, one of which is labeled "Update Prices," as shown in Figure 3.2. Pressing that button causes Excel to launch DynaComm in the background, and has it connect to an on-line information service to obtain current price data. Then, a series of DDE exchanges takes place, in which DynaComm sends the latest price information to Excel, which parses it and stores it in the appropriate table.

All of this takes place without any user interaction at all. The user never even sees DynaComm's window. Indeed, if it weren't for the minute or two of delay while DynaComm fetches the latest price information, the user would have no way of knowing that the price information wasn't being retrieved from a local database.

As was true with customizing a single application, you can link multiple Windows applications using either their built-in macro languages or a Windows batch language.

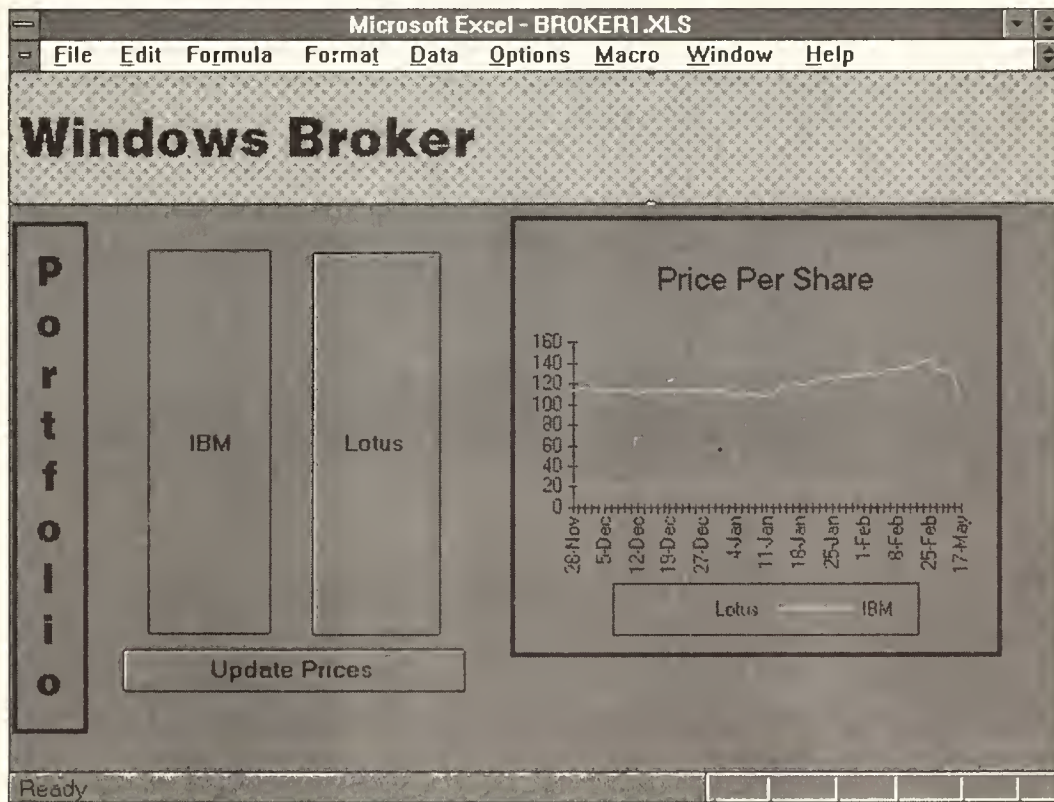
Linking with Macro versus Batch Languages

The advantage of using macro languages to link multiple applications is that they generally have access to all the data structures and capabilities of their underlying applications. For instance, DynaComm utilizes a structured table format, which allows you to create tables containing many records, each of which contains multiple fields that conform to a standard definition. For example, Field 1 might be defined as a 20-character text field, Field 2 might

be defined as an eight-character integer, and so on. By using DynaComm's TABLE SEND command, you can transmit the entire table through DDE to an application such as Excel. The following example would send Table 9 to an Excel worksheet called BUDGET.XLS:

```
ACCESS "Excel" "BUDGET.XLS"
TABLE SEND 9
ACCESS CANCEL "Excel" "BUDGET.XLS"
```

Figure 3.2
The Windows
Broker main screen



In contrast, there would be no way to directly copy the table from DynaComm to Excel using a batch language such as WinBatch, since it lacks both DDE functions and access to DynaComm's internal data formats. To move the same table from DynaComm to Excel using WinBatch, you would have to either have DynaComm display the entire table on screen and then copy it to the Clipboard, or have it save the table to an ASCII file readable by WinBatch. Then WinBatch would have to activate Excel, load the BUDGET.XLS file, and paste the table into it. Either way, the process would entail many more steps, and proceed much more slowly, than it would using DynaComm's TABLE SEND command.

On the other hand, some Windows applications don't offer macro languages of their own, or don't support DDE, or otherwise aren't up to the

task of moving data to another application without a helping hand. In those cases, a batch language can be of immense value. Basically, a Windows batch language can automate any process you can perform from the keyboard of your computer that involves one or more applications. So if you can create a link between two applications manually—even one so simple as highlighting text in one application, copying it to the Clipboard, and then switching to another application and pasting the text there—you can automate that process using a batch language.

When you add that kind of functionality to a batch language's ability to create dialog boxes and otherwise interact with the user, you've got the heart of a custom application that not only links off-the-shelf products, but can take advantage of their functional capabilities as well. That's cut-and-paste programming at its simplest.

Using Third Party Code

Obviously the principles of cut-and-paste programming can be a giant aid to developing custom Windows applications when it is possible to do so by customizing one or more existing applications. But you can also apply those principles to independent applications developed in languages such as Visual BASIC, Turbo Pascal for Windows, and Realizer, by taking advantage of the language's ability to access external code modules in the form of dynamic link libraries.

Incorporating a commercially available dynamic link library into your application can save you an immense amount of development time. In addition, it could provide a higher level of performance on time-critical functions, such as file transfers, than may be achievable through a language such as BASIC.

There are dozens of dynamic link libraries available commercially for use by Windows developers. These offer finished, ready-to-incorporate code that can be used with any development language that supports DLLs to achieve a broad range of functions—everything from generating graphics to querying remote SQL databases. You'll find an extensive list of commercially available DLLs in Appendix A.

In addition, some languages, including Visual BASIC and Realizer, include support for proprietary DLLs. In the case of Visual BASIC, these are called *custom controls*, and are designed to be used only with programs written in Visual BASIC. The advantage of these language-specific code libraries is that they can be integrated more tightly into the language's programming environment than is possible with a generic DLL. For instance, a custom control—such as an animated button—can appear on the Visual BASIC tools palette alongside standard Visual BASIC controls such as radio buttons and list boxes, as shown in Figure 3.3.

Figure 3.3
A custom control
on the Visual BASIC
tools palette



Incorporating Dynamic Link Libraries

Let's imagine that you want to build an application to access an SQL database on your network server. If you were to write the application from scratch, the application's user interface and the functions it makes available to the user would be the least of your worries. Instead, you'd probably be sweating blood trying to figure out how to make your high-level development language communicate smoothly with your network database server.

With the addition of a commercial DLL such as Pioneer Software's Q&E Database Library, however, the whole focus of your development effort can shift. Using that particular library, developers working with any language that can call a DLL have access to a wide variety of database servers, including NetWare SQL and Btrieve databases, IBM's DB2 database, OS/2 Extended Edition databases, Sybase and Microsoft SQL Server databases, and Paradox, dBASE, and Oracle databases. With a few simple commands, you can use the Q&E Database Library to add full database functionality to your application, including (where supported by the database server), such advanced capabilities as COMMIT and ROLLBACK, as described below.

An SQL Example

SQL databases allow you to group multiple changes to a database into larger units, called *transactions*. The COMMIT and ROLLBACK statements are used to confirm or back out of all the changes made to the database during a transaction. That is, when the COMMIT command is issued, all the changes made during the current transaction are permanently recorded, whereas when the ROLLBACK command is issued, all the changes made during the current transaction are discarded.

You can add the COMMIT and ROLLBACK functions to a Visual BASIC program by declaring them as follows:

```
Declare Function qeCOMMIT Lib "QELIB.DLL" (ByVal hdbc%) As Integer
Declare Function qeROLLBACK Lib "QELIB.DLL" (ByVal hdbc%) As Integer
```

These declarations tell Visual BASIC to pass calls to the functions defined as qeCOMMIT and qeROLLBACK to the dynamic link library called QELIB.DLL. The integer variable hdbc% is used to identify the database link upon which the command should operate.

In order to use the functions declared above in your Visual BASIC program, you would first have to establish the connection to QELIB.DLL with a statement such as this:

```
hdbc=qeCONNECT("DRV=QEDBF")
```

Then, to make use of the qeROLLBACK function, you would issue a command such as

```
Rollback=qeROLLBACK(Currenthdbc)
```

which would tell the Q&E Database Library DLL to undo the current transaction.

Because Q&E Database Library is designed to work with any application that can make use of DLLs, you could add the same functions to a custom application written in Excel's macro language by defining them as follows:

```
func21=REGISTER("QELIB.DLL","qeCOMMIT","II","qeCOMMIT","hdbc")
func22=REGISTER("QELIB.DLL","qeROLLBACK","II","qeROLLBACK","hdbc")
```

In either case, the result is that you can concentrate your development efforts on creating an attractive, useful interface to these database functions, and on deciding how those functions can best be implemented to provide the maximum benefit to your application's users. You can leave the down-and-dirty work of actually writing the SQL interface to an expert in that field.

Making Cut-and-Paste Programming Work for You

There are two primary benefits of cut-and-paste programming. The first is that it saves you time. For a relatively small expense, you can add functions to your applications that would take countless hours to program yourself. So, unless you severely underestimate the value of your time, it isn't hard to justify the expense of a package such as MicroHelp Communications Library, a \$189 DLL that provides a wide variety of asynchronous communications routines, terminal emulations, and protocol file-transfer routines for use in your applications.

Equally as important, cut-and-paste programming allows you to concentrate on the part of the program that will mostly directly determine its benefit to you—its user interface and the manner in which it presents or structures its functions. You can devote yourself to those parts of the program that you want to have operate in a custom manner, without having to worry about the parts that should work in a standard way.

No matter whether you're using Ami Professional's macro language, combining Excel and DynaComm functions, or generating a custom program in Realizer, with cut-and-paste programming you can avoid much of the drudgery of writing original code. In all likelihood, someone else has already done the dirty work for you. Why write a YMODEM file transfer routine when there's a perfectly good one available in your copy of DynaComm? Why should you create text-formatting routines when Word for Windows can format the text for you?

And not only will cut-and-paste programming get you out of writing the standardized parts of your application, let you concentrate on the things that will make it unique, and save you countless hours of work, it will add immensely to the richness and reliability of your application.

So don't waste your time reinventing any wheels.

C H A P T E R

4

**Choosing
Your Tools**

Evaluating Tools
Windows
Development Tools

THE FIRST STEP IN ANY WINDOWS APPLICATION-DEVELOPMENT PROJECT IS choosing the right development tool for the job. The tool you choose will play a large role in determining the structure of your application, the features you can give it, the degree to which it can be extended to serve other purposes in the future, and the kind of performance and results you can expect from it. Your choice here will also have a major impact on how long it will take you to master the tool's intricacies and, ultimately, how long your development project will take.

The high-level development tools available for Windows cover a huge range in terms of their complexity and capabilities. Some, like the Windows batch languages discussed below, are extremely simple to use. They even have recorder features that will turn your keystrokes into batch language program commands. Others, like the implementations of BASIC and Pascal for Windows, are considerably more complex. However, they also allow you to build much more complete and robust applications than do batch languages, which are suited primarily for use in building small utility programs. Other tools, such as the macro languages built into many Windows applications, fall somewhere in the middle, mixing a powerful, but limited, range of functions with a relatively simple command syntax.

So how do you choose one Windows development tool from among all those available to you?

The Biggest Hammer

There are several possible strategies you might employ. One, which I'll call the Use The Biggest Hammer You Can Find theory, is to select the most comprehensive tool available, no matter how simple the development project you have in mind might be.

The reasoning behind this strategy is that if you select the most powerful and comprehensive development tool available right from the outset, it will be capable of handling any extensions or modifications you might wish to make to your initial project at any time in the future. It also will be capable of handling any other Windows development project you later choose to undertake. Because you'll be using the same tool for every project, you'll only need to learn how to use one programming language. You might even be able to save yourself some work on subsequent projects by reusing parts of the applications that you've already created. In addition, you can be fairly sure that whatever applications you write using the tool will be able to exchange data freely, since they will all have been written using the same set of functions.

However, there are also several disadvantages to this approach. The biggest hammer isn't always the best one for the job. It might be difficult to control when you're aiming at little nails, and using it is surely going to consume a lot more energy than swinging a hammer sized just right for the job at hand. For instance, if you're taking the Biggest Hammer approach, and you

want to produce a utility to format documents in a particular fashion, you might end up writing thousands of lines of code and producing a program occupying hundreds of kilobytes of disk space, when all you really needed to do was to write a few dozen lines of macro code in a professional word processor to achieve the same, if not better, results.

Then too, there are times when a hammer just won't do—when you really need a screwdriver or a socket wrench. The high-level languages discussed in this chapter are all well-suited to specific tasks, but none of them is really capable of producing useful results for every programming task that comes down the pike. The only Windows programming tool that actually can do that is the Microsoft Windows Software Development Kit (Windows SDK). But the Windows SDK is so complex that it makes *every* Windows programming task, no matter how trivial, into an ordeal. With the SDK, you have to forge the hammer before you even start hunting for the nail that you want to hit. The reason higher-level languages exist in the first place is that the SDK is simply too complex to be cost effective for developing anything but mass-market commercial applications.

The Smallest Hammer

That brings us to the second approach to choosing Windows development tools, using the simplest tool possible—the Use The Smallest Hammer That Will Do The Job theory.

The advantages of this approach are fairly obvious. By using the simplest tool that is capable of handling the job you have in mind, you can take the most direct route possible to solving your immediate problems. Simple tools are easier to program—as long as you don't try to exceed their capabilities—so your application development will proceed more quickly than with a more complex tool. In addition, since this route will often entail using either the macro language from a Windows application you already have or a low-cost batch language, it might also be the most cost-effective way to develop or customize an application.

The downside of the Smallest Hammer theory is that sometimes the nail is a lot bigger than it first looks. Even the simplest application-development projects often end up being a good deal more complex than they appear initially—either because of hidden difficulties that you don't discover until you're halfway through the development process, or because once the application is finished you or your users start coming up with a long list of ways in which it might be enhanced or extended. You could be out of luck if the application development tool you've chosen isn't flexible or powerful enough to handle the changes—you'll either have to say “It can't be done” or scrap the work you've already finished and start over with a more powerful tool.

The Right Hammer

That leads us to the third strategy for picking a Windows development tool, which falls somewhere between the first two—and is the one followed in putting together the example development projects in Chapters 9 through 15. Let's call it the Pick The Right Tool For The Job theory.

This theory is a bit less dogmatic and, alas, a bit more amorphous than the other two. It rests on the premise that there is no perfect tool for every Windows development job. Instead, the savvy Windows developer must be familiar with and willing to make use of a variety of tools, depending on the demands of the application being developed. The selection you make of which tool to use must reflect both the initial functional requirements of the project at hand and its potential need for expansion in the future.

One way to classify development projects is to identify them as either *tactical* or *strategic*. Tactical projects fill an immediate need—like sticking your finger in a leaking dam. Strategic projects have a major long-term impact on the way you function—like building a complete new system of dikes and levees.

The Auto Print for Windows application (described in Chapter 11) is a good example of a tactical Windows development project. Auto Print allows the user to designate a list of files to be printed overnight to avoid tying up the PC with lengthy print jobs during the day. Auto Print didn't need a fancy interface, and it was unlikely the program would ever need to be extended to handle any major new functions, so I elected to use a Windows batch language—the fastest and easiest available development tool—for the project.

In contrast, the DocMan document management application, described in Chapter 13, is a good example of a strategic application. Designed to log and maintain a database of documents created by a variety of Windows applications, and to allow the user to select documents based on long file names, descriptions, or keywords, DocMan needed to be as slick and extensible as possible. Thus I elected to produce it in Visual BASIC, a language with good extensibility and fine user interface design tools.

The basic guidelines followed in this book are these:

- With tactical applications, choose the tool that will get you up and running as fast as possible.
- With strategic applications, make doubly sure the tool you choose is capable of handling every expansion of the original application requirements that might be necessary in the future.

How do you determine if a Windows development tool is up to that task? Read on.

Evaluating Tools

As you saw in Chapter 2, all programming languages or development tools are built around a series of basic elements. Although they may differ in name or implementation from language to language and tool to tool, objects such as variables, strings, functions, procedures, and commands are found in every development tool. Every programming language has loops or some similar way to perform iterative processes, and every programming language has other structures designed to control an application's flow of execution.

You can pretty much assume the presence of these basic elements when you evaluate Windows development tools. Certainly it would be noteworthy if someone offered a programming language for Windows that didn't include any means to compare the values of two variables or to add together two integers. However, I haven't seen one yet that fails to cover these bases, and I'm fairly confident I won't see one in the future. Obviously, there are significant differences in the way these basic elements are implemented from language to language, and these will be examined below in the discussion of individual tools. For the most part, however, you can count on any of the higher-level-language development tools discussed here to meet these basic requirements.

There are, however, some special facilities, unique to Windows programming, that you need to be a little more careful about. These facilities are not universal, but their presence and the manner in which they're implemented will go a long way toward ensuring that the application-development tool you choose is capable of doing everything you want it to do in the Windows environment. These facilities fall into three main functional areas:

- Those that facilitate the interaction of your application with other applications in a multitasking environment
- Those that help you build the user interface for your application
- Those that help you work with files created by other applications

Multitasking Facilities

Multitasking facilities allow your application to control and exchange data with other applications, and thereby allow you to customize the operation of those programs or to link multiple applications into an integrated system. The most important multitasking capabilities are Clipboard support, application control, and DDE support. In addition, there is a further adjunct to these capabilities, DLL support, that goes a long way toward determining the extensibility of the development tool.

Clipboard Support

A Windows development tool should have the ability to copy data to the Windows Clipboard and to retrieve data already there. Ideally you should be able to assign the contents of the Clipboard to a string (assuming that the clipboard holds textual, not graphic, data). This capability speeds the process of integrating existing applications by providing a simple way to move data between applications.

One Windows development tool that offers very good support for the Clipboard is Wilson WindowWare's WinBatch. The WinBatch language includes three Clipboard-related commands: CLIPAPPEND, CLIPGET, and CLIPPUT. CLIPAPPEND is used to *append* the contents of a string to the Clipboard. (That is, it attaches the string to the end of anything already on the Clipboard.) CLIPGET is used to retrieve the contents of the Clipboard and assign them to a string. And CLIPPUT is used to copy the contents of a string to the Clipboard.

Other application development tools implement Clipboard support differently. For instance, the macro language in Lotus Ami Pro 2.0 includes commands called Copy, Cut, and Paste, which, like their menu equivalents, are used to copy or cut selected data from the current document and place it onto the Clipboard, or to paste data from the Clipboard into the current document. In addition, Ami Pro's macro language includes functions called CLIPREAD and CLIPWRITE, which are equivalent to WinBatch's CLIPGET and CLIPPUT.

Application Control

If the application you're building is going to have to interact with other applications, it's important that it have the ability to launch them, send them key-strokes, and activate and deactivate them.

This kind of application control is less sophisticated than Dynamic Data Exchange, in which your application engages in a conversation with another Windows application and sends it explicit instructions. However, it is also more universal since an application development tool with these facilities should be able to control any other Windows application, not just those that support DDE.

Example: Visual BASIC

Microsoft's Visual BASIC is an example of a development tool that offers a fairly complete set of commands for controlling other Windows applications. Its SHELL command can be used to launch other programs. For instance,

```
SHELL("EXCEL.EXE", 1)
```

would launch the Excel spreadsheet program (assuming that the Excel directory was in the DOS path). The numeral 1 in the command tells Visual

BASIC to launch Excel as a normal window that will have the *input focus*—meaning that the keystrokes typed or mouse commands issued by the user will be directed to Excel, not to your Visual BASIC program. You could also launch it as an icon or as a maximized window, with or without the focus.

Once an application has been launched, Visual BASIC's SENDKEYS command can be used to send data to it in a way that makes the other application think the user is typing that data at the keyboard. This enables your Visual BASIC program to issue any commands to the other application that you would be able to enter at the keyboard. Your program can even issue menu commands in the other application or instruct it to close itself down by sending the keystrokes necessary to issue the application's File Exit menu command, as in

```
SENDKEYS "{ALT}FX"
```

Before you can send keystrokes to another application from your Visual BASIC program, however, you might have to *activate* it, if some other application—for instance your Visual BASIC application—has had the input focus. To do so you would use Visual BASIC's APPACTIVATE command, which requires one parameter—the exact text of what appears in the application's title bar. Thus, to activate Microsoft Word for Windows, you would issue the APPACTIVATE command, followed by the text that appears in the title bar, "Microsoft Word":

```
APPACTIVATE "Microsoft Word"
```

The problem with this is that you can't always predict what text is going to appear in the title bar of a Windows application. For instance, although in most cases the title bar for Word for Windows consists of just the text "Microsoft Word", if you maximize the document window within Word for Windows, the title bar changes to include the title of the current document, as in: "Microsoft Word - CHAPTER2.DOC". If you simply issue the command

```
APPACTIVATE "Microsoft Word"
```

when the title bar contains the document title, Visual BASIC won't find the correct window, which will generate an error message and more than likely crash your application.

Thus you need a way to obtain a list of the exact titles of all open windows so that you select the appropriate one. Unfortunately, Visual BASIC doesn't provide a command or function to do so. Instead, you've got to call a series of Windows API (application programming interface) functions to activate the window.

The use of Windows API functions from within Visual BASIC is discussed in detail in Chapter 12.

Example: WinBatch

Other languages make it easier to control Windows applications that are already running, and therefore may be better suited for use in controlling other applications. For instance, WinBatch uses a command called WINACTIVATE that performs the same function as Visual BASIC's APPACTIVATE. However, the former is forgiving enough to require only a partial window title—just enough to identify the application you're trying to activate. So, in the case cited above,

```
WINACTIVATE("Microsoft Word")
```

would activate the window entitled Microsoft Word - CHAPTER2.DOC, even though Visual BASIC was not able to.

In addition, WinBatch has several other commands that make it useful for controlling other applications. These include WINITEMIZE, which returns a tab-delimited list of all open windows, and WINGETACTIVE, which returns the title of the active application's active window. This can be useful if, for instance, Word for Windows is running and a dialog box labeled "Microsoft Word—Error Warning" is visible on screen. In that case, the WINGETACTIVE command will return the full title of the dialog box, enabling the program to determine whether an error has occurred. Other WinBatch commands for controlling applications include WINPLACE, WINTITLE, WINSHOW, WINHIDE, and WINICONIZE, which are used, respectively, to reposition, change the title of, show, hide, and iconize other applications.

DDE Support

Support for Dynamic Data Exchange is another key feature in a development tool that will be used to build applications that will interact with other applications.

DDE is a remarkably powerful tool that allows applications to create *hot links* to data in other applications, so that the data in one application is updated whenever the data in the other application changes. DDE also allows applications to communicate directly with one another, issuing commands through the DDE channel rather than via the keyboard.

Unfortunately, DDE is also a quagmire in which many a brave Windows developer has been lost forever. There are no official standards for DDE, and thus every Windows application, and every Windows development tool, implements it differently. About the only thing the dozens of implementations of DDE I've encountered have in common is that they all are designed to communicate first and foremost with Microsoft Excel, and then with whatever other applications happen to speak a reasonably similar DDE dialect.

Thus, Excel's implementation of DDE has become the de facto standard for how DDE should work. With that in mind, let's look at the way Excel's macro language implements DDE.

Example: Excel

DDE conversations in Excel start with an INITIATE command, which specifies both what application the conversation will be held with and the topic of the conversation. If you're creating a hot link between a worksheet and a Word for Windows document, the command would look like this:

```
=INITIATE("Winword", "CHAPTER2.DOC")
```

whereas if you're creating a link that will be used to send commands to the other application, the topic of the conversation will be the application's System topic, and thus the command would look like this:

```
=INITIATE("Winword", "System")
```

In either case, this command returns a value identifying the number of the DDE channel that has been opened.

To receive data from another application, you would use Excel's REQUEST command, as in

```
=REQUEST(Channel_Num, Requested_Data)
```

Similarly, you can send data or commands to another application using Excel's POKE command.

Excel will also respond to REQUEST or POKE commands, or to instructions addressed to its own System topic. For example, to instruct Excel to open a worksheet called MARKUPS.XLS, an application would send the following command through DDE to Excel's System channel:

```
[OPEN("MARKUP.XLS")]
```

(The syntax for sending a command to Excel's System channel will vary greatly from development tool to development tool.)

Finally, Excel uses a command called TERMINATE to close down a DDE conversation once it has been completed, as follows:

```
=TERMINATE(Channel_Num).
```

Other applications and development tools may implement additional DDE commands, including an ADVISE function, which is used to inform another application that the data linked through the current DDE channel has changed.

DLL Support

The ability to make use of dynamic link libraries is a critical one in a Windows development tool because it provides the tool with almost infinite extendibility. It can also eliminate hundreds of hours of coding time from a programming project.

A dynamic link library is a compiled program that is designed to be accessed by other Windows programs—not to run by itself. Most of the functions built into Windows 3.1 itself are actually stored in a series of dynamic link libraries that are called by the Windows program. These same libraries can also be accessed by any Windows application that can utilize DLLs.

For instance, consider Visual BASIC. The API call cited above as a way to obtain the text of a Windows application's title bar is actually a function in USER.EXE, a dynamic link library that contains most of Windows 3.1's user interface functions.

However, because Visual BASIC can address any DLL, not just those included with Windows 3.1, there are simpler ways to achieve the same ends. For example, you could use the GETWLST.DLL written by Todd Steinwart (available as freeware through CompuServe—look for it in the file VBSW11.ZIP in library 5 of the MSBASIC forum). GETWLST simplifies the process by providing your application with a list of the *handles* (the hexadecimal references by which Windows refers to active applications) of all active applications, as well as a function for obtaining the window title text of any application whose handle you specify.

In fact, through the addition of freeware, shareware, and commercial DLLs, you can extend the capabilities of any Windows application-development tool that can interface with DLLs. Using this capability, you can add capabilities such as network support, the ability to query SQL databases, and the ability to display graphics and charts to your applications with just a few lines of code as was discussed earlier in Chapter 3, "Cut-and-Paste Programming."

User Interface Facilities

The other unique-to-Windows set of facilities you should look for in any application-development tool for Windows are those facilities that will enable you to design and control the user interface of your application.

The user interface facilities of high-level Windows development tools vary quite a bit from tool to tool, in terms of both their capabilities and how easy they are to exploit and in what they are designed to accomplish.

In the case of application macro languages (where the intent is to customize an existing application) and batch languages (which help you develop utility applications quickly), user interface tools are generally more limited than they are in full-fledged programming languages designed to create stand-alone applications.

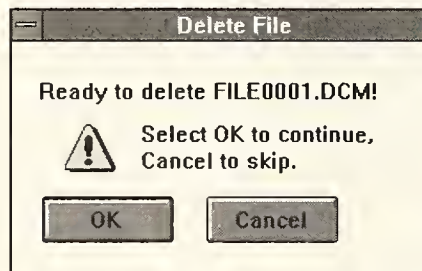
However, even the simplest macro and batch languages give you a fair degree of control over the user interface of your application, allowing you to create and make use of custom dialog boxes and, in most cases, custom menus. General-purpose programming languages go considerably further, allowing you to control every aspect of your Windows application's user interface.

Dialog Boxes

Dialog boxes and menus are the primary means by which Windows users communicate their intentions to an application, so it is important that whatever Windows development tool you use provides sufficient capabilities in these areas to support your application. At a minimum, your application will probably need the ability to create message boxes on screen that tell users what the application is about to do and elicit their response. Figure 4.1 shows a simple message box, created in the script language from DynaComm, Future Soft Engineering's asynchronous communications program.

Figure 4.1

A message box
created in
DynaComm



This particular dialog box displays a message alerting the user that a file is going to be erased, an icon that serves to draw attention to the dialog box, and a pair of buttons labeled “OK” and “Cancel.” These simple elements combine to create an informative and easily understood message.

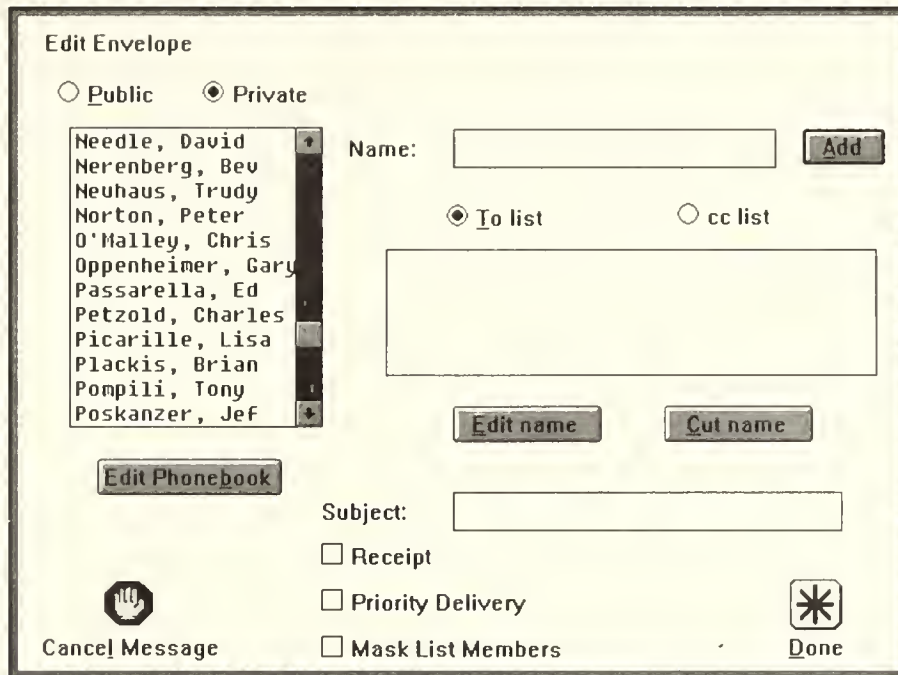
Of course, many applications require dialog boxes that are considerably more complex than a simple OK/Cancel message. For instance, Figure 4.2 shows a much more complex dialog box, also created in DynaComm, that is used to address mail in the M.M.M.:The MCI Mail Manager application, described in Chapter 15. This dialog box, used in the message-addressing phase of the application, allows the user to pick names from either a public or a private phone book and to add them to a mail message's To: or cc: field. In addition, the user can type in a new name, fill in the message's Subject field, assign handling options to the message, and choose to edit either phone book, all from this single dialog box.

This dialog box takes advantage of several additional Windows user interface controls, including edit fields, list boxes, and radio buttons. It also requires a capability found in DynaComm's macro language that is missing in some other application macro languages and batch languages—the ability to update a dialog box on the fly in response to user input. For instance, when the user clicks on the Public button, the public phone book is loaded into the list box so that names can be chosen from it. Similarly, when the user clicks on the button labeled “cc list”, the list of cc: addressees for the message is displayed, and the list of To: addressees is temporarily hidden. Meanwhile, double-clicking on

any name in the list box adds that name to either the To: list or the cc: list, depending upon which is active. Typing a new name into the Name edit field and then pressing Enter has the same effect.

Figure 4.2

The message addressing dialog box



These dialog boxes were both created by typing code into DynaComm's script editor. The code used to create the Delete File dialog box shown in Figure 4.1 looks like this:

```
DIALOG (72,52,126,70) "Delete File" ICON CAUTION
MESSAGE (8,11,104,10) "Ready to delete FILE0001.DCM!"
MESSAGE (39,24,86,20) "Select OK to continue,Cancel to skip."
BUTTON (9,47,40,14) DEFAULT "OK" SET OK% 1, RESUME
BUTTON (59,47,40,14) CANCEL "Cancel" SET OK% 0, RESUME
DIALOG END
WAIT RESUME
```

Because this is a fairly simple dialog box, the code to create it is also fairly simple. The first line creates the dialog box itself, using DynaComm's unique coordinate system, and places "Delete File" onto its title bar. The second line draws the caution icon—the exclamation point. The third line tells the user that FILE0001.DCM is about to be deleted. The fourth line tells the user how to proceed with the file deletion or how to cancel it. The fifth and sixth lines draw the OK and Cancel buttons and tell DynaComm what to do when each is pushed. Finally, the last two lines tell DynaComm

that the dialog definition has ended and that it should pause program execution until a RESUME command is issued—which will happen automatically when the user presses either button.

Because the dialog box shown in Figure 4.2 is much more complex, the code used to generate it is much longer. You can examine it later in Chapter 15. But even a cursory glance at the code used to create the Delete File box shows that there are some complex issues to be dealt with in even the simplest dialog box. For instance, you have to position each user interface element where you want it, using a complex coordinate system. That wasn't too difficult to achieve with the little Delete File dialog box, but the message-addressing dialog box has 20 unique user interface elements that have to be positioned in that way. You need to calculate horizontal and vertical coordinates and height and width for each element. Doing this can be a real chore, especially when halfway through the process you realize that the dialog box would be more easily understood if you moved some of those elements around.

Fortunately, many Windows development tools include graphic dialog box editors, which enable you to escape much of this drudgery. Rather than typing in archaic horizontal, vertical, and size coordinates, you simply select a user interface element from a menu or toolbox and draw it on the screen wherever you want it to appear. If you later decide that it isn't positioned or sized correctly, you simply move it by dragging it with the mouse or resize it by stretching its corners.

A dialog editor can thus save you hours of work by letting you design your dialog boxes quickly and effortlessly. The current version of DynaComm doesn't include one, but many Windows application macro languages do, as do most full-featured languages and some batch languages. For instance, Figure 4.3 shows the dialog editor that is included with Microsoft Excel 3.0.

Menus

Along with dialog boxes, a full-featured Windows application needs menus. Most of the high-level Windows development tools described below allow you to create your own menus or, in the case of application macro languages, to modify the menus of the underlying application. Again, however, the way in which this has been implemented varies from tool to tool. Some provide menu editors that allow you to create or customize menus with a few clicks of the mouse. Figure 4.4 shows the menu editor from Microsoft Visual BASIC.

Other tools require you to type the code necessary to create menus manually. In either case, the important thing is that you are able to create menu items and designate the actions that are to take place when the user selects a particular menu item.

Figure 4.3

The Excel 3.0
Dialog Editor
Screen

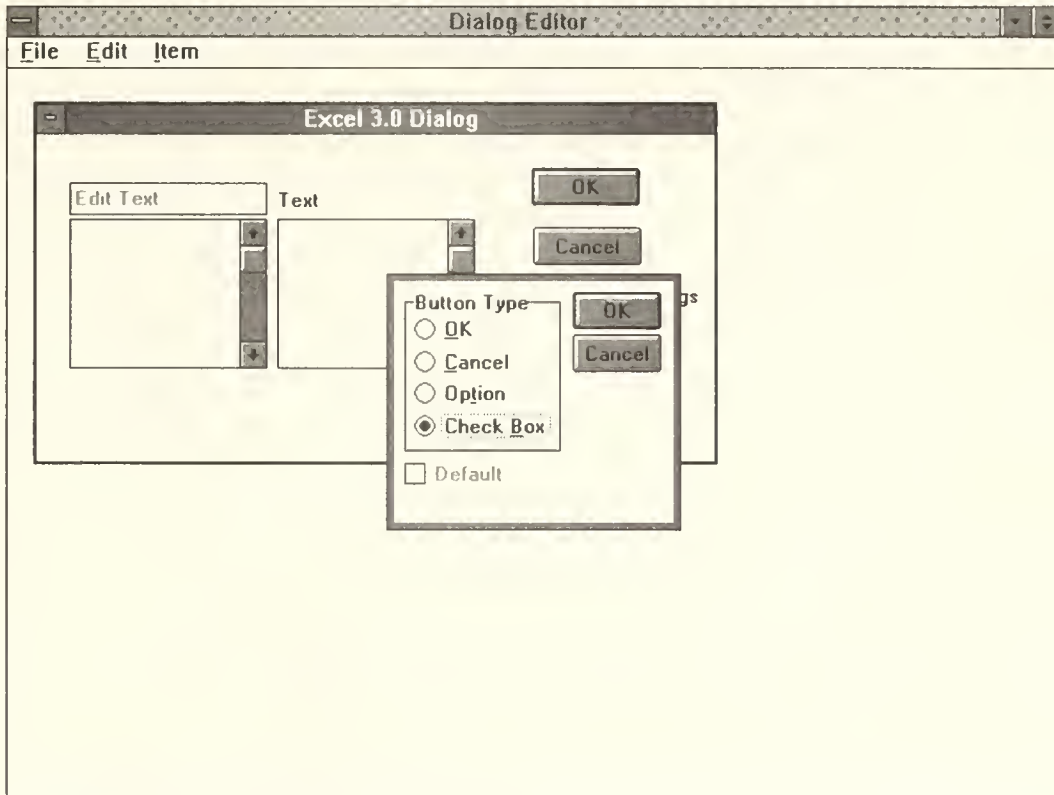
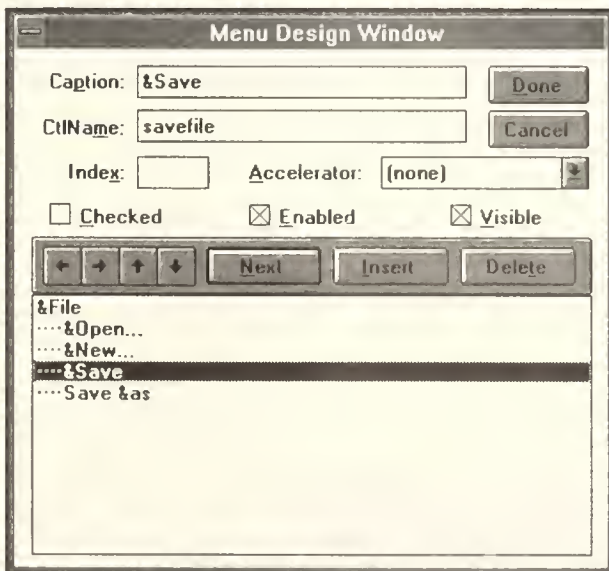


Figure 4.4

The Visual BASIC
menu editor



Screen Design

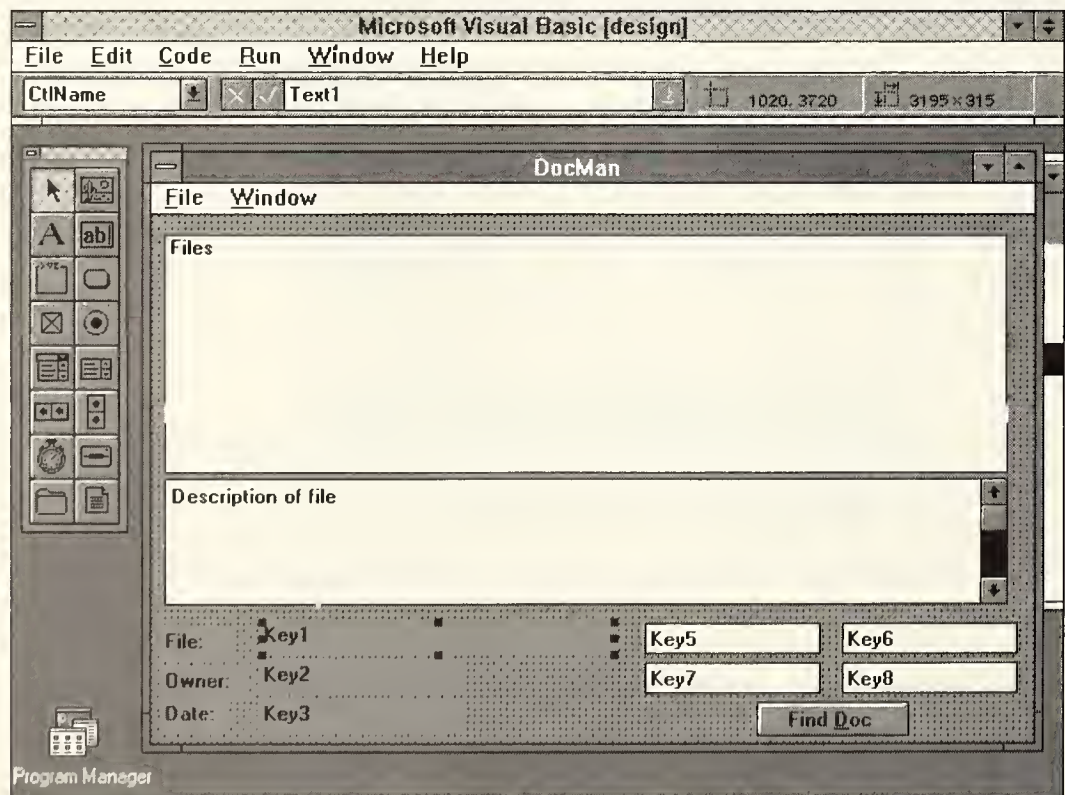
Simply being able to modify an application's menus or create dialog boxes is fine if you're only building an automated macro that utilizes the underlying capabilities of an existing application program, or if you're using a batch language to construct a quick utility, but when the time comes to build a complete stand-alone application, you need even more control over your application's user interface. You'll want to be able to control every aspect of your application's screen appearance.

Fortunately, most of the full-featured high-level programming languages provide excellent tools for designing your application's screen. These allow you to make use of the full gamut of Windows user interface controls—everything from buttons and list boxes to bit-mapped images and icons—in designing your application. And they usually use the same drawing program-like controls that a good dialog editor supplies.

Figure 4.5 shows the Visual BASIC screen editor being used to design the main window for the DocMan application (described in Chapter 14). The floating tools palette visible at the left side of the screen allows you to pick the controls you want to add to your application's window and drag them to the position where you want them to appear.

Figure 4.5

The Visual BASIC screen editor.



The user interface of any application you design in Windows will consist of a combination of menus, dialog boxes, and windows. Thus it is important to consider the facilities for creating these elements when selecting a high-level language development tool for your Windows development projects.

File Formats

A final consideration that may be of great importance in selecting a Windows development tool for your project is the ability of the tool to work with files created by other applications.

You can count on any Windows development tool having the ability to read and write ASCII-format text files, but frequently that capability is not enough. If your application is to work in concert with other Windows or DOS applications, it may need to be able to read files in and/or write files to “alien” file formats. This will enable your application to exchange data with others, or to modify the files created elsewhere to add additional information or functionality.

The available Windows high-level language-development tools offer a wide range of capability in this area. Probably the strongest support for alien file formats is offered by application macro languages, which can take advantage of the file import/export capabilities of their underlying applications. For instance, an application written in the macro language for Lotus Development Corporation’s Ami Pro 2.0 automatically gains the ability to read and write dozens of file formats, including data files from dBASE, Paradox, DisplayWrite 4, Enable, Excel, Microsoft Word, Samna Word, WordStar, WordPerfect, and SuperCalc, and a variety of graphic file formats including BMP, PCX, PIC, TIFF, EPS, and CGM.

Batch languages, on the other hand, can typically only read and write ASCII format files, whereas full-featured programming tools, such as Pascal and BASIC, can typically read a few additional formats—although this varies greatly from tool to tool. Visual BASIC, for instance, can only read ASCII data files and PCX and BMP graphic files, but another Windows BASIC, Realizer (from Within Technologies, Inc.), can read Excel, Lotus 1-2-3, and dBASE data files and BMP graphic files. You can, however, overcome some of the limitations of these products by purchasing dynamic link libraries that provide access to additional file formats, as described in Chapter 3.

Windows Development Tools

Now that we’ve covered some of the factors that you need to consider in selecting Windows development tools, it’s time to examine the best tools currently available for high-level programming in Windows.

These tools fall into five broad categories:

- Application macro languages
- Windows batch languages
- Windows implementations of BASIC
- Pascal for Windows
- Graphical hypertext products

There is some overlap among these categories, and quite a bit of duplication in the capabilities of the tools in a given category. But much benefit can be gained from looking at the tools in terms of categories because each category has a basic set of characteristics which can be used to evaluate its members.

The basic cost of each high-level development tool is included in the descriptions that follow. In comparing these costs, however, you should also consider the cost of distributing the applications you build. For instance, a program built using an application macro language—say WordBASIC—can only be used in the presence of the application that language is specific to, which would be Microsoft Word for Windows in this case.

Some development tools can compile programs written using them to stand-alone .EXE files, which can be distributed without a copy of the development tool. Others require that programs written using the tool be used in conjunction with a special runtime version of the tool. In some cases (noted below), there is a separate charge for a license to distribute the runtime version of the tool.

Finally, a few of the tools described below are distributed as *shareware*. These programs are distributed via electronic bulletin boards and by user groups or other organizations on an honor-system basis. You don't pay to acquire a shareware application or tool, but if you continue to use it beyond a brief trial period you are legally obligated to pay a specified license fee to its author.

Application Macro Languages

The first thing that may come to mind when you think of an application macro language is something on the order of the macro language incorporated in Lotus 1-2-3 from version 1.1 on. That language allowed you to automate repetitive procedures, such as inserting a column into a worksheet, by storing the keystrokes you would use to perform that operation manually, and then recalling them when you issued the hotkey command assigned to that macro. It also allowed you to construct menus that could be used to link multiple macros of that sort into more complex systems. For instance, you could create an entire accounting system in 1-2-3 by using macros to move the cursor around the screen, get user input, and process the data appropriately.

The basic principle behind a Windows application macro language is the same as that behind 1-2-3's: The macro language allows you to automate processes that make use of the underlying application's core functions. Like the macro languages found in DOS applications, the macro languages of major Windows applications allow you to either automate repetitive tasks or to build complete custom applications.

However, the results you can achieve with a Windows application macro language are often considerably more spectacular than those obtainable with one built into a DOS application, if only because Windows applications tend to have a functional richness that greatly exceeds that of any DOS-based application.

Moreover, the strongest Windows application macro languages give you an unprecedented ability to customize the appearance and operation of the underlying application. You can, for instance, build an application in Microsoft Excel that so completely insulates users from the underlying application that they never see anything that looks like a spreadsheet cell—as illustrated in the Windows Broker application discussed in Chapter 13. Or you could use DynaComm to build an automated electronic mail system that completely hides the on-line session from the user—as illustrated in M.M.M.: The MCI Mail Manager application discussed in Chapter 15.

And no matter how much you alter the appearance or operation of a Windows application, you can take full advantage of its underlying capabilities. So even though a Windows Broker user never sees a spreadsheet cell, the application takes full advantage of Excel's ability to perform financial calculations.

Of course, Windows application macro languages are also extremely useful for less ambitious projects. You might, for instance, simply want to build an automated telephone messaging system in Word for Windows or Ami Pro so that users can simply select a menu item labeled "New Message" to create and then store a new phone message. That kind of function is very simple to write using the macro languages built into these applications because most of the difficult parts of building such an application—providing editing functions for messages, deciding on file formats, figuring out how to print messages—have already been written for you. You can simply take advantage of the underlying application's editing, filing, and printing functions with macro commands as simple as File Save and Print.

To simplify the task of macro programming even further, most Windows application macro languages offer the ability to record keystrokes and then translate them into macro code. So you "write" much of the code for a repetitive function simply by executing it once with the language's recording feature turned on. Then all you have to do is incorporate the code that the recorder creates into your application.

There are, of course, limitations to what you can do with an application macro language—although I venture to say that the limitations are far less restrictive than many users imagine. You are limited to exploiting the

capabilities of the application whose macro language you're using. Those can be extended quite a bit through DDE links to other applications or through dynamic link libraries, but you still have to know what the application you're working with can do well and what it can't—and there you have to use your judgment. For instance, the macro language in Word for Windows provides nearly as wide a range of statistical and financial functions as the average spreadsheet program, but heavy duty math isn't what Word for Windows was designed to do best. So even though those capabilities are there, it would be a poor choice for building an application that requires extensive math. Your application might work, but it would too slow to be of much real use.

In general, you won't get the same speed from an application that you build in a Windows application macro language that you would from one you write from scratch in BASIC or Pascal or some other language. But if you plan your application well and don't try to get too ambitious with it, you can obtain a more-than-acceptable performance in most cases.

A more serious limitation of application macro languages is that there is no standardization of these languages. Even among vendors offering several macro language-equipped products, such as Microsoft's Excel and Word for Windows, the application macro languages of each product bear little resemblance to one another. That's all right if you're going to write all your programs in one application's macro language, but if you intend to write several applications in different macro languages—or if you attempt to write an application that utilizes DDE to link macros written in several applications—you'll have to learn a new macro language for each new application you use. That's not an impossible task by any means, but it is an annoyance and can slow down your development project.

The following sections detail the strongest macro programming languages for Windows applications.

Ami Pro

Ami Pro is a professional-level word processor that includes a spelling checker and thesaurus; drawing, charting, and image-processing modules; and a method of dealing with text and graphics frames that allows tremendous control of page formatting and layout. It also offers the ability to import and export a wide variety of file types and a feature called *revision marking*, a document information facility that can be used to assign descriptions and keywords to each document.

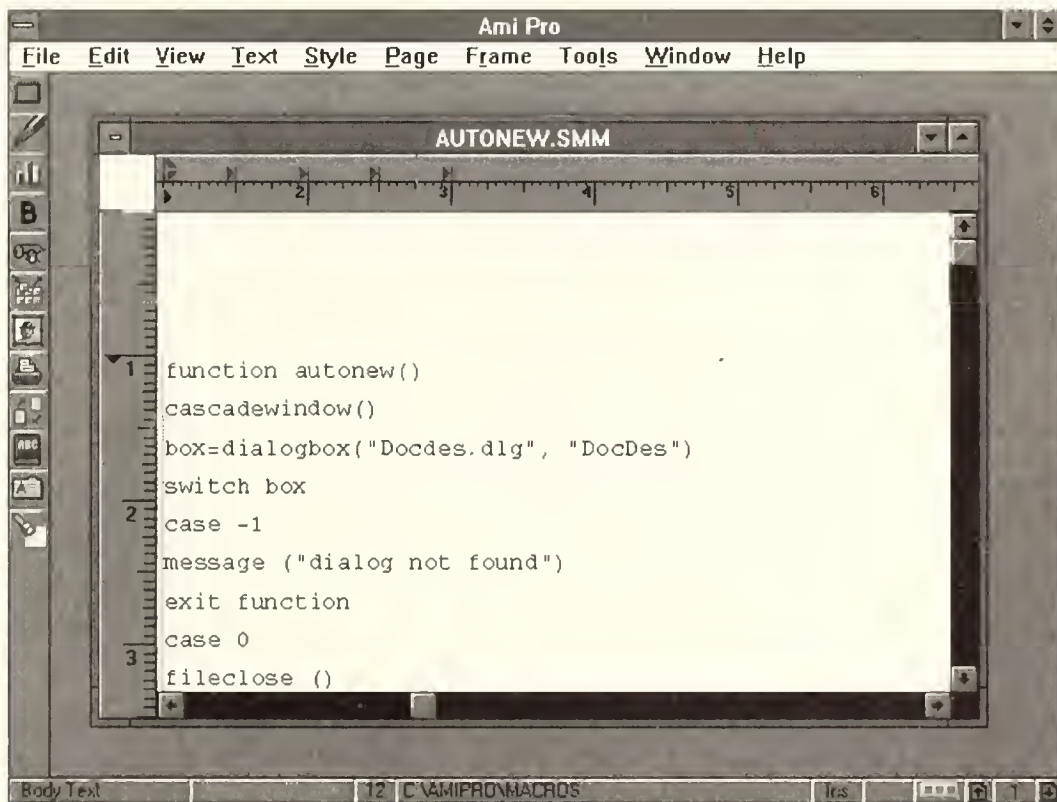
The Ami Pro macro language is a full-fledged development environment that allows you to completely customize every aspect of Ami Pro's operation—including its menus, dialog boxes, and basic functions. For instance, the DocMan application described in Chapter 14 employs an Ami Pro macro to modify the application's File New and File Save commands so that they automatically interact with the DocMan document management system.

The Ami Pro macro language includes a rich set of string, numeric, and file input/output functions, and offers support for DDE and for calling functions in external dynamic link libraries. The Lotus Dialog Editor, which is included in the Ami Pro package, simplifies the task of creating custom dialog boxes. And the program's macro recording feature can be used to turn keystrokes into macro code.

Macros are created and edited in Ami Pro using its standard editing tools, as shown in Figure 4.6, and then saved with an .SMM extension. The file is compiled as you save it. Ami Pro identifies any syntax errors in the file as you save it, and brings them to your attention at that time.

Figure 4.6

Macro editing in
Ami Pro



Most Ami Pro macro applications tend to execute quickly, but their execution speed still lags behind that of stand-alone applications. This macro language is best suited for developing applications that require automatic production and logging of documents or for customizing Ami Pro's menus and appearance.

Ami Pro for Windows Release 2.0
Lotus Development Corporation
55 Cambridge Parkway
Cambridge, MA 02142
(617) 577-8500
\$495

Crosstalk for Windows

Crosstalk for Windows is an asynchronous communications program that offers a wide range of terminal emulation and file transfer options.

The Crosstalk for Windows macro language is a subset of the powerful Crosstalk Application Script Language (CASL) from Crosstalk Mk.4 for DOS. CASL macros built for the DOS version can generally be brought into the Windows version, but because some CASL features are not available in that version, a fair amount of translation is necessary to get complex CASL scripts to run in the Windows version.

Despite compatibility problems with the DOS version, the Crosstalk for Windows implementation of CASL offers a rich set of functions for automating communications processes. The program includes a macro record feature, which simplifies this process. It falls down, however, in the area of user interface design facilities. You can create simple dialog boxes using Crosstalk for Windows, but you cannot create custom menus or utilize icons or list boxes in dialog boxes. This makes it difficult to create full applications in Crosstalk for Windows. You can, however, use the program's strong DDE support to link CASL scripts to a front-end application built using a product that offers more robust user interface development features, such as Visual BASIC or Toolbook from Asymmetrix.

Crosstalk for Windows 1.2
Digital Communications Associates
1000 Alderman Drive
Alpharetta, GA 30202
(800) 348-3221
\$195

dbFast/Win

dbFast/Win is a dBASE-compatible database language and compiler that has been extended to offer support for Windows features such as dialog boxes and buttons.

Because the dbFast/Win language is compatible with the dBASE language, it can be used to recompile existing dBASE applications for use in Windows, or to write entirely new applications in dBASE code.

This compatibility with the thousands upon thousands of existing dBASE applications is the product's strong point. Experienced dBASE programmers will be able to add Windows-specific features to their existing applications with only a modicum of effort. These features include support for DDE and external dynamic link libraries.

dbFast/Win is less suited for original development of Windows applications since it supports only a subset of Windows' user interface features. The program offers decent support for using multiple fonts and colors, building custom menus, and creating applications that utilize multiple windows. However, its dialog box creation capabilities are limited to list boxes, text messages, and

a variety of buttons. Advanced features such as icons and combo boxes are missing, and there is no dialog editor to simplify screen layout tasks.

dbFast/Win 1.55

Computer Associates International, Inc.

711 Stewart Avenue

Garden City, NY 11530

(800) 645-3003

\$199


DynaComm

DynaComm is a Windows asynchronous communications program that offers a full range of terminal emulations and file transfer protocols. Additional versions of DynaComm offering support for the Macintosh and for synchronous communications are also available.

DynaComm's macro language is among the richest available for any Windows application. It includes strong support for creating menus and dialog boxes, using script language commands like those in Figure 4.7. DynaComm's macro language also excels at utilizing Dynamic Data Exchange and external DLLs, creating random- and sequential-access text files, and automating the communications process. Its recorder function can be used to turn recorded keystrokes into macro code.

Figure 4.7

Macro editing in
DynaComm



```

DynaComm - AUTOMCI.DCP
File Edit Search Settings Phone Transfers Script Window Help
*main
if iconic() goto main
dialog (,4,306,176)
Message (4,4,,) " Account: " | $account
listbox (2,24,258,112) %table %i

message (82,4,,) "Sort by:"
listbox (110,4,50,44) 11 0 combobox set %sort listbox(2), if %sort>0 perform

message (172,4,,) "View:"
listbox (192,4,50,78) 10 %table combobox set %table listbox(3), perform dialo

widebutton (2,140,129,16) Default "&Read Message" dialog cancel, set %i
widebutton (131,140,129,16) "&New Message" dialog cancel, set %init 0,
button (1,156,43,16) "&Answer" dialog cancel, set %i listbox(), set %an
button (44,156,44,16) "&Move" set %i listbox(), record read %table at %
button (88,156,43,16) "&Export" dialog cancel, set %i listbox(), perfor
button (131,156,43,16) "&Forward" dialog cancel, set %i listbox(), set
button (174,156,43,16) "&Delete" set %i listbox(), perform delete, if %
button (217,156,43,16) "&Print" dialog cancel, set %i listbox(), perfor

button (263,24,40,16) "Send/Rec&v" dialog cancel, set @s8 str(-1), set
button (263,40,40,16) "A&utoMCI" dialog cancel, set @s8 str(0), set %in
button (263,56,40,16) "&Terminal" set %init 0, set @s7 "", perform save
button (263,72,40,16) "&Set up" set %i listbox(1),dialog cancel, set %u
button (263,88,40,16) "P&honebook" set %i listbox(1), dialog cancel, se
button (263,104,40,16) "Stat&istics" set %i listbox(1), dialog cancel,

```


The biggest limitation in designing applications in DynaComm is its lack of a dialog editor. The DynaComm language supports almost every Windows user interface control (multiple selection list boxes are the only notable exception), but creating complex dialog boxes without the help of a dialog box editor can be trying. Fortunately, a public-domain dialog box editor is available for DynaComm (DCDLGED.EXE) through the Future Soft Engineering forum on CompuServe, and the company is working on a dialog box editor for inclusion in future versions of the program.

DynaComm does include a script editor and a compiler, and applications created using them execute quickly. The DynaComm language is well suited for automating any communications task.

DynaComm 3.0
Future Soft Engineering, Inc.
1001 South Dairy Ashford, Suite 101
Houston, TX 77077
(713) 496-9400
\$295

Lotus 1-2-3 for Windows

As its name suggests, Lotus 1-2-3 for Windows is the Windows version of Lotus 1-2-3, the perennially best-selling spreadsheet for DOS computers. It makes good use of the Windows environment while providing the full range of functions and capabilities users of the DOS version have come to expect, including the ability to work with multiple worksheets at once, solver capabilities, and integrated graphics and database functions.

1-2-3 for Windows's macro language, unfortunately, is not nearly so close to the state of the art. It duplicates all the capabilities of the DOS product's macro language and adds a smattering of Windows-specific functions, including support for DDE and for the Windows Clipboard. However, it fails to include any specific user-interface facilities, and there is no way to create dialog boxes using the 1-2-3 for Windows's macro language. The only way to create custom menus is in the 1-2-3 Classic menu bar—a holdover from the DOS version that fails to follow standard Windows conventions.

Beyond its user-interface design limitations, 1-2-3 for Windows's macro language suffers from the program's insistence that macros be stored within the active worksheet. 1-2-3 doesn't recognize the concept of *global* macros—macros stored on disk for use with every worksheet.

Despite these drawbacks, the 1-2-3 for Windows's macro language is a powerful tool for automating spreadsheet functions and reducing repetitive tasks to a single keystroke.

1-2-3 for Windows
Lotus Development Corporation
55 Cambridge, MA 02142
(617) 577-8500
\$595

Microsoft Excel

Microsoft Excel is a powerful spreadsheet with integral graphics and database functions. It allows you to open multiple worksheets at once and embed graphics and command buttons on your worksheet. It includes strong support for Dynamic Data Exchange and external DLLs.

The Excel macro language provides the ability to completely automate and customize any spreadsheet function. It supports user-defined menus and dialog boxes, and allows macro programmers to completely mask the underlying spreadsheet by eliminating row and column headings, grid lines, and the formula bar. The program's macro recorder can be used to turn key-strokes into macro code.

Excel macros are stored in special *macro sheets*, which look much like a standard worksheet and can be edited using the program's standard formula editing commands. (A macro sheet is shown in Figure 4.8.)

Figure 4.8

A macro sheet in Excel



This method of creating and storing code has the benefit of being familiar to those who are comfortable with spreadsheets, but it is still awkward compared to a standard full-screen editor such as DynaComm. And it becomes even more so when you're editing menus or dialog boxes, although the latter task is simplified by the program's dialog editor.

Overall, however, Excel's macro language is well suited to automating any spreadsheet task and for building applications that require its strong calculating, graphics, and database functions.

Excel
Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052
(800) 426-9400
\$495

Microsoft Word for Windows

Microsoft Word for Windows is a powerful, professional-level word processor that includes spelling and grammar checkers, a thesaurus, revision marking, drawing, charting and image-processing modules, strong mail merge capabilities, the ability to embed command buttons in documents, DDE and DLL access, and a host of other advanced features.

WordBASIC, the Word for Windows macro language, utilizes a BASIC-like syntax to provide the macro programmer access to customization capabilities. The language can access any of Word for Windows' functions, and allows customization of the program's menus, dialog boxes, and basic functions. A macro recorder can be used to turn repetitive keystrokes into simple macros.

WordBASIC macros are edited within Word for Windows, which provides a set of special-purpose tools for stepping through and debugging macros. The editing screen is shown in Figure 4.9.

The WordBASIC dialog editor, new in version 2.0 of the program, simplifies the task of creating dialog boxes in WordBASIC. WordBASIC is well suited to any task requiring automated document production, or for customizing Word for Windows.

Word for Windows 2.0
Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052
(800) 426-9400
\$495

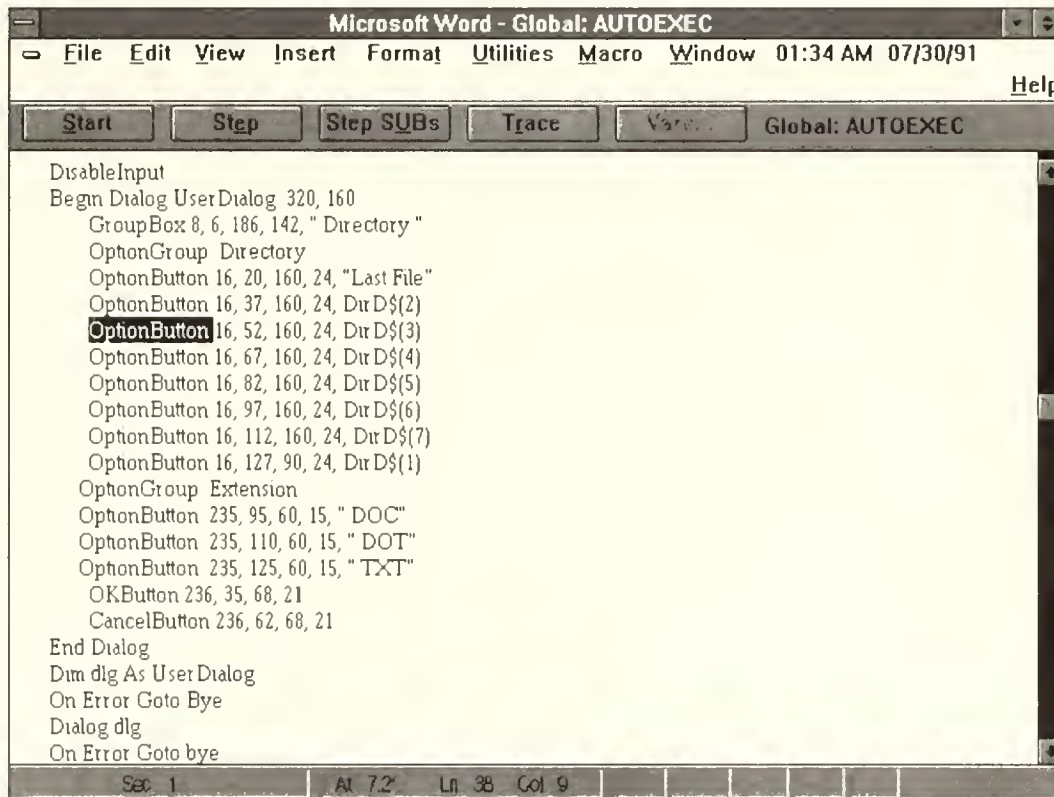
Object Vision 2.0

Object Vision is a unique cross between a forms automation package, an expert system development tool, and an object-oriented database language. It provides a powerful set of graphic tools that enable even novice programmers to turn out finished applications quickly.

Object Vision can read and write to files created by Paradox, dBASE and Btrieve, and can interact with dynamic link libraries. It also supports DDE.

Figure 4.9

A macro in Word for Windows



Object Vision doesn't provide an actual programming language. Instead, the process of developing an application in Object Vision starts with designing a screen form using a set of graphic screen-design tools. Then formulas are entered to define the values of each field on the form, using a combination of a graphic decision tree and spreadsheet-like @ functions. Similar methods are used to specify the actions to be taken (such as a dialog box opening or another form being loaded) when a button is pushed or a form is closed.

Object Vision is pretty much limited to producing forms-based database applications, and its lack of a programming language further limits its customizability. Nevertheless, the speed and ease with which you can create a database application in Object Vision make it a winner.

Object Vision 2.0
 Borland International, Inc.
 1800 Green Hills Road
 Scotts Valley, CA 95066
 (408) 438-5300
 \$495

SuperBase 4

SuperBase 4 is a full-featured relational database management system that includes strong support for graphics, DLLs, DDE, telecommunications, and

SQL (structured query language) queries. The program utilizes a unique visual design tool for defining relational links between data files, plus an overly cute but surprisingly effective series of VCR-like buttons for maneuvering through databases. The program can read and write dBASE files and can interact with a variety of SQL servers, including Oracle, Microsoft SQL Server, and SQLBase.

DML, the SuperBase 4 macro language, is a rich development tool for developing database applications. It includes good support for designing data input screens and reports and for automating any database management function. Using DML you can create multiple-window applications complete with text and graphics fields, multiple fonts and colors, and formatted reports.

Overall, SuperBase 4 is a powerful environment for building relational database applications that take full advantage of the Windows user interface.

SuperBase 4

Precision Software Inc.

c/o Software Publishing Corporation

3165 Kifer Road

Santa Clara, CA 95051

(800) 562-9909

\$695

Windows Batch Languages

The term *Windows batch languages* is a bit of a misnomer. It comes from the resemblance of the syntax of these languages to that of the DOS batch language and from their suitability for the same kind of “quick and dirty” programming. But, given the differences between programming for DOS and for Windows, the similarities are actually fairly superficial.

Like their DOS counterpart, these languages utilize a relatively simple syntax, making it easy to develop applications. But whereas the DOS batch language is best suited for simple tasks such as automating the loading of several TSRs prior to loading an application program, Windows batch languages are designed to automate processes involving multiple Windows applications and to build utility applications or front ends for DOS utilities.

These languages excel at manipulating other Windows applications by resizing, hiding, or showing their windows; sending them keystrokes; and cutting, copying, and pasting their data. Such languages also offer the ability to build dialog boxes to accept user input and to launch DOS applications with user-specified command line parameters. (However, with the exception of Bridge Toolkit, they cannot interact with DOS applications once the DOS application has been launched.)

Windows batch languages also all offer the ability to record keystrokes and then translate those keystrokes to macro code, thus simplifying the process of developing applications.

The strongest points about these languages are the speed with which you can build small applications or application prototypes and their ability to link multiple Windows applications. Their weakest point is their lack of functional depth—you wouldn't want to call on any of them to perform extensive calculations or manipulate large data files.

You'll quickly run into the limits of these languages if you try to develop full-scale applications with them. But if you confine your use of them to the tasks to which they are best suited, you'll discover that they are invaluable tools for producing simple applications quickly and with minimum effort.

Ideal uses for these applications include building Windows front ends for DOS utilities such as file compression/decompression programs and creating menu systems that automatically set up your Windows workspace, loading sets of programs with a single command. They can also be used to automate processes in Windows applications that lack internal macro languages and to automate time-consuming or repetitive tasks.

Bridge Batch

Bridge Batch is a remarkably full-featured development tool that includes a powerful language, support for DDE and dynamic link libraries, and excellent dialog box and program editors.

The Bridge Batch language is geared toward automating processes involving other Windows applications. It includes a full complement of commands for launching, controlling, and exchanging data with other applications, plus a strong set of user interface design tools that allow you to build menu systems or dialog boxes for controlling those processes; an example is shown in Figure 4.10. It doesn't offer the data processing power necessary for stand-alone application development, but it has tremendous capability as a tool for linking multiple Windows applications into integrated systems.

The Bridge Batch macro recorder is a well-designed complement to the language. It can record both mouse and keystroke commands. Mouse commands recorded in this fashion cannot be converted to macro code or edited, but they can be assigned to hotkeys or called from macro programs.

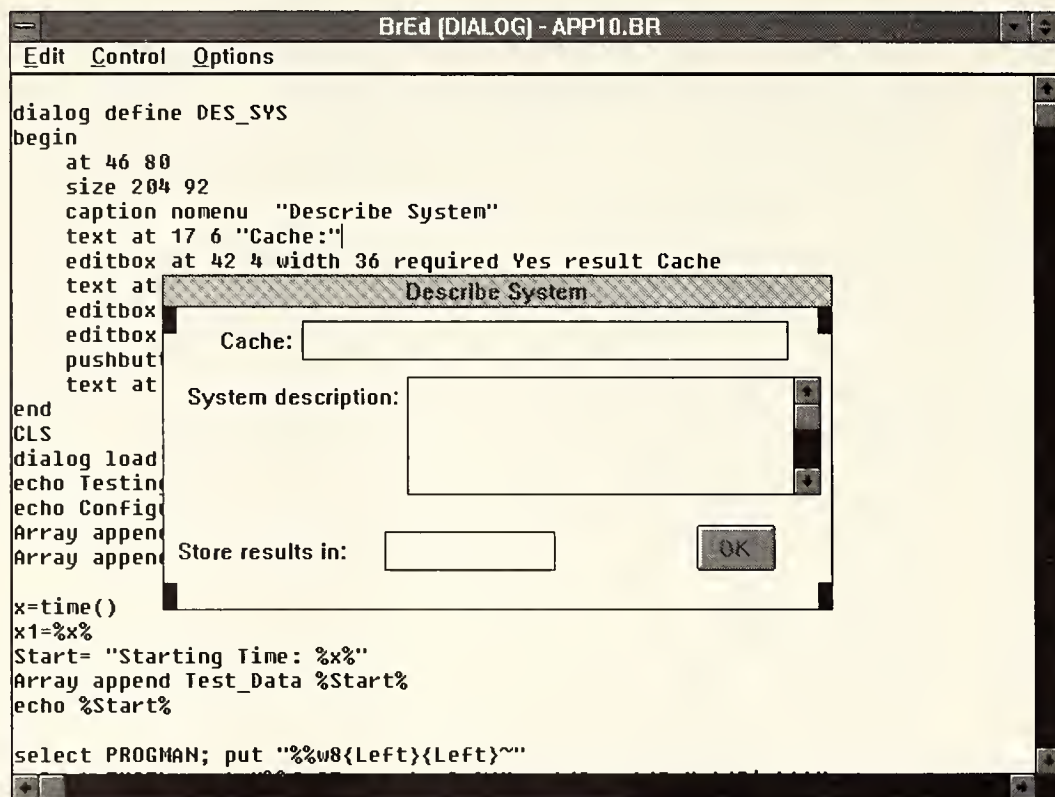
Bridge Batch
Softbridge, Inc.
125 Cambridge Park Drive
Cambridge, MA 02140
(617) 576-2257
\$179

Bridge Toolkit

Bridge Toolkit is a superset of Bridge Batch that incorporates several additional capabilities, including the ability to interact with DOS applications running in 386 enhanced mode and to communicate across a local area network (LAN) with other Bridge Toolkit applications.

Figure 4.10

The Bridge Batch editor



Bridge Toolkit's ability to interact with DOS applications is unique among Windows development tools. To make use of it you must load a small TSR before loading Windows. Bridge Toolkit utilizes this TSR to monitor the activity of DOS applications and send them keystrokes. For instance, you might have a Bridge Toolkit program that interacts with Lotus 1-2-3, sending it first the command to load a file and then a series of commands to manipulate that file, waiting after each command for 1-2-3 to indicate that it is ready to receive another command string.

The network capabilities of Bridge Toolkit are also unique. Using this capability, you can easily build a LAN-based messaging system, or automate tasks that are to take place on multiple workstations around the network.

A runtime version of Bridge Toolkit is available on a per user basis for distribution with Bridge Toolkit applications.

Bridge Toolkit
 Softbridge, Inc.
 125 Cambridge Park Drive
 Cambridge, MA 02140
 (617) 576-2257
 \$695

Norton Desktop for Windows

Norton Desktop for Windows is a powerful collection of utilities for Windows that offer the user a tremendous range of tools for customizing the appearance and operation of the Windows environment itself. It includes replacements for Program Manager and File Manager, an enhanced drag-and-drop desktop interface, and a variety of utility programs including a backup program, data-recovery utilities, a scheduler, and file-searching and viewing capabilities.

In addition, Norton Desktop includes a version of WinBatch (see below) that offers a powerful tool for automating repetitive activities or customizing Windows applications. Included with this is a useful editor for creating batch files, complete with a drop-down reference to all the commands and functions in the batch language.

The batch language can be used in conjunction with Norton Desktop's scheduler to create automated routines that run at regularly scheduled intervals.

Norton Desktop for Windows
Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
(408) 253-9600
\$149

PubTech BatchWorks

PubTech BatchWorks is the commercial version of Wilson WindoWare's WinBatch (described below). Both products offer a macro recorder, the ability to attach new menu items linked to macros to the Control menu of any Windows application (the menu that appears when you click the gray box at the upper-left corner of any window), and strong capabilities for interacting with Windows applications.

The ability to add new functions to the Control menu of other Windows applications gives BatchWorks unique strength as a tool for customizing other applications.

BatchWorks also offers the unique ability to read the titles of dialog boxes displayed by other Windows applications. (In contrast, other Windows development tools are generally limited to reading the title of the main window of another application.) This enables you to build a higher level of error checking and control into batch programs that automate other applications because you can check whether the application has displayed an error message in a dialog box in response to previous commands you have sent it through the batch program.

Unfortunately, BatchWorks falls short in the area of user interface design. It includes neither a program editor nor a dialog editor, although a program editor is available in the BatchWorks SDK, a companion product. Any ASCII editor (such as Notepad) can be used to create batch programs,

but the lack of a dialog editor is a serious shortcoming. In addition, its dialog boxes can utilize only a brief subset of the standard Windows controls. They are limited to displaying list boxes, buttons, check boxes, text messages, and radio buttons.

Other shortcomings of the product include the inability to create new drop-down menus (although this is obviated somewhat by the ability to add items to the Control menu) and the product's lack of support for DDE. Nevertheless, BatchWorks is a valuable tool for building quick utilities or front ends for DOS utilities and for controlling the operation of, or customizing, other Windows applications.

PubTech also offers the \$99.95 BatchWorks SDK, which can compile batch files into stand-alone .EXE programs or encrypt and password protect them to prevent unauthorized alterations. Encrypted or password-protected batch files can be used with the \$20 runtime version of BatchWorks. The BatchWorks SDK includes the PubTech Text Editor, a Windows-based program and ASCII file editor.

PubTech BatchWorks 2.0
Publishing Technologies, Inc.
7719 Wood Hollow Drive, Suite 260
Austin, TX 78731
(800) 782-8324
\$99.95

WinBatch

WinBatch is a shareware version of PubTech BatchWorks. The primary difference between the two products is the difference in their price—which might be reason enough to opt for the less expensive shareware version.

WinBatch was used to produce two of the projects in this book: The Ultimate Notepad project in Chapter 9 and the AutoPrint project in Chapter 11. We've included the latest version of WinBatch on the program examples disk that accompanies the book, to enable you to test these projects right away. You are, however, obligated to register WinBatch with Wilson WindowWare and pay its shareware fee if you continue to use the program beyond a short trial period.

In addition Wilson WindowWare offers a compiler for WinBatch that can be used to turn batch programs into stand-alone .EXE files, so that the WinBatch program is no longer required to execute them.

WinBatch
Wilson WindowWare
2701 California Avenue S.W., Suite 212
Seattle, WA 98116
(800) 762-8383
\$60.95

Windows BASICs

Not long ago, BASIC was considered as dead as Latin. It might have been the first programming language most DOS users learned, but it was the last they would ever want to use for serious application development. Then a funny thing happened on the way to Windows: BASIC grew up.

The implementations of BASIC for Windows don't have very much in common with the ugly language you may remember. BASIC's line numbers have been replaced with named subroutines, and its GOTOs and GOSUBS have abdicated to a rich set of program control structures. Moreover, most implementations of BASIC for Windows allow you to compile your program to an .EXE file that, as long as you have the language's runtime DLL available on your system, can be used without the BASIC development system or interpreter. This means that you can create small, speedy .EXE files. Of course, the runtime DLLs tend to be rather large (about 260k in the case of Visual BASIC), but the same runtime DLL is used by every .EXE file you create. So although the first program you create might require 300k of disk space, including the DLL, each additional one might only occupy 10 to 50k, depending on the complexity of the program.

Gone too are the ugly teletype screens produced by most BASIC programs. In their place, the Windows implementations of BASIC offer you access to the full set of Windows controls for creating attractive, easy-to-use user interfaces.

The strongest point about these implementations of BASIC is that BASIC remains an easy programming language to learn. Compared to C, or even Pascal, BASIC code is almost readable, and given sufficient documentation, one can follow the flow of a BASIC program's logic relatively easily.

However, compared to an application macro language, these are still fairly low-level programming languages. If your application needs a sort routine or a text-search routine, you'll have to write it yourself; you can't simply call the application's sort or search function as you could with Ami Pro or Excel or another application macro language. So considerably more programming effort may be required to achieve the same result, compared with a macro language. On the other hand, these languages allow you to create applications that aren't dependent on the limitations of another program. They can also be distributed without also having to distribute the huge application programs upon which application macro languages are based.

GFA-BASIC for Windows

GFA-BASIC for Windows is a structured implementation of BASIC that provides the unique benefit of allowing *cross-systems* development of applications for both DOS and Windows. Applications written in the Windows version of GFA-BASIC can be recompiled in the DOS version of the language. The DOS version will retain the basic windowed appearance of your application and

duplicate all but its most Windows-specific features (such as Clipboard support and DDE) for use under DOS. In addition, the manufacturer is promising to deliver OS/2 Presentation Manager and UNIX versions of the language in the near future, and claims that these will offer the same degree of compatibility.

One might expect that this cross-systems development capability would come at the expense of the language's ability to fully exploit the Windows environment. However, GFA-BASIC for Windows boasts a huge command set that provides access to every Windows function and takes full advantage of Windows' memory management capabilities. Thus you can, for instance, create arrays that occupy up to 20Mb of RAM.

The GFA-BASIC editing environment is less graphic than those of the other Windows BASICs. Most code is written in a traditional editor, although a graphic editor is provided for creating dialog boxes.

GFA-BASIC applications can be compiled to an .EXE file, or can be used with the program's royalty-free runtime package. The language offers support for the Windows multiple-document interface, and can access DLLs and DDE functions.

GFA-BASIC for Windows
GFA Software Technologies, Inc.
27 Congress Street
Salem, MA 01970
(508) 744-0201
\$295

ObjectScript

ObjectScript is built around a screen editor that works like a drawing program. You place user interface controls (buttons, list boxes, and so forth) on the screen by dragging them with the mouse. Then you write the code that will be executed when the user selects the item using ObjectScript's implementation of BASIC. Figure 4.11 shows the process of defining a table. This makes it simple to create attractive user interfaces for your application. The language supports DDE and use of external dynamic link libraries and has the ability to read and write dBASE, ASCII, and PCX-format files.

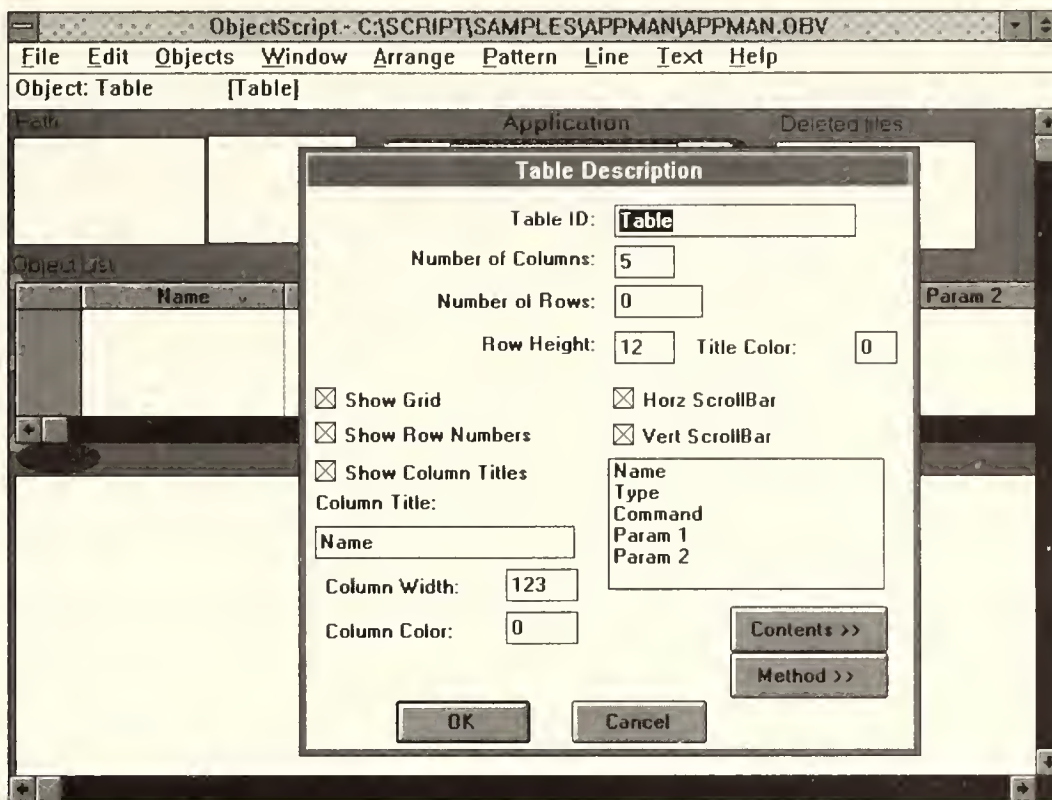
ObjectScript Professional is an enhanced version of the product. It features improved printer support, support for more external databases (including Novell's BTRIEVE), advanced keyboard-handling functions, time and date functions, and support for additional user interface controls and messages.

The ObjectScript runtime can be distributed without charge, but a one-time \$495 fee is required for the ObjectScript Professional runtime.

ObjectScript is useful for building graphical front ends to dBASE files and for building small utilities. For more ambitious projects you'll need the additional functions and capabilities of ObjectScript Professional.

Figure 4.11

The ObjectScript editor



ObjectScript and ObjectScript Professional
 Matesys Corporation
 900 Larkspur Landing Circle, Suite 175
 Larkspur, CA 94939
 (415) 925-2900
 \$150 and \$495

ObjectView

ObjectView is a superset of ObjectScript Professional that adds a set of high-level SQL functions for interacting with Microsoft's SQL Server and Gupta Technologies' SQL Base and Oracle and DB2 databases. It can be used to develop SQL applications with remarkable speed and ease.

ObjectView
 Matesys Corporation
 900 Larkspur Landing Circle, Suite 175
 (415) 925-2900
 \$899

Realizer

Realizer is a highly structured superset of BASIC that combines a visual screen-design tool with a powerful language. It includes a set of high-level objects—including spreadsheet, charting, and text editing windows—that speed the development process by allowing you to include a spreadsheet, chart, or text editor in your application with a single line of code.

Realizer produces fast, efficient applications, and includes support for the multiple-document interface, DDE, dynamic link libraries, and user-defined custom controls, and the ability to read and write Lotus 1-2-3, dBASE, and Excel as user-defined file formats. Registered users can distribute the Realizer runtime DLL free of charge.

Realizer's screen-design tool, called FormDev, is somewhat clumsy to use, but provides access to the full range of Windows controls, as shown in Figure 4.12.

Realizer produces fast, efficient code, and its complement of high-level objects speeds the development process.

Realizer

Within Technologies

8000 Midlantic Drive, Suite 201 South

Mt. Laurel, NJ 08054

(609) 273-9880

\$395

Visual BASIC

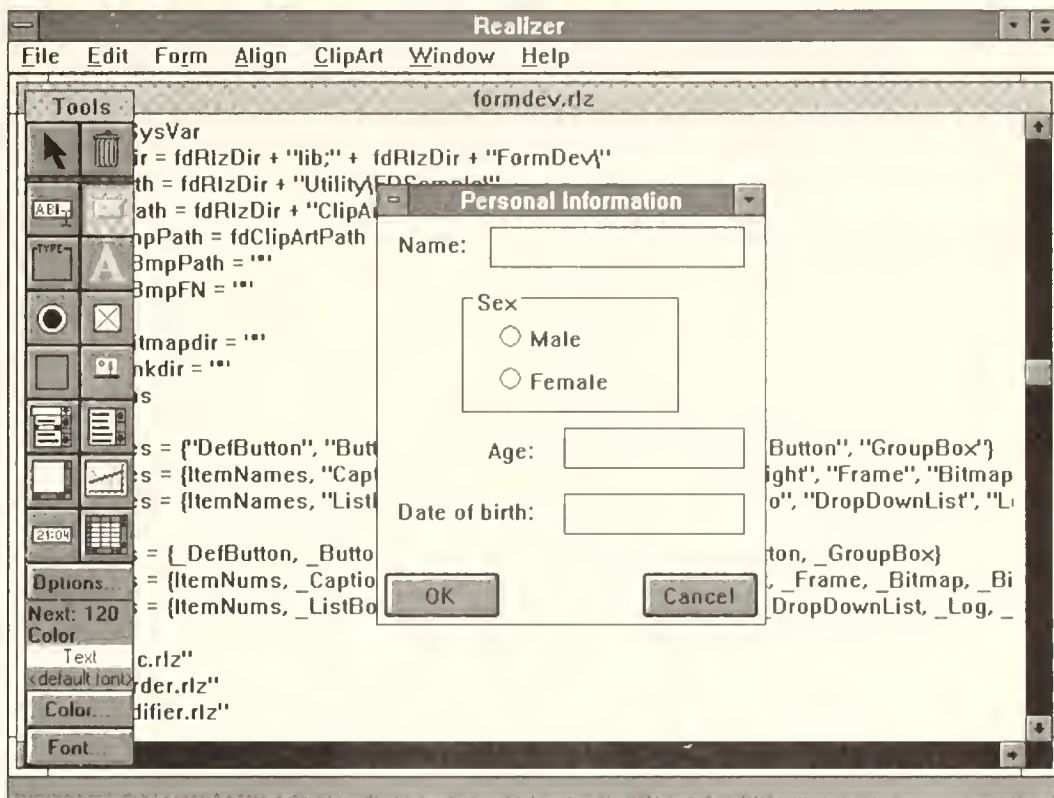
Visual BASIC is a superset of BASIC that takes its organizing principles from programs such as Apple's Hypercard. Program screens are created using a set of drawing program-like tools, and then code is attached to each object for each event the programmer wishes to have the program respond to. For instance, a button might have separate functions programmed for a mouse-down event, a mouse-up event, and a keypress event.

The compartmentalization of code engendered by Visual BASIC's event-driven programming model is both a blessing and a curse. On the one hand, it breaks the programming task down into a series of manageable steps: Each action and each process can be developed as a separate entity. But it also makes it difficult to get a grasp on your whole application because your program code is actually scattered through the application—hiding behind buttons and list boxes rather than appearing in a single stream, as is the case with the other Windows BASICs. This can complicate the development and debugging process with large applications, at least until you become comfortable with this new approach to programming.

Visual BASIC supports custom controls, Dynamic Data Exchange, and the ability to call external DLLs. There has been a tremendous amount of third-party support for Visual BASIC, resulting in the availability of a wide variety of custom controls and DLLs that can be incorporated into your programs.

Figure 4.12

Realizer's FormDev screen-design tools



Registered users can distribute the Visual BASIC runtime DLL freely.

Overall, Visual BASIC is best suited for small to medium-sized programming tasks. It is well suited for designing attractive, easy-to-use graphical applications, but may not be robust enough for large-scale development projects. (Although it is an excellent prototyping tool for applications of any size.) The standard version of the product does not support the multiple-document interface (which allows a program window to open several child windows at once and minimize or arrange their windows at will) but this capability is available in the Visual BASIC Professional Toolkit.

Visual BASIC was used to develop the projects described in Chapters 12 and 14. The Visual BASIC runtime DLL is included on the disk accompanying this book to allow you to see these projects in action. However, you'll need the full Visual BASIC development environment if you wish to modify or customize those projects.

Visual BASIC
 Microsoft Corporation
 1 Microsoft Way
 Redmond, WA 98052
 (800) 426-9400
 \$199

Pascal

Thanks in large part to the success of Turbo Pascal for DOS, Pascal has eclipsed BASIC as a tool for nonprofessional programmers developing DOS applications. Turbo Pascal for Windows, the only current implementation of the Pascal language for the Windows environment, may have the same effect.

Turbo Pascal for Windows

Turbo Pascal for Windows is a full-fledged object-oriented programming environment for creating stand-alone Windows applications. At the heart of the Turbo Pascal for Windows environment is a code library called ObjectWindows, which contains the rough framework for creating a generic Windows application. By modifying this framework, one can create Windows applications of any description with extraordinary speed.

Turbo Pascal for Windows applications can access DDE, dynamic link libraries, and the Windows Help engine. The Turbo Pascal for Windows package includes the Resource Workshop Toolkit (a set of tools for creating icons, windows, and dialog boxes), Turbo Debugger for Windows (a tool for testing and debugging applications), a resource and help compiler, the ability to create dynamic link libraries, and the ability to run standard DOS Pascal code in a Teletype-like window.

Combined, these features make Turbo Pascal for Windows at least the equal of Realizer and Visual BASIC as a tool for creating complete Windows applications from scratch. And unlike either of those products, Turbo Pascal for Windows programs compile to stand-alone .EXE programs that don't require the presence of a runtime DLL. Overall, Turbo Pascal for Windows is a rich environment for creating Windows applications of nearly any description. It provides access to every Windows function without your having to access the Windows SDK.

Turbo Pascal for Windows
Borland International, Inc.
1800 Green Hills Road
Scotts Valley, CA 95066
(408) 438-5300
\$249.95

Graphical Hypertext Products

The two products in this category are both closely related to Apple's Hypercard. Like Hypercard, they make it simple to design attractive user interfaces with hypertext links between data. They employ easy-to-learn programming languages and have good support for graphics and animation.

In graphical hypertext products, your program code is linked directly to the screen object (button, list box, and so on) that calls it. So if you have a

button that sorts a list box, you can copy that button to another screen in your application, and the code associated with it will be copied too. This is a real help to development because once you've created a few routines that work you can reuse them in any application you're developing in that language simply by copying their associated screen objects to the new application.

The appeal of these products for novice programmers is in the English-like syntax of their programming languages. For instance, a script written in Spinnaker Plus might contain the line

```
Put the TopLeft of Me Into MyPos
```

which would tell Plus to create a variable called MyPos and to store the coordinates of the top-left corner of the control containing the statement in the newly created variable.

However, these products also have limitations to their suitability for general-purpose application development. These include the massive size of the files they create when working with internal data and the relative weakness of their languages in areas such as calculation and heavy-duty data processing. You might be able to build a spreadsheet using one of these tools, but you wouldn't want to wait around for it to recalculate a thousand cells.

As a result, these products are best suited for building front ends for external databases, for multimedia presentations, and for application prototyping.

Spinnaker Plus

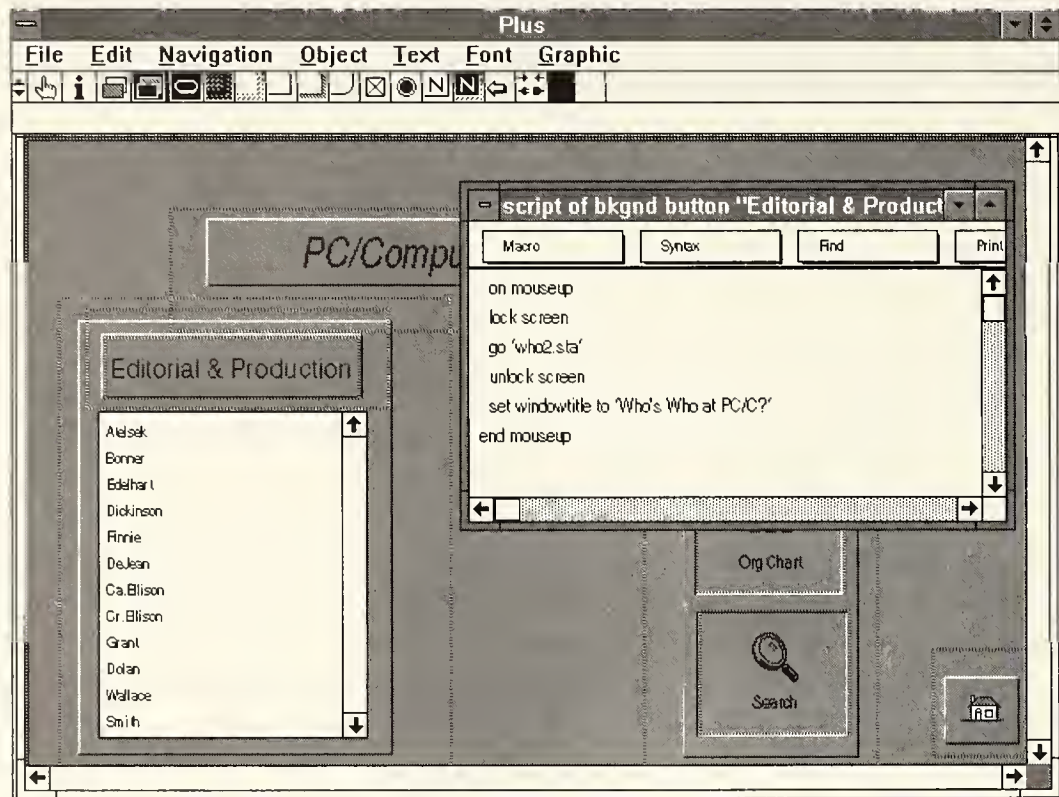
Spinnaker Plus is a superset of Hypercard that runs under Windows, on the Macintosh, and under OS/2 Presentation Manager, making cross-platform compatibility one of the product's strongest points. The Macintosh version can import Hypercard scripts, which can then be moved to the Windows or OS/2 version with very little difficulty. Spinnaker Plus can also access external database information through DDE or through database DLLs.

The Spinnaker Plus development environment features a strong script editor and easy-to-use graphics tools. You design the user interface of your application by selecting controls from a toolbar and then dragging them onto the screen. Then, when you're ready to write the script to be executed by that control, you can select programming statements from a drop-down menu, eliminating the need to type statements into the editor. Figure 4.13 shows the Plus programming environment.

A runtime version is available that can be distributed with Spinnaker Plus scripts for a one-time fee.

Spinnaker Plus is best suited for building graphical databases with a limited number of records, application prototyping, or building front-ends to data obtainable through DDE or through a dynamic link library.

Figure 4.13
The Spinnaker Plus
script editor



Spinnaker Plus v. 2.5
Spinnaker Software Corporation
201 Broadway Avenue
Cambridge, MA 02139
(617) 494-1200
\$495

Toolbook

Toolbook marries a graphical development environment, with good support for animation, to a strong set of database functions. The Toolbook OpenScript language offers a rich function set, and Toolbook can read and write .DBF files and obtain data through DDE and DLLs. As with Spinnaker Plus, you design Toolbook screens by selecting user interface items from a palette of available controls, and then write code for each control separately.

Toolbook's strengths are its graphics capabilities and its ability to access external data. It isn't particularly well suited for storing large databases, but it can be invaluable for building graphical front ends to external data and for multimedia presentations and application prototyping.

Toolbook 1.5
Asymmetrix Corporation
110 110th Avenue N.W.
Bellevue, WA 98004
(800) 624-8999
\$395

Principles of Application Design

The Nuts and Bolts of Application Design

Implementing a Windows Interface

Prototypes, Testing, and Documentation

PART

2

**The Application-
Development Process**

C H A P T E R

5

**Principles of
Application
Design**

*Bonner's
Usability Guidelines*

WHAT DO YOU DO FOR A LIVING? ARE YOU A SALES REPRESENTATIVE? An entrepreneur? An accountant? A securities analyst? A physician? Well, enjoy it while you can. Your days in that profession may be numbered. As soon as you start building a computer application, you're in danger of becoming a software designer. Maybe you won't identify yourself as such on your Form 1040, but you'll start to think like one—which can be a terrible thing. Why? Because a software designer doesn't know half as much about what an accountant or physician or entrepreneur needs as the accountants, physicians, and entrepreneurs for whom he or she is writing the application.

Here's how you can tell if you've undergone this mutation: If you started out as a sales representative and you still think that the job of a sales rep is to sell, then you're still a sales rep. But if you think the job of a sales representative is to run sales-support software, you've turned into a software designer—and you've probably lost the ability to write useful sales-support software.

There is only one way to prevent this as you start writing applications, whether they are for your own use or others': You've got to remember, with every line of code you write, what life was like before you started writing software. Remember how you used to spend your time, and remember the things that used to matter to you. Above all, remember that to people who aren't professional software designers, computer applications are a means to an end, not an end in themselves. Your application should be as functional, unobtrusive, and dependable as the knob on your office door.

Of course, you might still turn into a software designer even if you do follow this advice. But at least you'll be a good one.

Designing for the Designer

Over the past ten years, I've worked with, reviewed, or seen demonstrations of at least 5,000 commercial PC software applications. By any measure—sales, market share, user loyalty, longevity—between 90 and 95 percent of these products eventually proved to be failures.

Although the factors that determine a commercial software product's success or failure are as complex as the weather, almost all those that failed shared at least one flaw: They were designed for the designer, not the user.

The difference is a matter of hubris—that for which the gods (or market) punishes one.

For example, many years ago an industry luminary who shall remain nameless told me that, although his new product did require significantly more effort to master than the current market leader, "It is an effort for which the user is amply rewarded."

That statement was pure balderdash since the judgment wasn't his to make. The decision whether the effort required to master a software package

justifies the result is entirely the purview of the user. And in most cases, smart users will respond to that sort of attitude with a statement on the order of “Not ample enough by a mile, Buster!”—as, in fact, they did with the product in question.

An application designed for the designer assumes that the designer knows more about the user’s job than does the user. It forces the user to conform to the designer’s theories, or to the program’s internal conventions and methods, rather than the program adapting to fit the user’s needs. The assumption is made that the application occupies the central role in the user’s work life—even that the user exists primarily to serve the application, rather than vice versa.

Fortunately, it’s fairly easy to spot these arrogant pretenders. For instance, an application that can’t be described without using a metaphor has almost always been designed for the edification of the designer. The user shouldn’t have to think about software as a notebook, a desktop, or a collection of index cards in order to understand it—the user shouldn’t have to think about good software at all. Also beware of applications that claim to “establish a new paradigm” or promise to “change the way you work”—you may not need, or want, either one.

Designing for the User

In contrast, the applications that over the years have proven to be successful show every sign of having been designed for the user. They provide efficient, responsive, and above all, useful services, without placing untoward demands on the user. They don’t presume to tell you how to go about your job, and they don’t require that you take a course in metaphysics before you can divine their basic concepts.

An application that has been designed for the designer makes you think too much about the process of using the application. In contrast, one that has been designed for the user clarifies the process, so that you can concentrate on the job that motivated you to pick up the application in the first place.

Of course, it’s all well and good to talk about designing for the user or designing for the designer, but what does it really mean? Aren’t I, to a large extent, playing Monday-morning quarterback here—looking at the programs that I consider successful and decreeing that they were “designed for the user”?

Yes and no. Certainly I don’t know of any software designer who deliberately sets out to make software that is not accessible to the user. Just as, I imagine, there are few fashion designers who deliberately set out, as is so often alleged, “to make women ugly.” But just as there are classic definitions of elegance to which some fashion designers might attend (whereas others are driven by more garish impulses) so too is there a set of general guidelines

that software designers can follow in order to ensure the usability of their applications.

Of course, it's also instructive to remember that "software usability," like "fashion elegance," is subjective. Some people might think of Madonna when they hear the word *elegance*, and others of Lauren Bacall. Similarly, some computer users and application designers might think of WordStar's control-key sequences as the ideal command interface for any application, whereas others might prefer the Windows user interface. No one design will satisfy everyone.

Nevertheless, beyond the allowances that one must make for personal preference, there is empirical evidence that some user-interface designs work better than others. The user-interface elements found in Windows 3.1, for instance, are the result of extensive laboratory tests (at Xerox's Palo Alto Research Center, Apple, IBM, and Microsoft) and years of experimentation in various commercial products.

However, just because an application runs under Windows 3.1 doesn't mean that it is truly usable. I've seen applications developed for 40-column Apple II displays that were more intuitive and simply more *right* than some Windows applications. You can design an oppressive, unintuitive, clumsy application under Windows, just as you can with any other development environment. It simply takes a little more effort to do so.

The fact that you'll be building applications under Windows doesn't guarantee that the applications you design will be truly usable. Nor, unfortunately, does any other single step that you can take. Including Help screens doesn't do it. Supporting both mouse and keyboard users doesn't do it. Using a pastel color palette doesn't do it. In fact, there is simply no magic formula to designing for the user.

However, there are some general principles you can follow that will at least help you avoid the big traps. So, in lieu of a magic formula, I offer Bonner's Usability Guidelines, a set of ten general principles that define applications designed for the user.

Bonner's Usability Guidelines

The guidelines that follow are a mixture of common sense and observations I've made over the years about the qualities shared by all successful applications. They're also influenced by my obviously strong bias in favor of the user: As far as I'm concerned, if a design decision doesn't make life easier for the user, then it wasn't justified. Period.

You'll notice an almost-contradictory tension between some of these guidelines, on the order of "The porridge has to be hot enough, but the porridge can't be too hot." Good application design is a dialectic—a process in which one has to measure these tensions and keep them in balance at every

step. The most classic example of this is “ease of learning” versus “ease of use.” On the one hand you want to make your application easy for new users to learn, but on the other hand you don’t want to slow down or frustrate experienced users.

So how do you balance the two? Very carefully. (Actually, you balance them by prototyping and testing your application and judging users’ reactions to it, as described in Chapter 8. But for now, let’s just say that you do it very carefully.)

These guidelines apply primarily to applications that you develop for use by others, rather than quick little utilities you cobble together just for yourself. You don’t need to think very much about usability when you’re designing an application for your own use—if you don’t like the way it works you can always change it. But when other users are involved, usability becomes a much more complex issue. Many a programmer has been shocked to find that an application that works exactly as expected, and solves precisely the problem he or she thought needed solving, is all but unusable in the hands of any other individual. It’s an awful revelation—one I hope you never encounter.

The following ten rules will help you avoid any such nasty discovery:

- Fit applications into the current work flow.
- Improve on existing methods.
- Don’t surprise the user.
- Try to delight the user.
- Finish the job.
- Make applications open-ended.
- Design for reliability.
- Don’t overwhelm new users.
- Don’t delay experienced users.
- Above all, design for the user’s convenience—not your own.

Let’s look at each of these rules in detail.

Rule 1: Fit Applications into the Current Work Flow

How many times have you heard the promise “This application will change the way you work?”

That promise always sounded more like a threat to me. PC users aren’t necessarily looking for a new gestalt every time they walk into a software store. Often they’re simply looking for a small application or utility program

to replace an equally small, less-efficient piece of their work life—a spelling checker to replace a spelling dictionary, for example. In such cases, it should do its job smoothly, without disrupting the rest of the user's routine. It shouldn't add additional steps to the process just to accommodate its internal requirements.

That's the ideal, but it is inevitable that most new applications will require limited retraining, a change in routine, or additional steps. Such is the price of progress. But the more you are able to minimize this sort of disruption, the more easily your application will be accepted. So, as you design an application, think twice about any change that will disrupt the current work flow, try to find a way to avoid the disruption, and, if it can't be avoided, make sure the results justify it.

Other times a program plays a larger role—replacing not a single step, but rather an entire system or process. In such cases, the software should indeed change the way you work. However, such cases are few and far between, occurring primarily in situations where a grossly outdated and inefficient system is crying out for replacement. And they require extensive study and planning before you attempt to implement a solution. The other situation—one in which an incremental productivity gain can be made by adding a new application to the process, without disrupting its other components—is far more common, and much more likely to be solvable by means of a quick application-development project.

Rule 2: Improve On Existing Methods

As important as it is that an application fit within the current work flow, just fitting in isn't enough. If a new application doesn't improve on existing methods, then there's no reason to use it.

Take, for example, a form-based data entry application. If users have to enter every bit as much information into the computerized version of a form as they do on the paper version—and have just as much chance of making a mistake in the process—there is little reason for them to prefer the computerized version. But if the computerized form takes advantage of the PC's capabilities to fill in dates, provide a list of customer numbers, and automatically enter the customer's address and billing status, then it is certainly an improvement on the old method. And users will welcome the new application because it speeds the process of filling out the form and reduces the opportunity for errors.

Of course, there are all kinds of ways of improving on existing methods. The new application might be faster than the old one, or provide more useful data, or produce the same results at a lower cost.

At the same time, however, there are trade-offs involved in almost any new application. The new way of doing things may be faster, but may also require an extra step on the part of the user. Or it may provide more

comprehensive data, but at the cost of slower execution. Or it may require that the user learn a new set of procedures.

It would be wonderful if the applications you develop were better in every part of the process, but this is seldom the case. Instead, in most cases you'll have to measure the trade-offs involved to ensure that the net result is positive.

In other words, it's important to ensure that the benefits justify the cost—both from the standpoint of your organization and from that of the individual end user. A common mistake in developing applications is to consider only the cost or benefit to the organization, ignoring the impact of the new application on the user. The gain to the organization is very important, of course, but your application won't be successful unless the user sees it as an improvement as well. But how do you determine what is an improvement from the user's standpoint?

A colleague once revealed to me what he considered the cardinal rule for writing mainframe terminal applications: "You can do anything you want to the system, as long as you don't increase response time. Users won't stand for anything that lengthens response time." Even in an environment as unfriendly as a mainframe terminal system, where users were already used to a 15-second delay between the time they pressed the Enter key and the time they got a response from the system, there were unwritten usability rules in effect.

In the PC environment, the user's expectations are much higher, so there are more usability rules to consider, which makes weighing the trade-offs more complicated. Specifics will vary from application to application, but you can use the following generalizations:

- You can do anything you want, but don't make the user's job more complicated.
- If there is no way to avoid increasing the complexity of a process, make sure your application also offers the user a tangible, compensating benefit.

Overlooking either of these points almost guarantees that you'll end up with unhappy users, and the anticipated benefits to your organization will surely go up in smoke.

Rule 3: Don't Surprise the User

In the right circumstances, surprises are great fun. For example, a surprise birthday party can be the stuff of a lifelong good memory.

On the other hand, a surprise gas main explosion is not much fun. And that's the kind of surprise that PC users have come to dread over the years—because a generation of badly written software has left them picking up after big booms far more often than after big parties.

Here's a simple example of a big-boom surprise: You've just spent five hours creating a complex document. Satisfied with your work, you pull down the application's File menu and select the Exit item, expecting the application—like all good Windows applications—to prompt you to save the file before shutting down. But it doesn't. Instead, it simply exits to Program Manager.

BOOM! Five hours work gone in a flash.

Makes you want to kill, doesn't it?

That kind of "Whoops, that didn't work the way I thought it would" explosion happened all the time with DOS applications, because they lacked any standard user-interface guidelines. It's less common with Windows applications—at least on a scale this heinous. But there are lots of less catastrophic but still annoying cases of Windows applications not acting the way the user expects them to, and thus raising the user's irritation level.

For instance, take Lotus Notes version 2.0. Notes has been highly lauded for the new ground it broke as a group communications application. But it also deserves a Tin Turkey award for its flagrant violations of standard Windows user-interface conventions. The standard says that if you double-click on a word in a Windows-based editor, the word should be highlighted. But nothing happens when you double-click on a word in the Notes editor. Similarly, the standard says that you if minimize a document in a multiple-document interface application, the document should be reduced to an icon. But when you elect to minimize a document in Notes, the document's window is simply made smaller—not iconized.

This degree of surprise doesn't cost you any data, but it is disconcerting when a Windows application doesn't work the way you've come to expect Windows applications to work. Your work is interrupted while you wonder why an action that has become instinctive didn't have the expected result.

Keep the User Informed

Menu items and mouse clicks that don't work the way they should aren't the only surprises users have come to dread. Worse, sometimes, are the kinds of surprises you get from the long, dark, and silent type of application—the ones that put an hourglass on the screen and start spinning your hard disk wildly, while you sit there wondering, "What's it doing?... Is it saving something?... Is it hung up?... Is it trashing my disk?... Should I call 911?"

Which brings us to the first corollary to the No Surprises rule: Applications should always communicate what they are doing and why.

Given Windows' graphical nature, in many cases your application's normal screen updates will suffice to supply this communication. As your application performs its tasks, the windows being displayed and the data being presented will change, providing the user with the necessary clues about what is happening.

At other times, however, your application will embark on a long task that involves no immediate screen updates, such as indexing a large data file. If the application simply puts up a busy indicator and does its stuff, the user has no way of monitoring its progress. You can prevent that by simply displaying an indicator on the screen that shows the routine's progress—a percentage meter, perhaps, or a counter showing the number of records that have been indexed.

Make the Application Wait

That brings us to the second corollary to the No Surprises rule: Applications should wait for the user's command.

If you're like me, you hate applications that make alterations to your PC's AUTOEXEC.BAT or CONFIG.SYS file without asking your permission. It's an obnoxious habit that shouldn't be countenanced in any application.

Initiative is all well and good, but you can trip up the user by showing too much of it. How much is too much? That's a tough call, but generally it is safe to say that an application should take care of its internal housekeeping chores (such as maintaining its private .INI configuration file) automatically, but should wait for the user's command or approval before taking any action that might affect the user, the user's work, or the user's environment.

What this means in practice is that an application shouldn't save changes in a document before the user tells it to, unless you provide a way to undo the changes. Nor should it embark on long procedures without letting the user know that they will be time consuming. And it shouldn't make any changes to the Windows configuration files, or create or rearrange Program Manager program groups, without asking permission first. No one likes this kind of surprise.

Rule 4: Try to Delight the User

Here's where I contradict the No Surprises rule: Good surprises—surprises that delight the user—are okay. In fact, they are to be encouraged. Generally, a good surprise is one that anticipates what the user will want to do next.

A spelling check program provides a classic example of the difference between a bad surprise and a good one. You wouldn't want a spell checker to automatically correct everything that it perceived as a misspelling in your text—that would be an unpleasant surprise, to say the least. Given the limitations of most electronic dictionaries, it would inevitably introduce as many errors as it removed, or more.

On the other hand, a spelling checker can pleasantly surprise the user by automatically suggesting alternatives for any word it identifies as a misspelling. That goes beyond its original mission of simply identifying errors, but still leaves the user the option of ignoring its suggestions.

The “Do you want to save changes...” dialog box that most Windows applications put on screen when you try to exit without saving your most recent work is another example of a program anticipating a user's desires and reacting accordingly. And, once again, it takes the middle ground between a literal interpretation of the user's command (to shut down the application without saving the changes) and a presumptuous one (to save the changes automatically without determining whether that is what the user wants).

Another area where you can pleasantly surprise the user—or at least anticipate the user's wants—is in error handling. Say, for instance, you try to start a second instance of an application that, for one reason or another, allows only one instance of the application to run. The application can simply react with an error message, such as “WordMan already running,” or it can anticipate that perhaps what the user actually wants to do is to use WordMan, and thus react to the error by activating the running copy of WordMan.

Rule 5: Finish the Job

It's just common sense that your applications should finish what they set out to do. The user expects an application to do everything necessary to get the job done, and will be disappointed, or worse, if it doesn't do so.

In order to be sure that your application meets this criterion, you have to fully analyze the task for which the application will be used. You've got to identify the task's boundaries—where it begins and ends—and all the steps necessary to take the user from one to the other.

M.M.M., the MCI mail management system discussed in Chapter 15 is a good example of this. Most E-mail scripts start and finish with the task of logging onto a remote computer and sending or receiving messages. But although those tasks are integral to the process of electronic-mail management, they aren't the whole story. The E-mail user needs a text editor to create messages, an electronic address book, ways of organizing messages that have been sent and received, and simple methods for responding to messages or forwarding them to other individuals. A complete electronic-mail management system, such as M.M.M., has to address all these needs, not merely the core functions of actually sending and receiving messages.

Chapter 6, “The Nuts and Bolts of Application Design,” describes the process of task analysis in more detail.

Rule 6: Make Applications Open-ended

It happens all the time: You set out to build a simple utility, a disk labeler maybe, and before you know it some user is telling you, “Yeah, but if you just changed this it would be a great music synthesizer.”

And that's a good thing. If the first computer programmers had been right about what people would do with computers, there would be five or six

computers in the whole world today. Fortunately, users are more inventive. They see uses for programs that the programmer never imagined, and push every program's limits in finding new uses.

In fact, the rarest occurrence in the programming world is the instance of a programmer actually knowing everything that users will want to do with an application. So, given that *you* won't know, how do you make room for future expansion?

The primary step you can take is to make your applications as open-ended as possible. Provide ways for them to interact and exchange data with other applications. Use standard rather than proprietary file formats, if possible.

Throughout the application-design process, at every step think of other ways the user might want to use your application, other things he or she might want to do with it. Then, as you test your application with real users, watch them and talk to them to see what new ideas they come up with, and what else they would like the application to do.

Rule 7: Design for Reliability

Reliability is one of the most important factors in usability, and yet it is one that seldom gets mentioned. Most people don't think of reliability as an element of usability, but it is as basic as you can get. An application that isn't reliable, that doesn't do what it is supposed to do when it is supposed to do it, isn't usable.

There are two key steps to making your applications reliable: First, test them yourself under every conceivable circumstance. Torture-test every possible combination of commands and procedures; test it on as many different kinds of hardware and with as many other applications running as possible.

Then, when you're sure you've gotten every possible bug out of the thing, give it to some users to test. They'll try to do things with it that you never imagined, issue commands in an order you never anticipated, and, undoubtedly, they'll come up with dozens of problems that you missed.

Rule 8: Don't Overwhelm New Users

If you want your application ever to have experienced users, you've got to make sure that it doesn't stump new ones. Otherwise, you'll just scare people away before they get to know your program.

The best thing you can do to help new users approach your program is to make sure that it follows standard Windows conventions—so that users can apply their knowledge of other applications to your application—and that at every stage it makes clear what it is doing.

If, on the other hand, you consistently ignore the standard Windows conventions, and don't follow the guidelines outlined here, your application will

be both difficult to learn and hard to love. What seem like little decisions in the design process may end up costing a lot in terms of usability.

Take, for example, Lotus Notes' failure to conform to the Windows convention for highlighting words the user double-clicks on or for minimizing documents. "Innovations" like these don't strike anyone as good ideas—they simply make your application harder for novices and experienced users alike to use.

As long as you make sure that new users understand what your application is doing and what it expects from them, you won't overwhelm them—and pretty soon your problem will be one of satisfying experienced users.

Rule 9: Don't Delay Experienced Users

In the world of software, to be slow is to be unfriendly. The more experienced a user is with your application, the more frustrating slow performance becomes. It's one thing to be a little slow for the user who is new to your application and still needs to read menus and dialog boxes, but it's something else entirely to make users who know what they're doing wait for your application to catch up at every step. The latter user expects action when he or she presses a command button, not an agonizing wait for an hourglass-shaped "this application is busy" cursor to go away.

Your goal should be to have every part of your application respond and complete its task instantly. That's impossible, of course—even the fastest computers and best-written software make you wait if you give them a complex enough task. But if you aim for that goal, and try relentlessly to speed up any routine that doesn't meet it, you'll end up with an application that will please even the most experienced user.

One of the best ways to speed up the operation of a Windows application is to provide shortcuts for experienced users. Build in keyboard shortcuts or hot keys for menu tasks, so that, for instance, rather than pulling down the Edit menu and selecting Cut, an experienced user can simply press Shift-Del to cut data from a document.

Beyond that, experienced users will benefit from the open-endedness and flexibility that you build into your application. Whereas new users will likely use the application only in the ways you predicted, experienced users will imagine new uses for it, and will be frustrated if it isn't capable of handling them.

Rule 10: Design for the User's Convenience—Not Your Own

Hubris alone doesn't account for the preponderance of applications that show insufficient regard for the user. Software designers may be an opinionated lot, but they're not all guilty of such arrogance. Frequently the source of the problem isn't conceit; it's laziness.

There is a right way to do every job, and there is a way to cut corners. In a world where there never seems to be enough time, it's inevitable that you'll sometimes have to settle for "good enough." It even takes some skill—you have to know your job well enough to know what corners you can cut without having the whole structure collapse on your head.

When you're programming, you can't cut corners designing an application's underlying data structures. The application is either going to be able to handle the data requirements of the job, or it isn't, and it will be obvious almost immediately if it can't. Similarly, only a fool cuts corners in an area like error-handling—since any shortcuts you take there will come back to haunt you a thousand times over.

Consequently, more often than not, the user interface is the place where corners get cut in program design. You start off by saying, "I can avoid writing 10 or 15 routines if I just insist that the user meet me halfway on this point," and then, gee, that felt so good you do it again and again.

But pretty soon you can be insisting that the user meet you halfway on so many points that he or she would have been better off walking than trying to take your bus. For example, maybe it's more convenient for you to deal with data in comma-delimited format, so you insist that the user supply it in that format, rather than in its existing format. Or it's more convenient for you to accept commands in a certain sequence, so you insist that the user perform them in that sequence, even just to access a subset of the functions provided by your program. Or your program stores two types of data in different ways internally, so you insist that the user treat them as distinct items, even if from the user's perspective they are linked.

That's how a lot of bad programs get written: Design decisions are made for the convenience of the developer, not the user. As a result, the program forces the user to focus on the process of using the program, not on the job to which the program is being applied. In short, the tool becomes more important than the task.

User interface has always been the "safest" area in which to cut corners as a programmer because it's the hardest one for users to pinpoint later. They may know that the application doesn't "feel right," but there are no obvious bugs that they can identify. Again, however, scrimping on the interface is a self-defeating strategy, because in the long run, people won't use programs that don't feel right. They'll either seek another solution, or fudge their way through the process using your application as little as possible, or simply resent the application so much that their work as a whole suffers. Only a hardened cynic would view that kind of result as "good enough."

That's it for Bonner's Rules. Keep them in mind as you read the next chapters on application development and as you build your own applications. That way, if you do end up mutating into a software designer, you'll at least be able to hold your head up.

C H A P T E R

6

The Nuts and Bolts of Application Design

*Identifying the
Application's Purpose*

*Picking a Development
Tool*

Drawing a Flowchart

*Defining Input
Requirements*

Planning Data Structures

*Defining Internal
Processing*

Putting It All Together

THIS CHAPTER EXAMINES THE NUTS-AND-BOLTS ASPECTS OF APPLICATION design—the “under the hood” stuff that actually makes your application do what it is supposed to do. We’ll look at the processes of identifying application requirements, picking a development tool, developing a flowchart for your application, planning links to other applications, and determining your application’s data requirements and data structures.

Throughout this chapter I use the decisions made during the planning of the AutoPrint for Windows print queue application (described in Chapter 11) to illustrate points about the application-design process. AutoPrint allows you to schedule lengthy print jobs for a time when your computer is otherwise unoccupied. Ultimately I built two versions of the application: one that works with Norton Desktop for Windows and one that works with the Windows 3.1 File Manager and the WinBatch batch language. Unless otherwise noted, the discussion in this chapter concerns the latter version.

Although AutoPrint is a simple application with a minimal amount of code, the decisions that went into its design and development mirror those you must make in application-design projects of any size.

Identifying the Application's Purpose

Before you begin to design an application, you have to identify what it is the application is supposed to do. You’ll often find that, as you begin work on an application, your understanding of its scope and requirements is sketchy at best. You know you want the application to take you from point A to point B, but you’re not really certain how many stops it will have to make in between or what mode of locomotion the application should employ.

For instance, the AutoPrint project grew out of my desire for a way to implement unattended printing of files created by Windows applications. I wanted to be able to schedule lengthy print jobs so that they would take place overnight or while I was otherwise away from my desk. But when I started work on the project, I didn’t know what kind of application development tool I would use, or how the print queue application would actually manage to print documents from other applications, or how the user would add files to the queue. In fact, all I really knew at that point was that I wanted to build a print queue.

So how did I end up with the finished AutoPrint for Windows application? For that matter, how does any application find its way from the first rough sketch through to being a finished product?

The process is somewhat involved and occasionally circuitous, but it starts with a simple step: determining the application’s functional requirements.

Determining Application Requirements

An application's functional requirements consist of the tasks the application is intended to perform and the desired result of those tasks. Generally, these can be derived from your basic goals for the application. Ask yourself, "What actions must the application perform?" "What results do I want from it?" and "How do I make it produce those results?"

For instance, consider the basic goal of AutoPrint for Windows: unattended printing of selected files. In order to achieve that goal, AutoPrint had to do the following:

1. Allow the user to add files to the print queue
2. Allow the user to specify the time at which the queue is to be printed
3. Activate itself at the specified time
4. Print all files in the queue

Each of these tasks was critical to AutoPrint's central purpose. A print queue that didn't provide a way for the user to add files to it would be pretty useless. Likewise, there would be little value to a print queue designed for unattended use if the user couldn't specify the time that printing should begin, or if the queue application was unable to recognize that time when it arrived. And, of course, a print queue must be able to print files. Thus, these tasks represent AutoPrint's functional requirements.

Note that the issue of how these tasks will be implemented has not yet been addressed. Nothing in AutoPrint's list of functional requirements specified, for instance, *how* the user would add files to the queue. One possibility was to require that the user copy all the files to be printed to a queue subdirectory. Another option was to have the user create a text file listing the files to be printed and make manual additions to it. And still another was to have the application present the user with a directory listing from which he or she could pick the files destined for the queue.

As it turned out, the actual AutoPrint application doesn't use any of those alternatives. Instead, it allows the user to designate files that are to be added to the queue from within the Windows 3.1 File Manager. To add a file to the queue, you simply highlight its listing and press Alt-P—at which point AutoPrint springs into action, automatically adding the file to its print queue.

As you can see, the functional requirements for AutoPrint played very little part in dictating the actual implementation of the project. They didn't, for instance, dictate the choice of development tool, or the user-interface design, or the exact methods by which any of the functional goals themselves were met.

Similarly, consider the programming project that started the whole PC industry on its way: Dan Bricklin and Bob Frankston building VisiCalc, the

first spreadsheet. What was VisiCalc's purpose? To let users build financial models on a microcomputer. What were its functional requirements? Users had to be able to enter numbers and formulas into the program and see how changes in one number or formula affected the results of other calculations. The program also had to let users save and recall their work, print it out, and perform a few other housekeeping tasks. But the critical element was for users to be able to see the result of a change to Assumption A on Result B.

So, in what way did that requirement dictate that VisiCalc employ a rows-and-columns format like a traditional accounting ledger? Absolutely none—there were all sorts of other ways Bricklin and Frankston could have designed VisiCalc's screen. For instance, they might have divided the screen into sections, one for numeric data, one for formulas, and one for results. They chose the design they did because that layout was already familiar to many of its intended users, and was intuitive enough to entice new users, not because it was in any way implied by the application's functional requirements.

Obviously you have to understand the functional requirements of your application before you can go very far with it. But it is equally clear that even once you do so, many questions of how to accomplish those requirements remain open. The answers to those questions will emerge as a result of choices you'll make in several areas—everything from how involved a project you're willing to undertake to the application's data-handling and user-interface requirements. But the first step toward clarifying those answers is the selection of a development tool for the project.

Picking a Development Tool

Once you've identified the functional requirements for your application, you can use those requirements to begin thinking about which development tool you'll use to create the application, as described in Chapter 4, "Choosing Your Tools."

The application's functional requirements are not, of course, the sole basis on which your choice of tools will rest. But weighing those requirements against the capabilities of the tools available to you is often the fastest way to identify the other critical elements that will affect your selection. This is not the case if your tool of choice is a language such as C or Turbo Pascal for Windows that provides access to the full range of Windows functions. But if you're going to be developing the application in a macro language or batch language, or even in Visual BASIC, the capabilities and limitations of the development tool will play a large role in determining the appearance and functionality of your application.

For instance, let's look again at the AutoPrint for Windows print queue application. Because almost any high-level development tool can be used to

build applications that interact with the user, keep track of time, and maintain lists, the first three of AutoPrint's functional requirements weren't very important in this selection process; they could have been fulfilled with almost any of the development tools described in Chapter 4.

The fourth requirement, printing all the files in the queue, was the critical element, at least in part because its implementation was so open to question. Given the diverse file-handling and printing capabilities of the various high-level development tools for Windows, it wasn't possible to select a tool for the AutoPrint project until a basic question was resolved concerning how the fourth functional goal was to be implemented.

The question that had to be answered was whether AutoPrint should actually print the files in the queue itself, or simply instruct the application that created each file to print it. The former alternative would have required AutoPrint to include extensive printer-control capabilities and intimate knowledge of the file structure of every file to be printed, whereas the latter alternative required only that AutoPrint be able to identify the application that created each document and instruct that application to print a document. I opted for the latter solution because it was so obviously the simpler alternative.

That decision simplified the choice of a development tool for the AutoPrint application. It meant that in order to meet the most critical of the application's functional requirements, the development tool had to excel at interacting with other applications. Given that need, and the simplicity of the application's remaining functional requirements, it seemed reasonable to build AutoPrint using a Windows batch language.

Other considerations might have changed that. For instance, although batch languages are wonderful for controlling other applications, their user-interface and data processing capabilities are rather weak. So if AutoPrint had required an elaborate user interface or extensive number crunching or data handling, the choice of a batch language would have been inappropriate. But even without having determined all the details of AutoPrint's operation, I was certain that its needs in those areas wouldn't exceed the capabilities of a language such as WinBatch or the batch language in Norton Desktop for Windows, and thus I felt comfortable making the commitment to develop AutoPrint with a batch language.

As described in Chapter 4, the factors that go into the choice of a development tool are complex, and in every case some compromises have to be made. You have to weigh power against speed of development and convenience versus cost. The best rule of thumb is to pick a tool whose strengths correspond to the most critical aspects of your application, and whose weaknesses won't be noticeable in the application at hand.

Once you have selected a development tool, you can begin the process of plotting the step-by-step operation of the application. One way to do this is with program flowcharts.

Drawing a Flowchart

A program flowchart is a diagram that depicts the logical relation between successive events in an application. Over the years, data processing professionals have developed an elaborate set of guidelines for flowcharts. They discriminate, for instance, between *outline* flowcharts, which merely identify a program's individual routines and input and output functions, and *detail* flowcharts, which define the programming techniques and provide directions for coding each routine. Moreover, organizations such as the American National Standards Institute (ANSI) and the European Computer Manufacturers' Association have defined elaborate sets of standard symbols to be used in these flowcharts.

For applications on the level of those presented in this book, you'll seldom need to be quite that serious about flowcharts. The conventions of a formal flowchart are designed to aid in the documentation of complex applications and to help large teams of programmers work together in a coherent manner. For a small project on which you'll be the only programmer, you should feel free to drop some of the formalities.

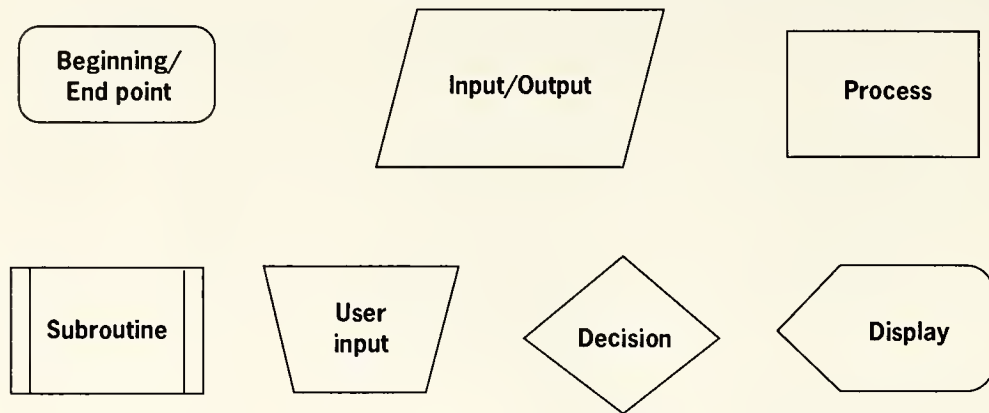
Nevertheless, the fundamental concepts of flowcharting are valuable on any programming project, no matter how modest its scope. As you build a flowchart, it serves both as a kind of scratchpad for your ideas about the application and as a tickler that helps you to identify those areas in which more work needs to be done.

As a result, you might find it worth your while to learn some of the basic flowchart symbols and conventions. You can, of course, get by with just drawing boxes or circles to represent various parts of your application, but the ability to use at least some of the standard flowchart symbols, such as the seven shown in Figure 6.1, is useful because they enable you to distinguish at a glance between the major components of an application:

- Beginning and end points
- User input
- Decisions
- Screen display
- Disk input/output
- Computational processes
- Subroutines

Figure 6.1

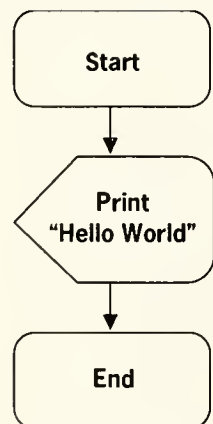
A selection of useful flowchart symbols



As you draw the flowchart, you should label each symbol and link it to the next symbol with an arrow, as shown in Figure 6.2, in order to show that the event or process represented by the second symbol follows that represented by the first symbol.

Figure 6.2

Flowchart for a simple "Hello World" application



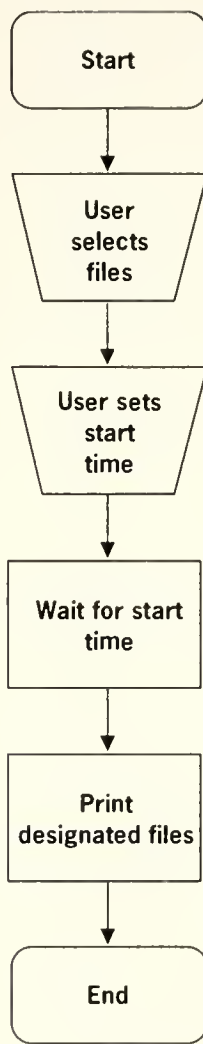
Flowchart Iterations

I generally create several flowcharts during the course of an application-development project, making each successive chart more detailed.

The first iteration of one of these flowcharts usually does nothing more than represent the functional requirements for the application in some sort of logical order. For instance, returning again to AutoPrint for Windows, the first iteration of its flowchart consisted of just four steps between the start and end boxes—user selects files, user sets start time, wait for start time, and print designated files—as shown in Figure 6.3.

Figure 6.3

A first-iteration flowchart for AutoPrint



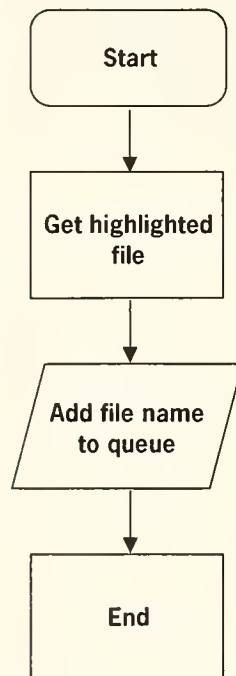
Looking at this first iteration of the flowchart, the next steps in the program-development process become obvious. You have to ask, and answer, questions such as, “How does the user select files?” “How does the program know what time it is?” and “What method does the program use to print files?”

The answers to those kinds of questions will be influenced by a number of factors, including the development tool you’ve chosen, the style of user interface you wish to employ, and the nature of the other applications with which your application will interact. So they may not all be answerable at this point. But by producing successive iterations of the flowchart, plugging in the answers as you arrive at them, and highlighting the holes where questions remain to be answered, you’ll find that the flowchart becomes a valuable tool in the application-development process.

In the case of AutoPrint, for instance, I soon made the decision that I actually needed not one program, but two: One would spring into action when the user pressed the hotkey to activate it, obtain the name of the file

the user had highlighted in File Manager or a Norton Desktop file pane, and add that file to the queue. The other program would control the actual printing of the files in the queue when the designated time arrived. The first program would store the name of each file added to the queue in a text file, which the second program would later read. So I modified my original flowchart to reflect this change in plans. The resulting flowchart of the first program (called GETFILE.WBT) is shown in Figure 6.4.

Figure 6.4
The initial flowchart
for GETFILE.WBT

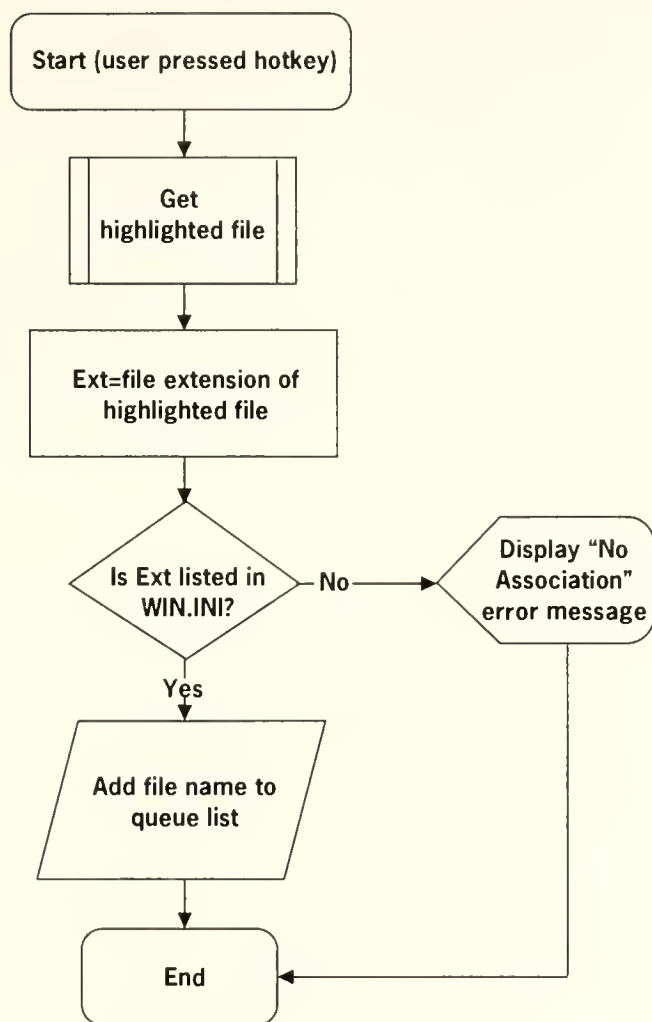


Eventually, as the program-development effort progressed, I turned the process labeled “Get highlighted file” into a subroutine, and added an error-checking routine that compared the extension of the selected file with those listed in WIN.INI’s Associations section, to ensure that AutoPrint would be able to print all the files added to the queue. Thus, the flowchart for GETFILE.WBT continued to evolve, until it looked like that shown in Figure 6.5.

Neater Flowcharts

AutoPrint is a very simple application, so it’s not surprising that the flowcharts for it are also pretty simple and easy to follow. As your applications grow more complex, however, it becomes more and more difficult to maintain neat flowcharts on paper. As you add more and more subroutines and decision points, symbols and lines start dashing off every which way until your hand-drawn flowchart starts to look like a ball of string after the cat’s done with playing with it.

Figure 6.5
A later iteration of
the GETFILE.WBT
flowchart



One way to combat the entropic effect a complex application can have on your dreams of a neatly organized flowchart is to use software, rather than pen and paper, to draw your flowcharts. Doing so won't actually reduce the complexity of the application, of course, but it might make the flowchart easier to understand and easier to maintain.

You can create flowcharts using any drawing-type program, such as Micrografx Draw for Windows, or, in a pinch, even a paint-style program such as Windows Paintbrush. But if you want a tool that's tailored for this purpose, try ABC Flowcharter, from Roykore Inc., 2215 Filbert Street, San Francisco, CA 94123; (415) 563-9175.

ABC Flowcharter is a Windows application that combines drawing-program-like tools with a set of predefined flowchart symbols and the ability to automatically route lines between symbols. And, because it allows you to create links between flowcharts, you can produce detailed charts of complex projects simply and elegantly. Symbols linked to another flowchart are

depicted with shadows in ABC Flowcharter. Double-clicking on a linked symbol opens up the flowchart to which it is linked.

For small projects, you don't necessarily need flowcharts as sophisticated as those produced by ABC Flowcharter, but it can be an invaluable aid on larger, more complex projects—serving both as a drawing board for sketching out new parts of the application and a road map to the parts you've already completed.

Once you've got a rough outline of your program in the form of an initial flowchart, it is time to start filling in the gaps by determining more of the details of how the program will actually accomplish the tasks you've defined for it. In order to do so, you have to define the application's input requirements, its data structures, and its internal processes. Let's look at each step in succession.

Defining Input Requirements

One of the key stages in any application-development project is determining your application's input requirements. Here you figure out what data the application will need in order to do its job, and from where it will obtain that data.

Almost every application requires some kind of data input, either from the user or from other sources. The exceptions are applications that are completely self-contained. For instance, a self-extracting archive contains all the data that it needs to do its job: the names and contents of the compressed files and the method by which they were compressed and should be decompressed. But in the vast majority of cases, an application requires some kind of data—keystroke input from the user or a command line parameter or some information about the status of Windows or another application.

The nature and complexity of these data input requirements vary considerably from application to application. In some cases they are very simple; for instance, consider the Windows Calculator accessory. In order to perform its basic functions, all it requires in the way of data input are the numbers it is to use and the mathematical functions it should perform on them. Both are supplied by the user.

Most applications end up having more complex data input requirements than this, however, and in most cases they rely on a combination of direct user input and other sources of data. For instance, AutoPrint makes use of direct user input to initially select files for addition to the queue and to determine the time at which the queue should be printed. But it also uses an internal file listing of a special queue directory to keep track of the files in the queue. When the time comes for AutoPrint to print the files in the queue, it opens that data file and prints the files it finds listed there, rather than relying on the user to enter the names directly at print time

(which would be most inconvenient in an application designed to provide unattended printing services).

Applications that interact with other Windows applications may also require information about the status of another application or of Windows itself in order to make that interaction possible. For instance, while it is printing the files in the print queue, AutoPrint for Windows keeps tracks of both the total number of screen windows that are open in the Windows environment and the title of the active window, in order to monitor the progress of the current print operation. If your application will require data of this type, you have to make sure that the development tool you build it with provides access to that data, either through direct program commands (as in the case of most Windows batch languages) or through access to the Windows API functions that provide that information (as in the case of Turbo Pascal for Windows or Visual BASIC, among others).

Links to Other Applications

The other applications operating within the Windows environment can also be important sources of data for a Windows application. Obtaining data from other applications eliminates the need for the user to recreate the data within your application.

There are several ways to obtain data from other applications. One way is to import data from the other application's disk-based files. This is often the speediest way to move another application's data into your application, especially when a large quantity of data is involved, but it requires that your application be able to read the files created by the other application. Thus, the ability of your development tool to provide access to files created by the other application is a key factor in determining whether your application can use this method.

Dynamic Data Exchange

Another way to integrate data from another application within your own is through a DDE (Dynamic Data Exchange) link. This alternative offers you the ability to create *hotlinks* between data in your application and data in another application, so that your application automatically responds to changes in data being collected or generated by the other application. The Windows Broker application described in Chapter 14 uses DDE to link an Excel portfolio-analysis spreadsheet to stock price data that a DynaComm script program obtains from an on-line electronic source.

There are a several limitations you should consider in planning DDE links. The first is that not all Windows applications support DDE. If the application you're developing is to obtain data via DDE, both your development tool and the potential source application must have the ability to create and respond to DDE links. The other limitation is that while DDE links are

fine for moving relatively small amounts of data, they become slow and inefficient when large amounts of data are involved. Furthermore, the source application must be in memory or your application must launch it, and the source file (such as an Excel worksheet) for the data must be available in order for your application to update a DDE hotlink to ensure that it represents the most up-to-date data.

Object Linking and Embedding

With Windows 3.1, Microsoft is attempting to popularize an alternative to DDE known as Object Linking and Embedding (OLE). OLE differs from DDE in that, rather than creating a hotlink to a data item in another application, it embeds an encapsulated copy of the file containing the remote data within your application. The encapsulated file identifies the program that created it, and tells your application how to display the data. So what you are actually embedding in your application is a picture of the data from another application—rendered on screen or printed as it would be by the other application. When you double-click on the embedded object, the application that created it is launched (if it is available on your system), enabling you to edit the embedded data in its native application.

There are advantages and disadvantages to OLE. On the one hand, it eliminates the need for you as an application developer to build kitchen-sink capabilities into your application. An application that supports OLE doesn't need, for instance, to know specifically how to display an image in order to incorporate scanned photographic images—you simply use OLE to embed a scanned image created by another application into your application.

On the other hand, OLE can be very inefficient in terms of disk space. For instance, if you're embedding data from an Excel worksheet, the size of your application's data file will grow by the entire size of the source worksheet file, even if you're actually only interested in a single cell of that worksheet.

The Windows Clipboard

The final method of linking data from other applications to your application is the simplest and most universal: the Windows Clipboard. When all else fails, you can almost always use the Clipboard to copy data from another application and paste it into your own. Some development tools will allow you to do this under program control, whereas with others you'll be forced to ask the user to intervene by selecting and copying the source data, but the option is almost always there. Of course, data imported in this way is entirely static—it won't register any subsequent changes in the source data—but copy and paste is still a great “when all else fails” option for the Windows application developer.

AutoPrint's Data Requirements

In most cases, the data requirements for an application, combined with your understanding of the capabilities and limitations of the development tool you're using, go a long way toward defining the application's structure and form. When you know what data the application will need, and from where it can expect to obtain that data, you can start thinking about how the application will interact with the user, how it will interact with other applications in the Windows environment, and what data structures it will need to maintain.

In the case of AutoPrint, I knew that the user would have to supply the time at which printing was to occur and the names of the files to print. Beyond that, however, I wanted to minimize the amount of information that the user had to supply to AutoPrint. After all, the purpose of utility applications is to provide a more convenient way to do things, and AutoPrint wouldn't be very convenient if it insisted that the user detail every aspect of how it should go about doing its job.

Still, AutoPrint needed some way to know that when the user said to print REPORT.DOC, Microsoft Word for Windows should do the printing, whereas Ami Pro should be used for printing REPORT.SAM, Aldus Page-Maker for REPORT.PM4, and Microsoft Excel for REPORT.XLS.

Fortunately, that information already exists in the Extensions section of the WIN.INI file for files created by any installed-Windows application. For instance, when you install Ami Pro it adds a line to the Extensions section that associates the file extensions .SAM and .SMM with AMIPRO.EXE. Which means that when you double-click on a file with either of those extensions in Windows File Manager, Windows will use Ami Pro to open the file.

Thus, I knew that by reading the WIN.INI file, AutoPrint could determine for itself which program should be used to print each file in the queue, and therefore that there was no need to make the user enter that data.

I still had to decide, though, how the user would add files to the queue and specify the starting time for the print job.

Obtaining Files for the Queue

I wanted to make the process of adding files to the queue as convenient as possible. So right away I eliminated the possibility of having the user type the file names into a text file in Notepad. And, a short time later, I also eliminated the possibility of having the user identify files using a file and directory listing in a dialog box created by WinBatch. WinBatch dialog boxes aren't persistent—as soon as you make a list-box selection the dialog box closes, so the user would have had to reload the program each time a file was to be added to the queue.

Instead, I hit upon the idea of using WinBatch's Macro facility (which allows you to create application-specific hotkeys for WinBatch applications) to link the file-selection routine to the Windows 3.1 File Manager. As soon as

the user pressed the hotkey, the batch program would run, adding the name of whatever file was highlighted in File Manager to the print queue.

It was a simple solution, and one that added value to a tool that the user would already be using (File Manager) rather than requiring that he or she master a new tool. Unfortunately, there was a catch. WinBatch doesn't offer any way to read the screen, so that even though the cursor was on the file name, there was no immediate way for the application to read it.

In the Norton Desktop version of AutoPrint, which I completed first, I solved this difficulty by having WinBatch issue the File Print command to open up Norton Desktop's Print dialog box. Then I had it issue the Ctrl-Ins (Copy) command to copy the contents of the edit field in which Norton Desktop lists the full name and path of the highlighted file, before sending an Esc command to back out of the dialog box. I thus succeeded in copying the full file name to the Clipboard, from where it was easy to add it to a list of other files that had previously been added to the queue.

Unfortunately, that method didn't work with File Manager because none of its dialog boxes ever display the full path of a file in an edit box. Instead, they simply display its name. And that wasn't good enough because the queue application had to know both what files to print and where to find them. So instead, I decided to have the file-selection batch program issue the standard File Manager Copy command to copy the highlighted file to a special queue directory called AUTOPRN. Then, once each file is printed, AutoPrint deletes it from the AUTOPRN directory.

Obtaining a Start Time

The final chunk of data that AutoPrint needed was the time at which files were to be printed. Again, I wanted to make things as convenient for the user as possible. So I thought the user should be able both to designate a standard time at which files in the queue should be printed every day and to override that when the need arose to print them at a different time.

The solution here was to allow AutoPrint to accept a command line parameter (something that WinBatch programs, like DOS batch language programs, do very well) specifying the starting time. This would allow the user to use the File Run commands in either Program Manager or File Manager to launch AutoPrint, and then specify a start time on the command line. Alternatively, if the user wanted to print the queue at the same time every day, a Program Manager icon with a standard starting time designated on its command line could be placed in the Startup group.

I realized, however, that there would be times when the user would neglect to specify a start time on the command line, so the first thing AutoPrint does when it launches is display a dialog box containing an edit field in which the user can enter a start time for the print job. The edit field initially displays either the time (if any) that was passed on the command line or "NOW", but the user can replace that with any time he or she wants. Then,

when the user clicks the OK button to close the dialog box, AutoPrint reacts by either immediately printing the queue (if the print-time field contained “NOW”) or by shrinking to an icon and waiting until the designated print time arrives.

That completed the process of defining AutoPrint’s data input needs. It now knew which files to print, when to print them, and how to print them.

Planning Data Structures

The structures that your application will use to maintain and store data are another important consideration early on in the program-development process.

These structures generally incorporate the standard data types that were discussed in Chapter 2—strings, integers, long numbers, and so on—although some development tools may support additional types, such as a date type that can be used to store and manipulate date-oriented information.

In addition, some development tools may utilize special forms of the standard data types. For instance, if you’re using Excel’s macro language to develop an application, you’ll think in terms of cell values and labels rather than numbers and strings, but you’ll find that the Excel data types act in very much the same way as their equivalent standard data types. You can even create arrays on the worksheet that serve the same purpose as a string or numeric array in a more traditional programming tool.

Defining Variables

When you plan the data structures for your application, you’re determining how your application will use the data types available to it to store the information that it needs. Some of the information it stores may be obtained from other sources, such as user input or links to other applications, whereas other information will be created within your application. For instance, an application that multiplies any two integers entered by the user and displays the result might use three integer variables: two, X% and Y%, for instance, to store the two integers entered by the user, and the third, Z%, to store the product of the first two numbers.

On the other hand, if you knew that you wouldn’t have any need for the original integers once their product had been obtained, you could conserve memory by redefining one of the variables used to hold the result of the multiplication operation, as follows:

$$X\% = X\% * Y\%$$

Prior to this statement, the variable X% would hold the first integer input by the user. But after this statement has been executed, it would hold

the product of the two integers input by the user, and the program would no longer “know” the original value of X% (although as long as Y% isn’t redefined, the program could redetermine the original value of X% by dividing its new value by Y%).

This simple example demonstrates that, in addition to needing to know what kinds of data your application will need to manipulate (and thus what data types to store it in), you need to think about the life span of each data item as you define it. Some variables are defined in one statement, used in the next, and then never used again, whereas others are used repeatedly throughout the execution of an application. For instance, let’s turn the example above into a guessing game in which the application selects a random integer from 1 to 50, and the user has to guess which one the computer has selected, as shown in Figure 6.6. You could create the functional guts of that application in Visual BASIC using the following code:

```
Sub Guess_Number
Randomize
Correct% = Int(Rnd(1) * 50) + 1
Do
    Guess% = Int(Val(InputBox$("Guess a number from 1 to 50")))
    Select Case Guess%
        Case Is = 0
            Exit Sub
        Case Is = Correct%
            Exit Do
        Case Is > Correct%
            Print Str$(Guess%) + " is too high"
        Case Is < Correct%
            Print Str$(Guess%) + " is too low"
    End Select
Loop
Print "You guessed it!"
End Sub
```

Correct% is the random integer selected by the application, and Guess% is the user’s current guess. Correct% is established early in this routine by the statement:

```
Correct% = Int(Rnd(1) * 50) + 1
```

which tells Visual BASIC to multiply 50 times a random number equal to or greater than 0 and less than 1, add 1 to the total, and set the variable Correct% equal to the integer value of that number (that is, any decimal places are dropped). The RANDOMIZE statement preceding this line tells Visual BASIC to “reseed” its random number generator—otherwise it would select the same “random” number every time, significantly reducing the complexity of the game.

Figure 6.6
A simple guessing
game application



Once the value of `Correct%` has been established, it does not change. `Guess%`, on the other hand, is redefined every time the user enters another guess. Visual BASIC creates an input box labeled “Guess a number from 1 to 50”, and then assigns the integer portion of the value of the string entered by the user to the variable `Guess%`. So, for example, if the user entered **49.5**, `Guess%` would be equal to 49. On the other hand, if the user entered **Dog**, `Guess%` would be equal to 0, since the value of a nonnumeric string is 0.

But what if you wanted to retain the value of each of the guesses made by the user? Suppose, for instance, that you wanted to limit the user to five guesses and display the value of each wrong guess if he or she doesn't hit upon the correct number within that limit. To do so, you could modify the above code as follows:

```
Sub Guess It
Randomize
Correct% = Int(Rnd(1) * 50) + 1
ReDim Guess%(5)
For X = 0 To 4
    Guess%(X) = Val(InputBox$("Guess a number from 1 to 50"))
    Select Case Guess%(X)
        Case Is = 0
            Exit Sub
    End Select
Next X
End Sub
```



```

    Case Is = Correct%
        Exit For
    Case Is > Correct%
        Print Str$(Guess%(X)) + " is too high"
    Case Is < Correct%
        Print Str$(Guess%(X)) + " is too low"
End Select
Next X
If X < 5 Then
    Print "You Gussed It"
Else Print "All your guesses were wrong!"
    For X = 0 To 4
        Print "Guess " + Str$(X+1) + " was " + Str$(Guess%(X))
    Next X
Print "The correct answer was: " + Str$(Correct%)
End If
End Sub

```

Here `Guess%` has been changed from a simple integer variable into an integer array with five elements (element 0 through element 4), and the `DO` loop, which would repeat endlessly until the user guessed the correct number, has also been changed, to a `FOR-NEXT` loop that gives the user only five guesses.

As each guess is made, its value is assigned to element `X` in the array `Guess%()`, so that the first guess is assigned to `Guess%(0)`, the second to `Guess%(1)`, and so on. If the user guesses the correct number, the `EXIT FOR` command makes the program exit the loop prematurely, so that the value of `X` will always be between 0 and 4 if the correct guess was made.

Otherwise, the loop continues, incrementing `X` by 1 each cycle until its value is greater than the upper boundary on the `FOR-NEXT` loop. So if the user fails to guess the correct answer, `X` will be equal to 5 at the conclusion of the `FOR-NEXT` loop. In that case, a second `FOR-NEXT` loop is performed in order to print the value of each incorrect guess, as recorded in the array `GUESS%()` and shown in Figure 6.7.

Global Variables

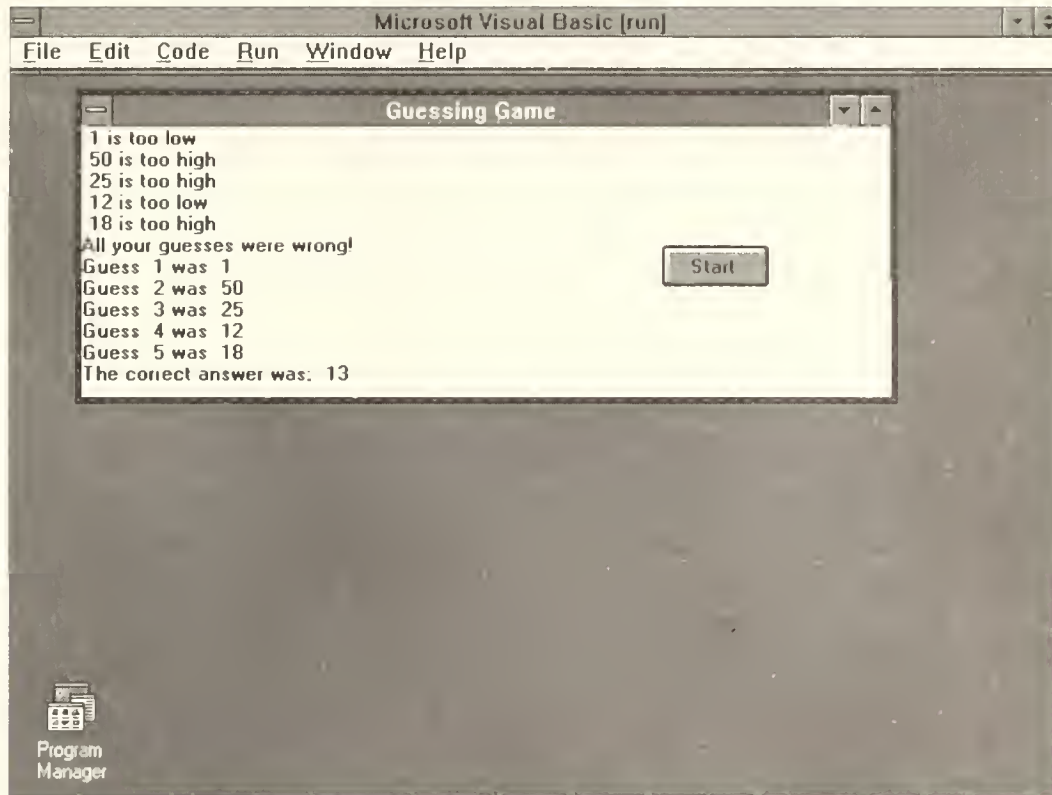
Because of the simplicity of the preceding application, there was no need to use the values of `Correct%` and `Guess%` anywhere outside the routine where they were defined. However, in a more complex application you might want to make use of those variables in additional routines. If so, then you'll need to follow your development tool's scoping rules to have it retain the definitions and values of these variables even when the current routine is done and program control jumps to another routine. In Visual BASIC, for instance, you could define `Correct%` as a global variable by including the line

```
GLOBAL Correct%
```

in your application's Global module. Doing so would make the current value of Correct% available to every routine in the application, and allow any routine to modify its value.

Figure 6.7

The revised
guessing game



Disk-Based Data Formats

If your application is going to use disk-based storage for data, then you have to think about disk-based data structures, as well as internal ones, as you write the application. Generally, you'll use disk storage if you want to preserve data from one time that the application is run until another, or if you want to work with large amounts of data that won't fit comfortably into RAM.

Planning disk-based data structures is simplest if you're using an application macro language for your development work because in most cases you'll use the underlying application's native file format for the disk-storage needs of your custom application. For instance, if you're writing an application in Excel's macro language, you'd use the Excel XLS file format to store any data used by your custom application. Similarly, you'd use the dBASE file format for an application developed with dbFast/Win, and the Paradox format for applications developed in ObjectVision.

Of course, not all applications that feature macro languages have such convenient file formats. You might not find the standard document formats

supported by Word for Windows or Ami Pro to be quite so useful for tabular data or data from small database applications written in their macro languages. And, of course, languages such as Turbo Pascal, Visual BASIC, and WinBatch don't have predefined file formats. So it may turn out that you'll need to devise your own data-file structures for your application.

There are endless ways to organize disk-based data. Almost every application has its own format(s) for storing data to disk, and seldom are any two directly interchangeable. In addition, the capabilities of the various development tools discussed in this book for using disk-based data, and the methods by which they do so, all vary greatly. But most Windows development tools provide access to at least three basic ways of storing data: sequential text files, random-access text files, and INI files.

Sequential Text Files

Sequential text files are useful for storing random-length data that will be read in sequence. Generally each field is anywhere from 0 to 255 characters long (although longer data items are possible). Each field is usually separated from the next field, and each record from the next record, by standard delimiters. For instance, a comma might be used to separate fields, and a carriage return-line feed combination to separate records.

Sequential text files are very easy to use and flexible—you don't need to know how much data is in any particular record, or the form that the data will take, or even how many records the file contains. But they're also inefficient in many ways. For instance, in order to read the contents of the 4,000th record in the file, you must first read all the records preceding it because without knowing how long each record is there is no way to jump directly to the beginning of the record 4,000. (You can get around this problem by indexing the records, so that you know the starting position and length of each record in the file, but doing so adds considerably to the complexity of the file-access process.)

In addition, it is difficult to make modifications to records in a sequential access text file, again because the length of each field and record can vary. Shortening or lengthening even a single data field in one record might mean that you have to rewrite the entire database because otherwise there would be no place to store the extra data (in the case of lengthening a field), and no way to indicate that the space a shortened field no longer uses is not part of the record. Attempts to delete a record, or to overwrite an existing record with a new one, pose similar problems.

For these reasons, it is best to use sequential text files in situations where all records will be read from disk or written to disk at once, such as an application's configuration file or a mini-text editor that stores each line of text as a separate record. Sequential files provide a straightforward, low-maintenance solution to those kinds of disk storage needs—where there is no need to know where one record ends and another begins because you're reading or writing all at once.

Random-Access Text Files

Random-access files are so named because you can read or write any record in a random-access file individually, without having to first go through all the records in the file that precede it. This is possible because, rather than using field and record delimiters, each record in the file has a fixed length, as does each field in each record. So if each record is 250 bytes long, you know that the first field of the second record will begin with the 251st byte of the file.

Random-access files are much more responsive than sequential text files in many ways. For instance, you can edit, overwrite, or delete records without having to rewrite the entire file. But they are in other ways somewhat more restrictive. You have to know in advance how much space will be required for each field and each record, and changing the length of a record or field can be difficult. For instance, if you discover that you actually need 55 characters to store each record, rather than 50, you'll have to convert all the records for which you've entered data so far to reflect the new record length. Otherwise, if you simply change your application to read 55-character records from the file, the first record it reads will contain all of the original first record and the first five characters of the original second record; the second record it reads will start with the sixth character of the original second record; and so on.

Thus, you have to plan carefully when defining random-access files. You'll be rewarded for that effort, though, by being able to read or write any record, or any field in any record, at any time, thus speeding up your application's data-storage and -access routines.

INI Files

INI files are specially formatted text files primarily intended to be used by Windows and Windows applications to store configuration data. Every Windows 3.1 system has a WIN.INI file in which some configuration data is stored, but applications may also create and maintain private INI files. For instance, Ami Pro uses a configuration file called AMIPRO.INI, Turbo Pascal for Windows uses one called TPW.INI, and so on.

INI files are text files that follow a standard format. Each INI file is divided into sections, the name of which appear inside brackets, as follows:

```
[section name]
```

Each section can contain one or more fields, which look like this:

```
Datadir=C:\SHEETS  
Macrodir=C:\MACROS
```

The text to the left of the equals sign in each field is known as the *key name*, and the text to the right of the equals is called the *key*.

INI files function as sort of a cross between random-access and sequential files, in that you can request the value of a single key in a specified section without having to instruct your application to read through the entire INI file (although that is what happens in the background). In addition, Windows provides some special support for the WIN.INI file, specifically a standard system message, WM_WININICHANGE, which can be used to notify applications that have read data from the WIN.INI file that it has been changed and they should consequently reread the file.

Some Windows development tools, such as WinBatch, have built-in commands for creating, reading, and writing INI files. Others, such as Visual BASIC and Turbo Pascal for Windows, allow you to access the standard Windows API routines for doing so. Thus, INI files provide a good way to store application configuration data. However, although it may be tempting to do so, you should resist the urge to use the INI file functions to store larger amounts of data. Windows INI file functions were not designed to work with huge amounts of data, and prove to be slow and inefficient when you try to use them to write hundreds of lines of data at once.

Other Data Formats

Beyond the three formats described above, there are many other ways to store data to disk, including a seemingly endless array of binary file formats. In binary files, data is transferred to disk in raw, unformatted form, rather than as ASCII or ANSI text. This enables binary files to be used to store a much wider variety of data types—everything from graphic bitmaps to executable program code—that cannot be rendered in ASCII form. However, it also makes using binary files considerably more complex than using text-based files. And, unless your application development tool provides implicit support for one or more binary file formats, you would be well advised to master other areas of the program-development process first, before tackling this one.

AutoPrint's Data Structures

AutoPrint's data structures are very simple. Partly this is due to the uncomplicated tasks that it performs, and partly to the limitations of the WinBatch language in which it is written. WinBatch doesn't support arrays and doesn't have any scoping rules—all variables are global to the application.

In any case, most of the variables in the two batch programs that make up AutoPrint are merely used to hold the results of functions or calculations. For instance, the line

```
NewApps=WINITEMIZE()
```

assigns the results of the WINITEMIZE function (which returns a tab-delimited list of all open windows) to the variable NewApps. The next line

in the application counts the windows listed in `NewApps`, by using the `ITEMCOUNT` function to count the number of times the contents of the variable `HTab` (which had earlier been defined as a horizontal tab character) appear in the contents of `NewApps`, and assigns the result to the variable `NewAppCount`:

```
NewAppCount=ITEMCOUNT(NewApps,HTab)
```

That is about as complex as `AutoPrint`'s data structures get. Because it copies the files to be printed to the `AUTOPRN` directory, the `WinBatch` version of `AutoPrint` doesn't even need to maintain a list of those files. Instead, it simply obtains a listing of the files in that directory when it is ready to begin the printing process.

The Norton Desktop version of `AutoPrint`, on the other hand, uses a disk-based data file to store the full path and file name of each file to be printed, in lieu of copying those files to a queue directory. It uses a simple sequential text file called `AUTOPRN.LST` for that purpose, simply appending the full name and path of each new file to be printed to the end of the list, followed by a carriage return. When the time comes to print the files in the queue, `AutoPrint` reads through `AUTOPRN.LST` line by line, printing each file listed there in sequence.

In addition, both versions of `AutoPrint` make heavy use of the `WIN.INI` file: to ensure that the file extension of the file being added to the queue is listed in the `Extensions` section of `WIN.INI`, to determine the default printer, and to determine whether or not the Windows Print Manager print spooler should be used. Fortunately, `WinBatch` makes access to either the `WIN.INI` file or private `INI` files quite simple, through commands such as `INIREAD`, which is used as follows:

```
Spooler=INIREAD("Windows","Spooler","yes")
```

This command assigns the key value of the key name `Spooler` in the `Windows` section of `WIN.INI` to the variable `Spooler`. The two possible settings for this key are `yes` and `no`. The third parameter in the `INIREAD` call instructs the `INIREAD` function to return a value of `yes` if the key is not found.

Defining Internal Processing

All the data structures and disk files and data sources in the world aren't much use if your application doesn't know how to do anything with them. In order to put those creations to work, you have to define the internal processing that your application will use to achieve its tasks.

Doing so is basically a problem of translation. You need to translate your mental description of each process the application must perform into one that your development tool will understand.

For instance, I knew that AutoPrint had to be able to find the application that created each file in the queue in order to print it. So it seemed reasonable to test whether or not WIN.INI contained an association for each file specified by the user at the time the user specified it. Figuring that out, however, was only half the job, because the description would mean nothing to WinBatch. I still had to translate that description into WinBatch code, which I did, resulting in these lines:

```
Ext=FILEEXTENSION(FileToPrint)
AssocExists=INIREAD("Extensions",Ext,@FALSE)
IF AssocExists==@FALSE THEN GOTO AssocErrJump
```

The Extensions section of WIN.INI uses the three-character file extension to associate document files with the applications that created them. So the first line of code here uses WinBatch's FILEEXTENSION function to obtain the extension of the file that the user has designated (the name of which is stored in the variable FileToPrint) and assigns the extension to the variable Ext.

The second line of code uses the INIREAD function to determine if the extension is listed in WIN.INI. If it is found there, the function will return the name of the application associated with that extension. If the extension is not found, it will return the default specified by the third parameter to the INIREAD function: @FALSE. In either case, the returned value is assigned to the variable AssocExists.

Finally, the third line tests the value of AssocExists and jumps to an error routine if it is equal to @FALSE. (@FALSE is a predefined constant in WinBatch.)

In order to be a successful translator, you need to know both languages inside and out. Fortunately, however, you needn't be terribly fluent in a programming language before you can start programming. Like a tourist traveling abroad with a traveler's phrase book in hand, you can look things up as you go along, skimming the language's reference manual for inspiration and relying on its index, the examples it contains, and its on-line help system for aid when you get stuck. Studying other peoples' work in the language can also be very instructive. You'll learn what control structures work best with the language and what things it seems to do easily, and you might pick up some good tips and shortcuts.

It's not always fun to struggle with a phrase book or a programming language reference guide, but you do have one advantage here compared to the tourist who's hoping he can find the words for "I'd like a half a kilo of cheese, a loaf of bread, and a bottle of red wine" before the grocer loses patience and throws him out of the store. Your PC doesn't care how long it takes you to come up with the right syntax, as long as you get it right in the end. And, although in the early stages you might occasionally spend a day or more struggling to figure out how to do something that in the end proves so

simple you could kick yourself, eventually the rules and syntax of any application development tool that you use frequently become as natural to you as those of your native tongue.

Putting It All Together

As the various projects presented in Chapters 9 through 15 will demonstrate, program development is an iterative process. It might be simpler if the route was a straighter line—if you could start by defining your application's objectives, then pick a programming tool, then plan the application's data requirements and structures, then throw together a user interface, and finally crank out the code to glue it all together—but that's not the way it works. The various parts of a programming project are all interconnected; you can't lose sight of any of them no matter what stage of the project you're at. Your user-interface design objectives will affect your choice of development tools, but your development tool will also mold your user-interface objectives. In the end, each of the pieces must work, and must blend harmoniously with all the others, in order for the application as a whole to succeed.

The next two chapters continue this discussion of the process of developing a Windows application. Chapter 7 discusses the standard elements of the Windows user interface and how they should be used to build effective applications, and Chapter 8 discusses prototyping and testing your applications.

C H A P T E R

7

Implementing a Windows Interface

*Application Windows
Features*

*Standard User-Interface
Controls*

The SAA Standard

*Common Extensions
to the Standard*

The Keyboard Interface

Dialog Box Design

Putting It All Together

NOW WE GET TO THE FUN PART—DESIGNING AND IMPLEMENTING A USER interface for your application. Every Windows application development project involves at least a little bit of user-interface design work. Even a simple little utility such as a WordBASIC keyboard macro requires that you answer at least some user-interface questions.

Say for instance, that you want to be able to highlight an entire paragraph in a Word for Windows document by issuing a single keyboard command. You could do that with a macro consisting of the command `EXTENDSELECTION` repeated four times, as follows:

```
Sub MAIN
EXTENDSELECTION
EXTENDSELECTION
EXTENDSELECTION
EXTENDSELECTION
End Sub
```

Once you've written or recorded this macro, to put it to use all you have to do is assign it a keystroke combination (using Word for Windows' Keyboard Options dialog box), so that the macro will be executed whenever you press those keys.

The only question is which keystroke combination you'll assign to it, and that's a user-interface design question.

Ideally, you would assign the macro to a keystroke combination that is easy to remember. Word for Windows allows you to assign macros to letter keys in combination with the Shift and/or Ctrl key. In this case, you might want to use a combination that includes the letter "p", since *paragraph* begins with that letter. Shift-P would be a bad idea, of course, since you need that combination to type a capital "P". Ctrl-P would be ideal, but Word for Windows already uses that combination to provide quick access to the drop-down list box that controls font size. You can change that, but since the Shift-Ctrl-P combination would work just as well for the paragraph selection macro, there is no need to change the default setting for Ctrl-P.

The choice of keystroke combination to use for accessing a single macro is a minor user-interface design problem, of course, but it is not a trivial one. If you pick a bad combination—one that is difficult to remember or awkward to reach—you may end up deciding that it is easier to use another method to select the paragraph.

Of course, the problem wouldn't arise at all if there were a standard keystroke combination used for highlighting paragraphs. Alas, there is no such standard for this command in the Windows environment. However, the Windows environment does offer standard solutions for a great many other user-interface design issues. The better you understand these standards, the easier the process of designing a user-interface design will be for you. Moreover, by

adhering to these standards, you'll make it easier for anyone who already knows how to use any other Windows application to learn how to use yours.

The Standard Advantages

They say that eventually you can get used to just about anything. That has certainly been true in the case of PC user interfaces. Many of the most popular PC applications over the years, including WordStar, WordPerfect, and just about every other major DOS word processor, have had truly awful user interfaces (in my humble opinion). They take years to master and only hours to forget. Most have so little internal consistency that your knowledge of one feature is of no benefit to you in learning to use another. Plus, if you don't use a feature for six months, you might as well never have used it because its procedures will probably have nothing in common with those of the features you have used more recently.

Of course, when you make those criticisms in front of a WordStar or WordPerfect user, you're as likely to get a punch in the nose as agreement. After all, not only did that person shell out good money for the program you're criticizing, but he or she also went through all the trouble of learning to use it. Some people don't take kindly to having all that effort questioned. Moreover, they'll argue, there is nothing intrinsically simpler or more intuitive about the Windows interface than about the interface used by their DOS application.

They're right, in a way. Although buttons and scroll bars and icons make a lot of sense once you've learned to use them, there is really no way that you could call the Windows interface intuitive, any more than you could call the user interface of an electric can opener or an automobile intuitive. You have to learn a specific set of procedures before you can use any of them. One might argue that a certain Windows application is marginally easier to use than a certain DOS application, but all computer applications, no matter what user interface they employ, require some study before you can put them to use.

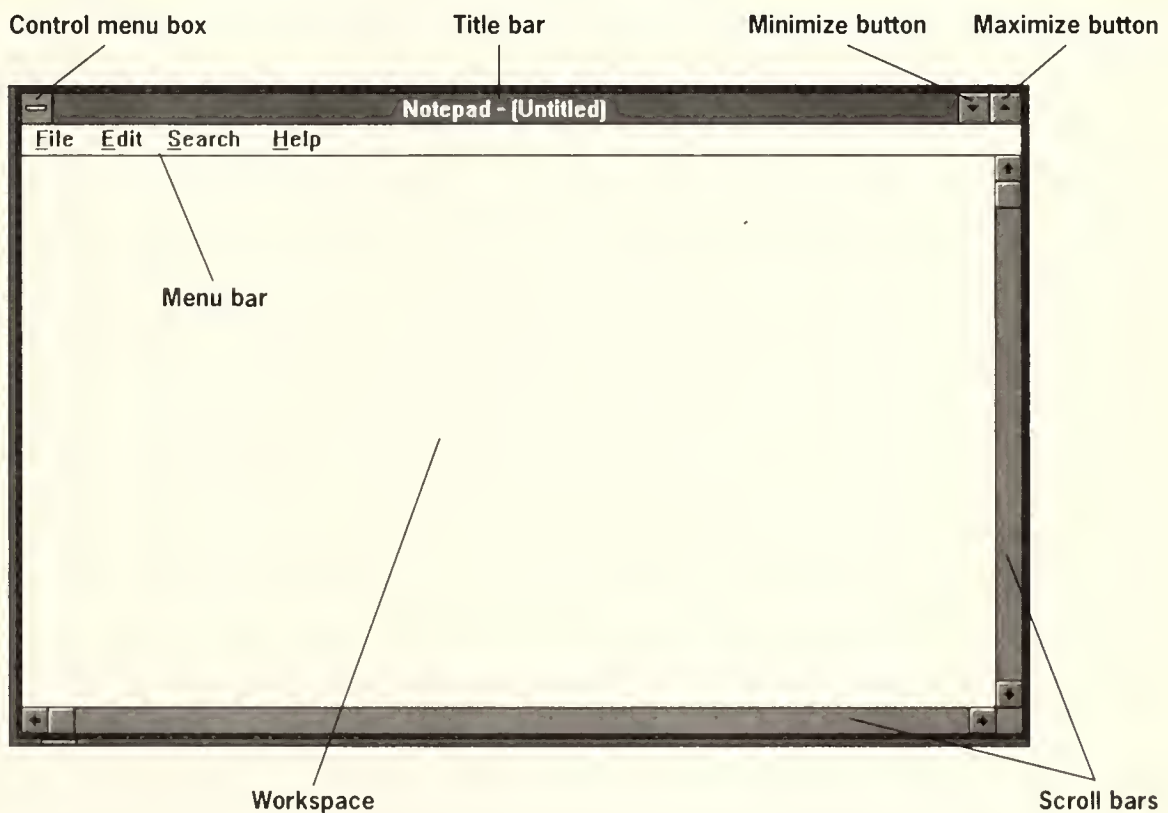
Nevertheless, the Windows interface remains the greatest single advantage that Windows applications have over DOS applications. How so? Because the standardization of the user interface among Windows applications makes the learning process much easier and faster. It might take just as long to teach a complete novice to use Word for Windows as it would Microsoft Word for DOS. But after having done so, you can teach the Windows-version user how to use any other Windows application in a tiny fraction of the time that it would take to teach the DOS-version user to learn another DOS application. The Windows user can build upon his or her knowledge of the Windows interface in learning new applications, whereas DOS application users often must try to *forget* all the user interface conventions used by one application in order to learn another.

As a developer of Windows applications, it is to your benefit to follow the standard Windows user-interface conventions at every opportunity. Doing so will simplify the learning process for any Windows user, which in turn convinces the user that yours is the simplest and most intuitive can opener he or she has ever seen.

Application Window Features

The most basic element in the Windows user interface is the *application window*, pictured in Figure 7.1. Nearly every Windows application makes use of a standard window—the primary exceptions being small utilities that communicate to the user entirely through dialog boxes.

Figure 7.1
A standard application window and its features



A standard application window is made up of several basic components, described in the following sections.

Title Bar

The *title bar* appears across the top of the application window, displaying the name of the application and, in some cases, the name of the document with

which it is working. For instance, the title bar for the standard Windows shell program always reads “Program Manager”, whereas the title bar for Windows Write reads “Write - (Untitled)” if the current document has not been saved to disk or “Write - SAVEDDOC.WRI”, for example, if the current document has been saved under the name SAVEDDOC.WRI.

Program Manager’s title bar doesn’t change because it works with the same data set—the program groups that you have established. Applications such as text editors, spreadsheets, databases, and so on, which use different data sets, should always display the name of their current document in the title bar because the name in the title bar also determines the caption of the application’s icon when it is minimized. (Incorporating the current document name into the icon caption makes it easier for the user to select the application and document with which he or she wants to work.)

Minimize and Maximize Buttons

The buttons that appear at the rightmost edge of the title bar, the *minimize* and *maximize buttons*, are used to shrink the application window to an icon and to enlarge it to fill the screen, respectively.

Most development tools give you control over whether your application’s window includes minimize and maximize buttons, and whether it can be minimized or maximized. Not all applications need maximize buttons. Many utility applications only need to display a small status screen or a dialog box from which to elicit user input, so there is no reason they should ever need to fill the screen. On the other hand, unless there is no possibility that the user will ever want to activate another application while yours is running—the likelihood of which I put at zero—you should always provide a minimize button.

Control Menu Box

Appearing at the leftmost edge of the title bar, the Control menu box is used to access the Control menu, a special menu that offers the user the ability to move, resize, or close the current application, or to access the Windows Task Manager. Some applications may modify the Control menu, adding new items to it or deleting some of the standard options.

Menu Bar

The menu bar, which appears directly beneath the title bar, contains the application’s menus. Most Windows applications include three or four drop-down menus, which follow a fairly standard organization.

- The Control menu (accessed from the Control menu box), offering the choices Restore, Move, Minimize, Maximize, Close, and Switch To
- The File menu, offering the choices New, Open, Close, Save, Save As, Print, Print Setup, and Exit
- The Edit menu, offering the choices Undo, Cut, Copy, and Paste
- The Help menu, offering the choices Contents, Search Help On, How to Use Help, and About *application name*

Of course, many applications omit some or all of these menus or options, and almost all add additional items to the standard list. For instance, an application that supports DDE might add a Paste Special command following the standard Paste command, and some applications add a list of recently opened files to the bottom of the File menu.

In addition, most applications add additional drop-down menus to the four standard ones. Unfortunately, there is almost no standardization of these menus. For instance, Microsoft Word for Windows, Write, Notepad, Lotus Notes, and Microsoft Excel all offer a Find option, to search for text or numbers within a document. Yet they place that item variously under menus called Edit, Find, Search, Options, and Data. So how is a poor user supposed to find Find?

Without a standard model for menu layout, it's not always clear how you should structure the menus in your application. But if you know that most of your users will be using the application in conjunction with or alongside another application, you should try as much as possible to model your application's menus after those of the other application, thereby injecting at least a little standardization into this sea of inconsistency.

Dynamic Menu Items

Windows provides several standard ways to update menus dynamically during the course of program execution. You can *gray out* or disable specific menu items, you can add and delete items, and you can place check marks alongside menu items.

Disabling a menu item is an easy way to indicate to the user that the function accessed by that item is not available at the current time. For instance, if the Clipboard is empty, the Paste item on the edit menu should be disabled because there is no data there to paste.

You can also disable menu items to indicate that there is no need to carry them out. For instance, PageMaker 4.0, which uses large, complex files that can take a minute or more to load and save, disables the File Save option until the user has changed the open document. This way the user doesn't spend time saving a file unnecessarily, although he or she can still use

the Save As option to save the current file under a new name or in a different location.

The ability to add or delete menu items offers several possibilities. It enables you to add a list of recently used documents to a file menu, or a varying list of available tools or functions to a utility menu. A communications program could list available connections, and a print utility the printers available to a particular workstation.

Menu check marks, meanwhile, enable you to make menu items work like toggle switches. For instance, a text editor's edit menu might include an item labeled "Word Wrap", as in the menu from Notepad shown here:

Edit	
<u>U</u> ndo	Ctrl+Z
Cu t	Ctrl+X
<u>C</u> opy	Ctrl+C
<u>P</u> aste	Ctrl+V
D elete	Del
Select All	
Time/ <u>D</u> ate	F5
√ <u>W</u> ord Wrap	

When the user selects the item for the first time, a check mark appears beside it, indicating that the option has been activated. Selecting it a second time turns the option off, and removes the check mark.

Cascading Menus

Cascading menus, or submenus, are a mixed blessing. On the one hand they offer a good way to organize related menu items under a single heading. Your application might include a menu item called Select, for instance, to which would be attached a cascading menu with a choice for each available printer, as shown here:

File	
<u>O</u> pen	
Sa <u>v</u> e	
S elect P rinter	LaserJet II on LPT1
E xit	Postscript printer on network

The problem with cascading menus is that they are frequently misused as a replacement for dialog boxes. Many applications force users to make choices from several different cascading menus, when they would be better served by a single dialog box on which they could make all those choices at once. For instance, in a word processor separate cascading menus for font, font size, type style, and alignment could be replaced by a single dialog box

called Type Settings—reducing the amount of times the user has to go back and reactivate the menu in order to adjust these settings.

This problem is exacerbated when developers attach cascading menus to cascading menus, so that the user has to go three or four menu layers deep in order to make a selection. It's tempting to do this sometimes, since it is easier and faster in most cases to add cascading menu items than to draw dialog boxes. But although cascading menus seem like a neat solution for the developer, they're generally cumbersome for the user, so you should avoid going overboard with them.

Scroll Bars

Horizontal and/or vertical scroll bars should appear along the bottom or right side of the application's window if the window is not large enough to display its entire contents (see Figure 7.1). The user can scroll the window by moving the scroll box, which represents the position of the current view relative to the whole file, or by clicking in the scroll bar or on the scroll bar's arrow controls.

Like the minimize and maximize buttons, scroll bars are of primary benefit to applications that display large amounts of data, which may not all fit onto the screen all at once. They are generally not needed for small utility applications.

Application Workspace

Windows applications that are capable of working with only one document at a time, such as Notepad or Paintbrush, display the contents of that document in the *application workspace*—which consists of the window area beneath the application's menu bar (Figure 7.1). Some applications, such as Paintbrush, devote some of this area to displaying tool palettes or status bars, but the remainder of the area beneath the menu bar is used to display the contents of the current document.

Document Windows

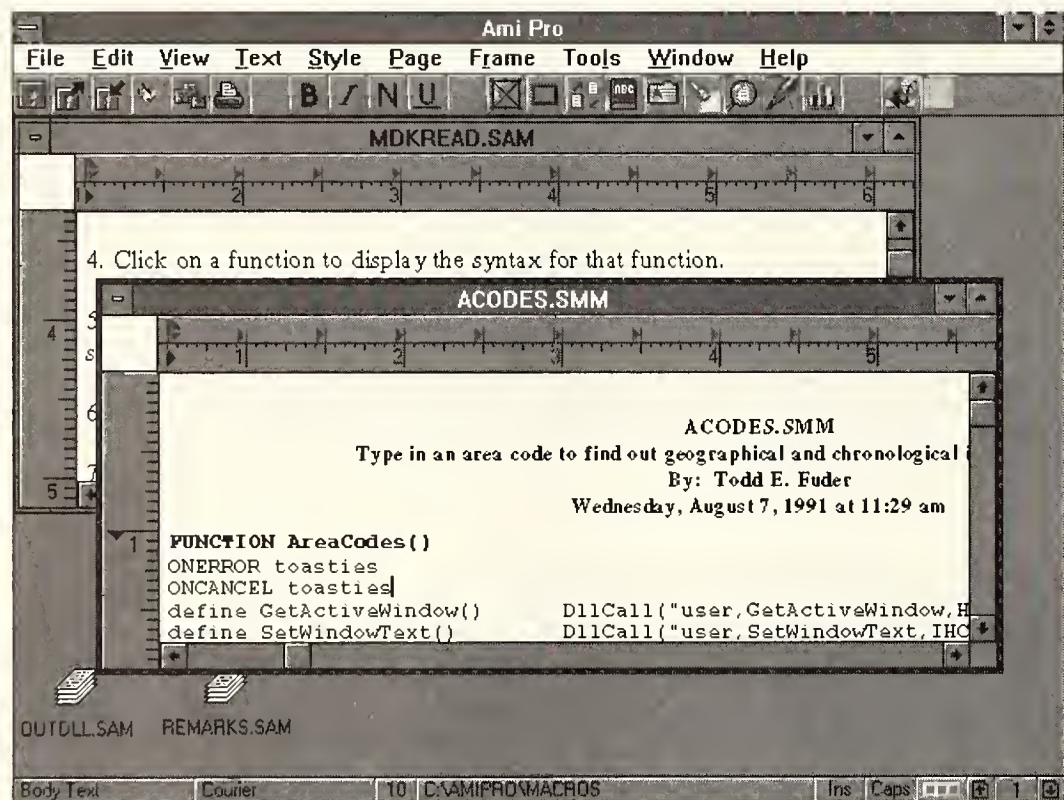
Windows applications that can work with more than one document at a time generally utilize the *multiple-document interface* (MDI)—a variation of the standard application window in which each document appears in its own sizeable, movable, minimizable window within the application workspace, as shown in Figure 7.2. Each open document in an MDI application can be enlarged to fill the application's workspace, minimized to the size of an icon at the bottom of the workspace, or moved anywhere within the workspace.

The multiple-document interface can be used for applications that work either with multiple independent documents (such as word processing

documents or spreadsheets), or multiple views of the same data (such as list and browse views of a database or directory and file lists for a disk drive). It should not be used to represent different parts of your application. For instance, floating palettes, such as the style palette in a word processor or the drawing-tools palette in a graphics program, should either be visible to the user or hidden, but not minimized. Minimization within the multiple-document interface should be reserved for documents or document views, not program screens.

Figure 7.2

An application utilizing the multiple-document interface



Standard User-Interface Controls

Along with a basic standard for laying out windows, the Windows interface also features a standard set of user-interface controls.

Command Buttons

The most basic user-interface controls are *command buttons*, also referred to sometimes as pushbuttons or action buttons. The purpose of a command button is to provide the user with an easy way to initiate an action. So when the user clicks a command button, your application should immediately respond by carrying out the action associated with that button.

The most common command buttons are the OK and Cancel buttons found in most Windows dialog boxes, as shown in Figure 7.3.

Figure 7.3
Dialog box with OK
and Cancel buttons



These buttons are used to signal that the application should proceed with or abandon the tasks indicated by the user's actions within the dialog box. The standard convention is to make the OK button the default button for the dialog box (meaning that it will be selected anytime the user presses the Enter key), and to make the Cancel button the one that will be selected anytime the user presses the Esc key.)

Command buttons may also be used in any other situation in which you want to give the user the opportunity to initiate an action. For instance, an application's main window may contain a row of buttons that are used to jump to other parts of the application or to provide quick access to commands that normally would be accessed from a pull-down menu.

Check Boxes

A *check box* is an on-off toggle switch. Check boxes are used to offer the user the option of activating or deactivating a particular program feature or option.

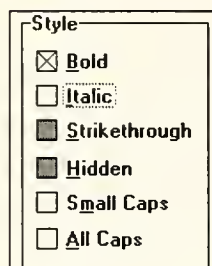
Check boxes may be presented singly or in groups. When presenting them in groups, however, remember that the options they offer should be compatible. For instance, it would make sense to use a pair of check boxes to give users the ability to select text attributes such as boldface or italic, as shown in Figure 7.3, since text can be both bold and italic at once. However, you shouldn't use check boxes to present a choice of text-alignment style, since text cannot be left-justified and centered, for instance, at the same time.

Grayed Check Boxes

Although most check boxes are used for two-state on-off options, in which the checked state turns the option on and unchecked turns it off, there is actually a third check box state: grayed. This state is traditionally under program control—that is, that the application itself, rather than the user, turns a check box gray to indicate that the choice it presents is inapplicable to the current situation. Most other user interface controls—including buttons,

menu choices, and icons—can also be grayed in this manner to indicate that an option is not available at the present time.

Recently, however, some applications have started to make another use of the grayed state for check boxes, providing a way for users to set the check box to that state in order to indicate that they don't care whether the option or state that it represents is active. For instance, Word for Windows utilizes this capability in a dialog box that allows users to search for particular styles of text within a document, as shown here:



For each of several attribute options, the user can either leave the option grayed, or click it once to search for text with that attribute, or click it twice to search for text without that attribute. Leaving any choice grayed indicates that the user doesn't care whether or not the search text has that attribute.

Radio Buttons

Radio buttons, also known as option buttons, are used to present the user with a choice of mutually exclusive options. One might, for instance, use a set of three radio buttons to allow the user to specify whether text should be left-aligned, right-aligned, or centered, as shown in Figure 7.4.

Figure 7.4

A group of radio buttons



Radio buttons should always be presented in groups, and it should be possible to select only one option in the group at any one time. So if the user selects the option button labeled "Centered" in the example above, both the Left and Right options should be automatically turned off.

Radio Buttons or Check Boxes?

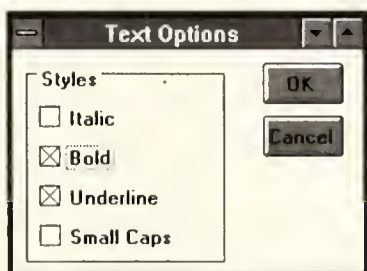
It may not always be apparent when you start to design a dialog box or input screen whether you should use radio buttons or check boxes to present a given option. For instance, it's not uncommon that an option seems to be a simple yes or no choice—and thus perfect for a check box—when you begin. In the course of the development process, however, your design grows more complex, so that in the end you need to present the user with a set of mutually exclusive options in the form of radio buttons. For example, a simple check box in a text editor for turning word wrap on and off might evolve into a series of radio buttons for specifying text-alignment options.

Changing your application to incorporate radio buttons in place of check boxes, or vice versa, is usually fairly simple, especially if the development tool you're using includes a graphical dialog box or screen editor. More difficult is recognizing the need to make that change when it arises. Just remember, check boxes should be used when you have one or more options that need to be toggled on and off, whereas radio buttons are the interface element of choice when you have two or more options that must be dealt with in a mutually exclusive fashion.

Group Boxes

A *group box* is a graphical dialog box element consisting of a label and a rectangular box that holds other user-interface elements. Generally a group box is used to indicate that the options within it are related. For instance, you might use a group box to present a group of radio buttons or a series of related check boxes. Then you could use the group box label to identify the general purpose of those options, as shown in the group box labeled “Styles” in Figure 7.5.

Figure 7.5
Using a group box
to present related
options



Icons

If you use Windows, you know what *icons* are: little pictures that generally do something when you click on them. I say “generally,” because some icons are purely informational: they serve as warning lights rather than prettified command buttons.

Icons offer two primary benefits to the application developer: They can attract the user’s attention, and they can help you to conserve screen real estate. The degree to which you can capitalize on those benefits will depend on how well you understand the capabilities and limitations of icons. Both can be seen in the most famous icon of all, the Macintosh’s Trash icon:



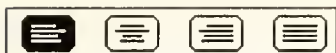
The trash can is a marvelous example of an icon. It stands out on the screen, and its primary use is immediately obvious: This is where you put files that you no longer want. Novice users grasp that idea immediately. And certainly, the trash can image gets that idea across in far less screen space than a button labeled “Repository for No-Longer-Needed Files,” and stands out on the screen far better than a plain button labeled “Trash.”

Nevertheless, the Trash icon falls short of conveying all the information the user needs to know about its operation. Nothing about the icon indicates, for instance, that the user will have to select the menu item Empty Trash before any files are actually deleted. Nor is there anything about its appearance to suggest that you can also eject diskettes from the Macintosh’s internal drive by dragging them onto the Trash icon.

Of course, these are things that the Macintosh user learns quickly, and once learned, they are seldom forgotten. The Trash icon makes so much sense for its purpose that it reminds the user of everything else necessary to its operation. This suggests an important rule about the design and use of icons: Their effectiveness rests as much upon the user recognizing and interpreting them in the context they appear in as on any intrinsic message conveyed by the icon itself.

However, even the best-designed icons convey only a limited amount of information. The more complicated the task, the more difficult it is to completely represent it through an icon, and the more you must rely on the user to interpret the icon in context.

For an example a little closer to home, consider the four text-alignment options presented as icons on the ruler in the Windows Write application:



These icons do a good job of communicating their meaning and of offering the Write user access to a common word processing function. But you still couldn't call them 100-percent effective at conveying their meaning. An uninitiated user might suppose, for instance, that they are used to indicate that you want to draw horizontal lines of varying lengths on the screen. And even after their general meaning is known, there is nothing about their appearance to convey that the actions they initiate will affect either currently selected text or any new text entered after the current insertion point, but not any other text already entered in the document.

Nevertheless, the text-alignment icons in Write are generally effective. Once the user has learned their purpose, it is easily recalled because the appearance of the icons suggests their meaning. And they thus provide an effective shortcut to the program's text-alignment functions.

Icons lose effectiveness quickly when you display too many at once, or when the images are so abstract that the user can't recall their meaning from one moment to the next. For instance, version 2.0 of Abacus' Becker Tools for Windows suffered greatly on both these counts, as Figure 7.6 illustrates. Its main screen featured no less than 65 icons, many of them highly abstract. Not surprisingly, the program's interface has since been redesigned.

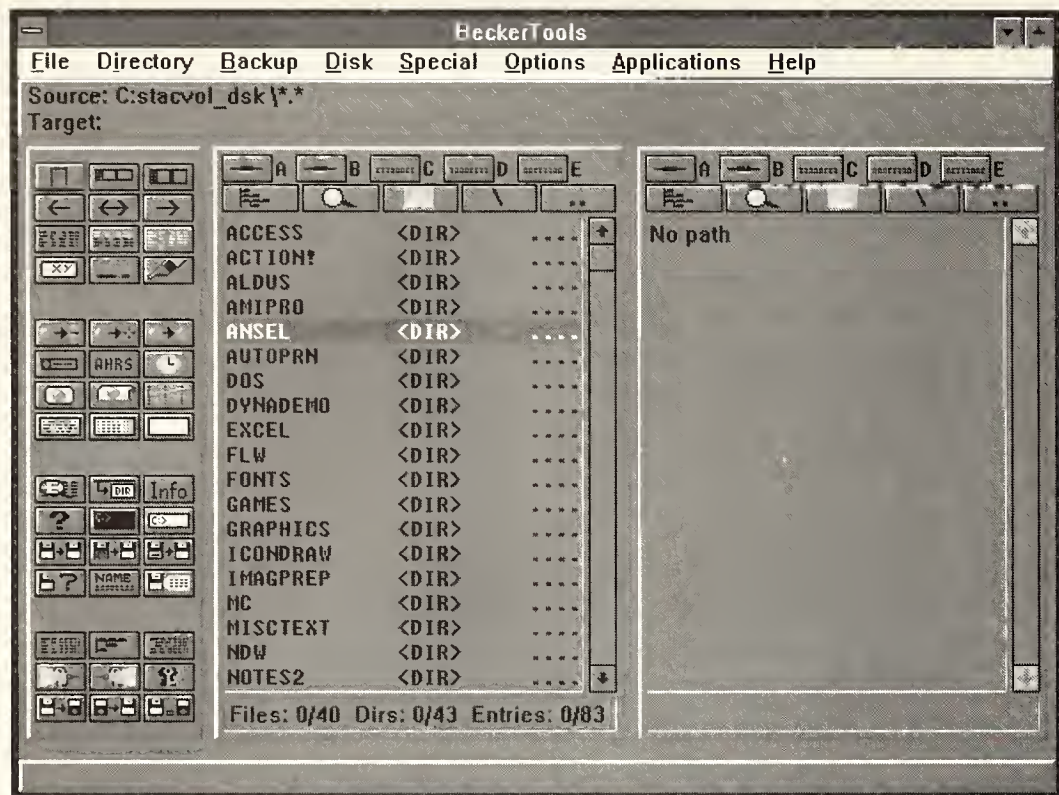
Informational Icons

In addition to action icons, which function as command buttons in disguise, some applications make use of icons for informational purposes only. These are primarily used in message boxes. The IBM Systems Application Architecture guidelines (discussed below) call for four standard message box icons:

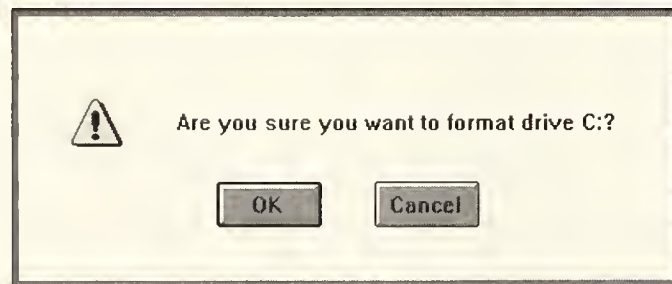
- A lowercase "i," indicating that the message is informational only, such as "File transfer complete."
- An exclamation point, !, for warning messages such as that shown in Figure 7.7.
- A question mark, ?, for questions.
- A stop sign, for messages that a problem has occurred that requires a response from the user, such as "File not found. Please insert a disk containing the file into drive A and press Enter."

Figure 7.6

How not to use icons

**Figure 7.7**

A warning icon



For icons of this type to be effective, the user has to see them frequently, so that they cause a sort of conditioned response. Even then, however, they don't carry a great deal of weight, so don't assume you can do without explanatory text in your dialog boxes.

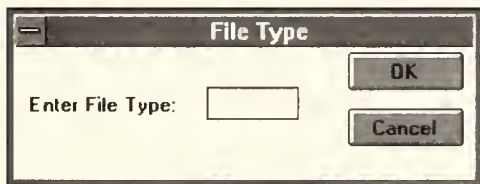
Static Text

Static text controls, or labels, are simply blocks of text that are used to identify the purpose of other controls or to provide instructions to the user. For

example, you might place a label that reads “Files” above a list box that contains a list of files.

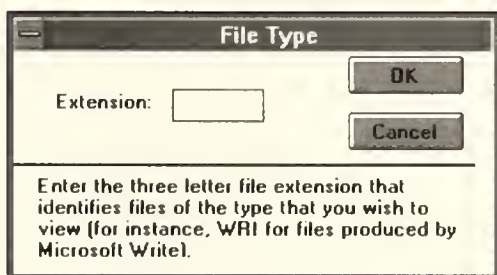
There seems to be an inherent tendency among programmers to be as terse as possible within dialog boxes. Maybe this stems from a natural aversion to typing, or maybe it’s a holdover from the days of trying to cram applications into 64k PCs, but either way it results in dialog boxes that aren’t as clear as they could be, or should be. The fact is, memory isn’t all that precious anymore, and a few extra keystrokes can make your dialog boxes far clearer.

Compare, for instance, the two dialog boxes that follow, both of which are designed to elicit a file extension from the user. The first is an example of the “less said the better” school of dialog box design:



The problem with this dialog box is that its meaning and operation will only be clear to someone who knows exactly what information it requires. For the casual or new user, it is a mystery—one that will require at least some experimentation to solve.

The second example does a far better job of communicating exactly what information it requires from the user, thanks to the addition of a few lines of explanatory text:



The addition of this text to the dialog box won’t slow down the experienced user at all, and will only cost a few dozen extra bytes of disk space, yet it all but eliminates the chance that the new user will be completely befuddled by the dialog box’s meaning.

Edit boxes

Edit boxes are used within a dialog box to allow the user to enter strings of textual information.

Edit boxes come in many varieties: single- and multiple-line, with and without horizontal or vertical scroll bars, and accepting limited or unlimited amounts of text (or nearly unlimited—depending on the development tool you're using, text boxes are actually limited to either 32k or 64k of text). Depending on which of these options you select, you can use edit boxes for everything from obtaining a one-word database field entry to serving as the core of a simple memo or message editor.

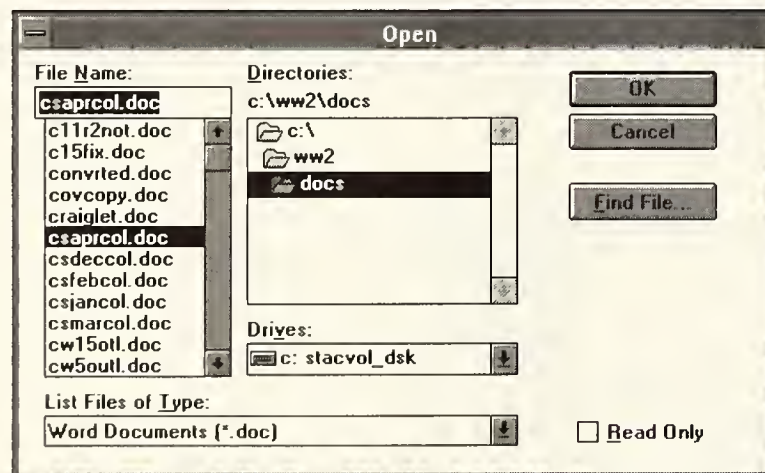
List Boxes

List boxes are a great tool for providing the user with a range of choices: Which document do you want to print? Which telephone number do you want to dial? And so on.

The beauty of using list boxes for this purpose is that you don't need to know in advance how many elements the list box will contain. There may be 5 or 500 files to choose from or telephone numbers to dial—it doesn't matter. The list box presents them in sorted, ascending order (if you select the option to sort the list box offered by most development tools), allowing the user to scroll through the list to find the item that he or she wants. Figure 7.8 shows one such list box.

Figure 7.8

A list box sorted in ascending order



There are some times, of course, when you do know exactly how many choices the user will have. If you're implementing binary file transfers in a communications application, for instance, and your application only supports the XMODEM, YMODEM, and Kermit protocols, then you know that the user will always have a choice of those three protocols.

In a case like that, however, where you have a known, finite number of mutually exclusive choices to present, you're better off using radio buttons

than a list box. Although both methods elicit a choice from the user, the use of radio buttons would be preferable in this instance because it is more precise: It takes advantage of the user's knowledge of Windows conventions to signal that the choices are both constant and mutually exclusive.

List boxes come in three standard varieties: single-selection, multiple-selection, and extended-selection.

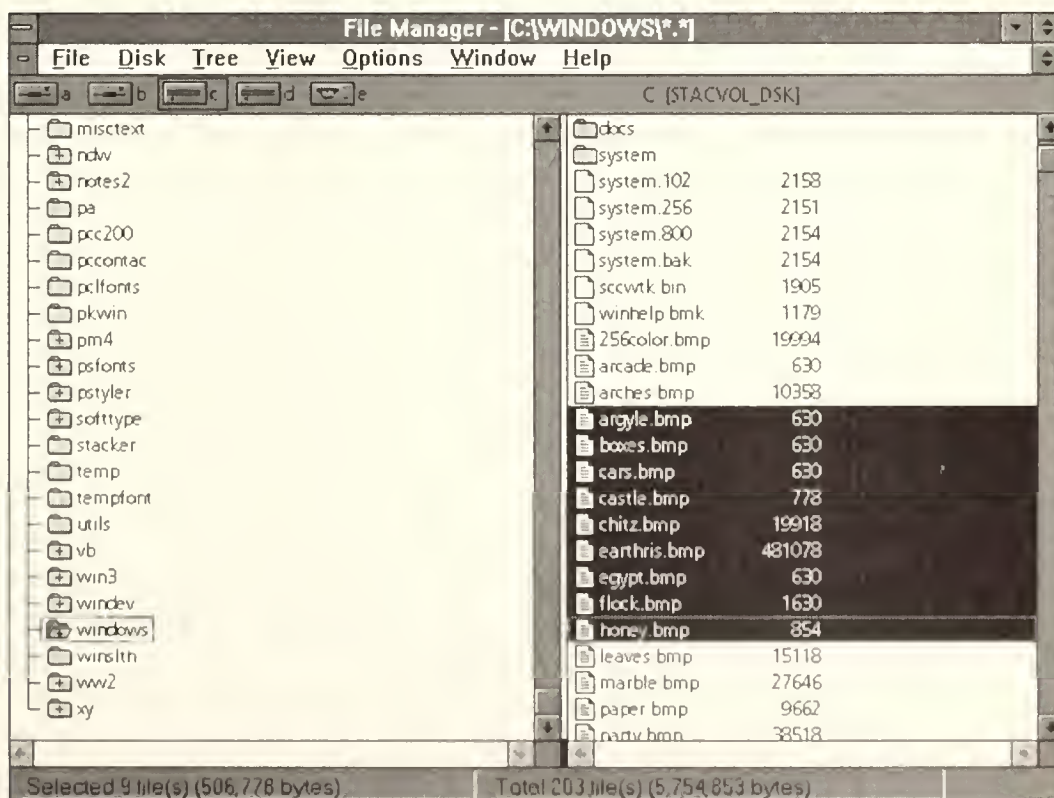
Single-selection list boxes allow the user to select only a single item from the list at a time. The standard File Open dialog box, shown in Figure 7.8, is a good example of a single-selection list box. Although it may present a list of hundreds of files, you can select only one at a time.

Multiple-selection list boxes offer the user the opportunity to select multiple items from the list. Any item in the list can be selected by clicking on it once. Clicking on an already selected item deselects it.

Extended-selection list boxes also offer the user the ability to select multiple items, but in a different manner. With an extended-selection list box, users can select a range of contiguous items by clicking on the first item in the range and then shift-clicking on the last item. Or they can select multiple, noncontiguous items by clicking on each one while holding down the Ctrl key. File Manager's file windows are an example of extended-selection lists (see Figure 7.9).

Figure 7.9

File Manager displays files in an extended-selection list



Which Style to Use?

Generally, multiple- and extended-selection list boxes should be used when you want to give the user the opportunity to indicate several items that should be affected by the next command he or she issues.

Which of these styles to use, multiple or extended, is pretty much a matter of personal choice, since both accomplish largely the same ends. Personally, I find multiple-selection list boxes annoying, since I often end up inadvertently selecting multiple items when I am merely trying to move the cursor within the list box by clicking on another item. So I prefer extended-selection boxes. But, on the other hand, the multiple-selection style does have the advantage of not requiring the use of Ctrl- or Shift-key combinations to select multiple items.

Unfortunately, you often don't have a choice of style. Many high-level languages only offer the ability to create single-selection list boxes. For instance, in the MCI mail management application described in Chapter 15, I would rather have used an extended-selection list box to allow the user to simultaneously select multiple mail messages to print, delete, or move to another storage folder. Unfortunately, the DynaComm script language I used to build the application doesn't support multiple- or extended-selection list boxes, so I was unable to do so. (I was able to work my way around at least part of the problem, however, by allowing the user to mark messages in the list box for later deletion.)

One thing you should keep in mind if you use multiple- or extended-selection list boxes is that their implementation in Windows is somewhat flawed from the perspective of user-interface transparency: There is no way the user can tell from looking at a list box which of the three varieties it is. As a result, there is a good chance that the list box will not act the way the user expects it to—which is, of course, not recommended in terms of user-interface design.

On the other hand, regrettable as this inconsistency may be, there are times when it makes so much sense for the user to be able to select multiple items from a list, that the inability to do so will cause more anger or frustration than the brief confusion of “Now do I click or Ctrl-click or what to make this work?”

Displaying Big Lists

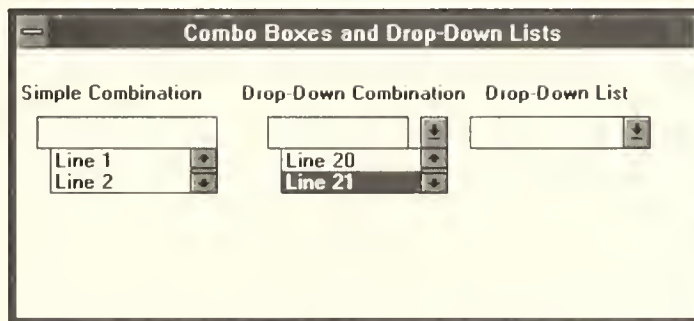
If you're going to use a list box to present a long list of files or other items, make the list box as big as you can. Many applications try to conserve screen space by using list boxes in which only four or five items are visible at once. As a result, the user may have to scroll or page down through dozens of items to find the one he or she wants. With a bigger list box, showing more items on screen at once, the user can simply click on the list box's scroll bar to jump 15 or 20 items down the list at a time—a far friendlier solution. So when a list box is going to be used to present large lists, make it as large as practical.

Combo Boxes and Drop-Down Lists

There are three closely related types of controls in this category:

- The *drop-down combination box*, which consists of an edit box joined to a drop-down list box
- The simple *combination box*, which consists of an edit box joined to a list that is always visible
- The *drop-down list box*, which resembles the drop-down combo box but omits the edit box

All three controls are shown in Figure 7.10.



The two combination-style controls are useful when you want to present users with a list of predefined choices, but also allow them to enter a new data item. The drop-down list box, conversely, should be used when you want to limit the user to a predefined set of choices. So, you might use a drop-down combo box for a File Save As... dialog box, from which the user could either pick an existing file name or enter a new one; and you would use a drop-down list box for a File Open dialog box from which the user would select only existing files.

The choice of whether to use a simple combo box or a drop-down combo box depends mainly on the number of predefined choices that you'll be presenting. The simple combo box is ideal if you'll never be offering more than three or four predefined choices, since you can make the list portion of the box large enough to display all those choices at once. If the set of choices is going to be much bigger than that, though, you'll be better off using a drop-down combination box and letting the user scroll through the list.

That concludes our discussion of the standard Windows user-interface elements. Before going on to look at nonstandard elements, let's take a brief look at where the standard originated.

The SAA Standard

All of the window elements and user-interface controls discussed so far are codified in a remarkable 1989 publication from IBM called *Systems Application Architecture Common User Access Advanced Interface Design Guide*—the SAA guidelines for short.

The SAA guidelines grew out of an attempt to formulate the principles and rules by which user interfaces should operate on computing devices—everything from dumb terminals linked to remote mainframes to the most powerful PCs and workstations. Eventually the decision was made to produce separate guidelines for programmable devices (PCs and workstations) and nonprogrammable ones (terminals), resulting in the publication of the work cited above for programmable devices.

The SAA guidelines are amazingly detailed, specifying everything from the keystrokes to be used for selecting text to the order of items on File and Edit menus. As such, they represent the closest thing we have to a set of rules for designing Windows applications.

Anyone who sets out to design Windows applications will find these guidelines of interest. However, I wouldn't recommend that you take a blood oath to follow them. Certainly, you should whenever possible—and think twice before violating them—but don't let their restrictions prevent you from implementing a better idea. It is far too early to lock user-interface design into a set of immutable rules. Too much creative innovation is still going on in user-interface design—and we still have too much to learn about what works and what doesn't—for any attempt to limit developers to a set of iron-clad rules to succeed. In this regard, the SAA's attempt to codify the PC user interface was doomed to failure from the start.

Nevertheless, the SAA guidelines are well worth studying. Even though some of the detailed elements are looking increasingly archaic, the basic principles espoused in the guidelines remain as valid today as they were in 1989: The user should feel confident that he or she understands the interface and how it will react, and the user should always be in control.

To achieve this, the SAA guidelines preach a message of consistency, immediate feedback, modeless operation (the same action should have the same result at every point in a program), and constant encouragement of the user's desire to explore. Make the interface transparent, the guide advises developers, allow the user to control the dialog, and make the interface forgiving.

There's always the chance that, like Woody Allen in *Sleeper*, we'll wake up one day to a world in which conventional wisdom has been turned on its head. Perhaps in that future steak and ice cream will be health food and the principles of good design will call for confusing users at every opportunity. But until then, the SAA guidelines offer as fine an exposition of the principles of good user-interface design as you'll find anywhere.

The *SAA Common User Access Guide* is included with the Microsoft Windows SDK, or may be obtained from your local IBM sales office. Ask for publication SC26-4582.

Common Extensions to the Standard

As the Windows environment has evolved and matured, there have been some additions to the basic set of user-interface controls. A new element would pop up in one application, then somebody else would take inspiration from it in designing another application, and before you knew it, every Windows application worth its salt had to have a ribbon or an icon bar.

These new elements earned their place as part of the Windows standard for a good reason: They work. They make sense to users and developers alike, because they make applications easier to understand and use.

Icon Bars

Perhaps the most ubiquitous of the new additions to the Windows interface is the *icon bar*—although the name changes from vendor to vendor. Microsoft calls it a toolbar, whereas Lotus calls the collection SmartIcons, and other vendors have come up with their own names. What all the variations boil down to though, is a selection of icons running across the top, bottom, or one side of the application's window, offering single-click access to program functions such as file opening, saving, or printing, or to macros. Microsoft Word for Windows' toolbar is shown here:



In some applications, the user can define the arrangement and purpose of each icon, whereas in others the set of icons is fixed. In either case, however, the icon bar has proven successful because of the immediate access it offers to program functions. With an icon bar available, there is no need for the user to hunt through menus to activate common program functions. Instead they're just a mouse click away.

Ribbons

Ribbons are closely related to icon bars, but are reserved for providing access to formatting commands, such as text styles and font names in a word processor or alignment options in a spreadsheet program.

Most ribbons, like that of Word for Windows (shown below), feature a combination of icons and other controls.

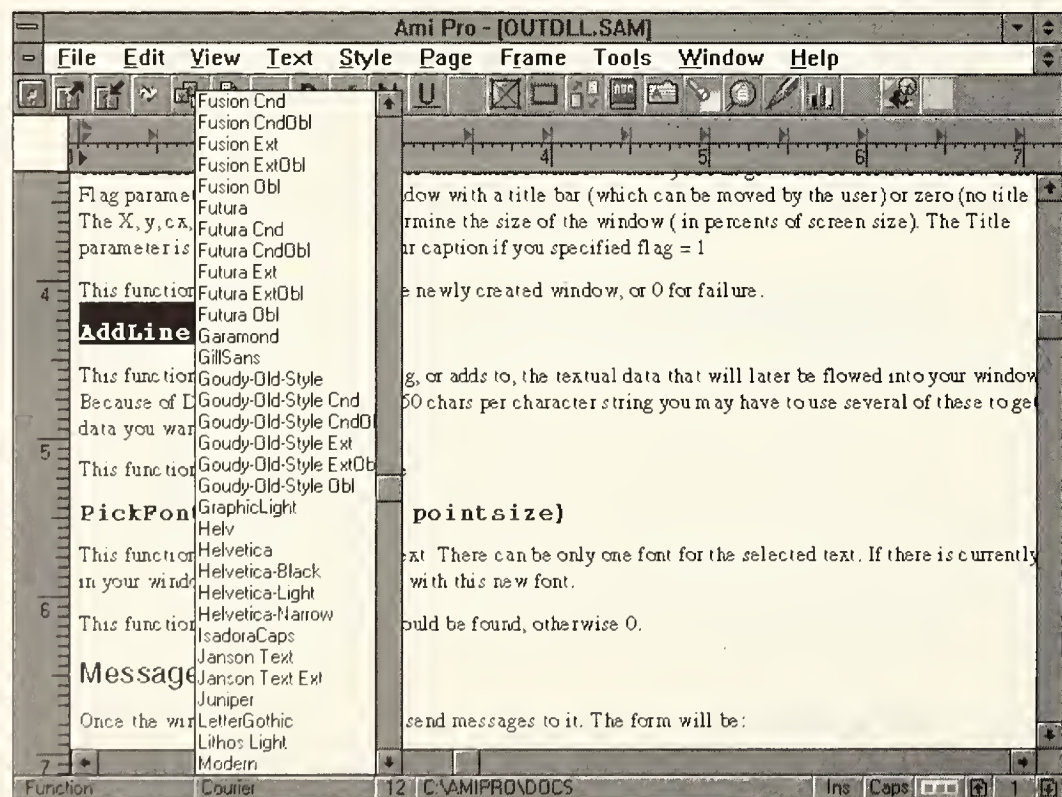


Active Status Lines

A somewhat less common recent user-interface invention is the *active status line*, which is used to its best advantage in Lotus applications such as Ami Pro and Freelance Graphics for Windows. Many Windows applications use the bottom line of their window to display document status information, such as page count or cursor position in a word processor, but generally they present that information as static text. Recently, however, some applications have turned this area into an active control, so that, for instance, when you click on the page counter a dialog box opens asking what page you want to go to; or if you click on the current font indicator a list of available fonts appears, as shown in Figure 7.11.

Figure 7.11

Status bar font menu in Ami Pro



Beyond providing the user with the same one-click access to commands as an icon bar, using the status bar in this manner is a good way to conserve screen space. Icon bars, status lines, ribbons, and menus all take valuable space away from the main document window, reducing the amount of the document that is visible to the user. The more you can put elements such as the status bar to double use, the more room you can reserve for the document, which is, after all, the user's prime concern.

The Keyboard Interface

The keyboard interface is easy to overlook in the user-interface design process. When you think of Windows applications, you probably think first of drop-down menus, scroll bars, and icons. Ctrl-key combinations come way down on the list. Nevertheless, the keyboard shortcuts and conventions used in your application will have a big effect upon its usability.

With the exception of freehand drawing applications, which require a mouse or other graphical input device, every aspect of every Windows application should be available from the keyboard. Providing complete keyboard access to your application will make it easier for notebook computer users and others operating without a mouse, and for those users who simply prefer to keep their hands on the keyboard.

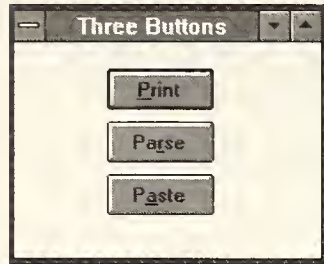
Keyboard Navigation

The most critical element of the keyboard interface is the user's ability to navigate through every dialog box, screen, and menu from the keyboard.

With most application-development tools, providing keyboard access to dialog box controls or menus is as simple as adding an ampersand symbol to the letter in the control's label or the menu item's name that, when pressed simultaneously with the Alt key, will be used to access that item. (The specified letter will be underlined when the control or menu is displayed.) For instance, the letter "P" would be underlined in a button named &Print, and the button would be accessed anytime the user pressed the Alt-P key combination. Conversely, one named Print would have no underlined letter and no way to be accessed via an Alt key combination.

Whenever possible, it's best to use the first letter of the menu item's name or the control's caption in the Alt key combination. Of course, if you've got buttons named Print, and Paste, and Parse, you can't use Alt-P for all three. So when more than one control or menu item starts with the same letter, you've got to pick other letters in the names of some of them, preferably letters the user will remember easily. For instance, you might

choose to underline the “r” in Parse, since the “r” sound is so dominant in its name, or the “a” in Paste, so that the three buttons look like these:



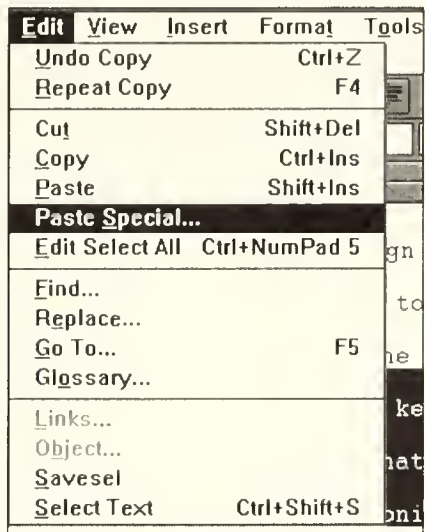
Most Windows development tools automatically provide several additional kinds of keyboard-navigation support, for instance, pressing the Alt-Spacebar combination to activate the menu bar, or pressing the Tab key to navigate from one control to another in a dialog box. Depending on the development tool you use, you may or may not have the ability to modify these, such as by modifying the order in which the Tab key selects dialog box items.

Keyboard Shortcuts

Beyond providing for keyboard access to menus and control structures, you may also want to provide shortcuts for directly accessing specific program functions. For instance, rather than having the user first select the Edit menu, and then the Paste item, most Windows applications that offer a Paste function provide direct access to it via the Shift-Ins key combination. Similarly, the combinations Shift-Del and Ctrl-Ins, for Cut and Copy, respectively, are used in many Windows applications, as shown in the Word for Windows menu pictured in Figure 7.12.

Keyboard shortcuts enable users familiar with your program to improve their speed and efficiency, since they don't have to maneuver through several levels of menus and dialog boxes.

Like icon bar options, keyboard shortcuts provide direct access to program functions. So it is a good idea to provide them for any functions that will be used frequently, and especially when you want to enable the user to reissue a command using the same parameters as were used for the previous instance of the command. For example, in most applications the user needs to go through at least one menu layer and one dialog box to initiate a search-and-replace function—which makes sense since the user needs to supply the text to search for and the text with which it should be replaced. But there is no reason to make the user go through the same menu and dialog box just to replace another instance of the same search text with the same replacement text. Instead, you should provide a shortcut key that will execute the Replace Next function immediately.

Figure 7.12Word for Windows'
Edit menu

Dialog Box Design

With the WYSIWYG dialog box editors that accompany most Windows development tools, laying out the controls in a dialog box is more of a graphic design project than a programming task. These editors generally allow you to position and resize the dialog box, add controls anywhere within the box, and move or resize controls to your heart's content. Thus, there is a large subjective element to dialog box design. Programmers tend to lay out dialog boxes according to what looks good or makes sense to them.

Nevertheless, there are some basic rules that, while not cast in stone, are useful to follow as you design dialog boxes:

- Direct the user's attention to the most important element in the dialog box. A dialog box might have a dozen buttons or check boxes or icons, all of which act upon the item that the user selects from a list box. In that case, the list box is obviously the most important element in the dialog box. So don't hide it in a corner, and don't use a drop-down list. Make the list box big and central, so that the user will understand right away that it is the most important element. You might also want to separate it from the other elements with an outline or rule.
- Group related elements. If your dialog box has several check boxes, radio buttons, or edit fields, all of which concern one aspect of the program's operation, arrange them in a group so that the user will understand right away that they are related. Use a group box to separate them from unrelated elements, or leave space between them and other elements.

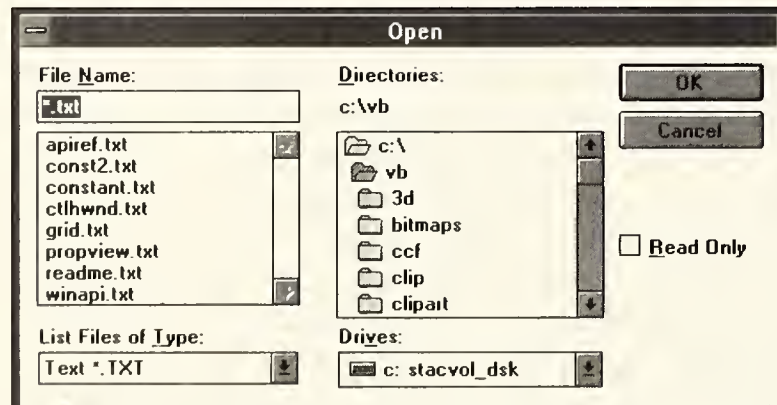
- Arrange elements in a sensible order. Keyboard users will want to tab between items in the dialog box. Don't confuse them by having the cursor jump all over the place when they hit the Tab key. Instead, move the cursor in a progression from the upper-left control to the one at the lower right, and then to the dialog box's OK and Cancel buttons.
- Establish internal standards for dialog boxes. There may not be any PC industry-wide established standards for the appearance of dialog boxes, but that doesn't mean you can't establish standards for the ones you design. As much as possible, you should come up with a standard dialog box layout and stick with it for every dialog box you design. Always put the OK and Cancel buttons in the same place, and always use the same structures (lines, group boxes, and so on) to isolate related elements and highlight key elements. You might want to model those standards after those used by another application that your users are already familiar with.

Using Common Dialog Boxes

The lack of standardization of dialog boxes may be easing, thanks to the inclusion in Windows 3.1 of standard dialog boxes for opening and saving files, printing files, and selecting fonts or colors. These are provided in the form of a DLL called COMMDLG.DLL, which is found in the WINDOWS\SYSTEM folder. Applications can utilize these dialog boxes (one of which is shown in Figure 7.13) by calling the COMMDLG.DLL file.

Figure 7.13

COMMDLG.DLL's dialog box for opening files



Putting It All Together

That concludes this general discussion of Windows user-interface elements. Chapter 8 describes the process of prototyping and testing your applications. Then, the programming projects described in Chapters 9 through 15 will provide hands-on examples that put these elements to use in actual application-design projects.

C H A P T E R

8

**Prototypes,
Testing, and
Documentation**

*Prototyping
Testing and Debugging
Documentation
Shrink-Wrap Time*

ONCE YOU'VE FINISHED DEFINING ALL THE DATA STRUCTURES, INPUT requirements, and internal procedures for your application, and have designed an effective user interface, the time finally comes to start turning all that work into a finished application. However, there are three more steps that you must complete before you're done with the program-development process: developing a prototype, testing and debugging, and documenting the application.

Prototyping

In the traditional world of program development, an application prototype serves the same sort of purpose as a pilot episode for a new television series: It gives the sponsors a chance to see how things are going to turn out before they commit big money to it.

Developing large applications in C or Fortran or COBOL is both expensive and time-consuming, so the traditional systems-development cycle calls for the programmer to first build a nonworking or limited-function prototype. Often the prototype will be built using tools that allow speedier development. On a PC, for instance, the programmer might use BASIC or Pascal (or an enhanced slide show application such as Dan Bricklin's Demo II from Intersolv, Inc.) to build the prototype.

When complete, the prototype is presented to the client, the manager in charge of the project, or the eventual user of the program to solicit suggestions and comments. Then the programmer modifies the prototype and presents it for review again, and so on until a final design is approved. Once that approval has been received, the programmer starts all over again, rewriting the application from scratch in C or COBOL or whatever language the project is to be built in.

That sounds like an awful lot of work, and it is. But it is also a necessary part of large programming projects in which the programmer won't be a user of the application, and the intended user isn't a programmer. One reason for this is, as you'll undoubtedly discover if you start writing applications for other people to use, that most users don't know what they want from an application until you give them what they think they want.

In other words, if you ask a group of people to describe precisely the application they want, and then you give them exactly that application, it won't be what they want. Instead, once the application is in front of them they'll ask you to move this and change that and, "How can we make this work a little better?" So it makes sense to turn out a prototype as quickly as possible and let the users take whacks at that, rather than invest too much time and money trying to produce a finished application before you've obtained approval of a prototype.

Iterative Prototypes

Just as word processing forever changed the writing process—eliminating, for most writers, multiple formal drafts and replacing them with an ongoing process of revision—so too the Windows development tools described in this book have changed the application-development process. Rather than having to go through a two-stage process of first producing a prototype (or a series of prototypes) and then producing the actual application, you can build your application in stages, adding functionality and making changes as you go along. Among other benefits, this allows you to get the user involved very early on in testing code that will eventually form the final application. So rather than build prototypes, you build iterative versions of your application.

This approach is possible because Windows reverses a key part of the traditional application-development process. Whereas in the character-based world it made sense to start off by defining an application's internal processes and procedures and then build up a user interface around them, in Windows it makes sense to build your applications from the user interface down. The user interface—windows, buttons, menus, and dialog boxes—becomes the framework on which you hang the code that actually puts those interface elements to use.

Of course, you can't entirely reverse the process, because you need to have at least an idea of the tasks that the application will perform and how it will go about them before you can design its user interface. But you don't need to nail either of these down entirely—instead, you can combine loose ideas about how the application will work with a solid user-interface design, and then go on from there.

The first iteration of a program produced in this manner might be what appears to be a complete application, except that every button click or menu choice results in a message box that tells the user, "This feature is not yet implemented." As you write more and more code, those messages will be replaced with working subroutines.

Why Do It This Way?

There are three major advantages to this user-interface-down approach to coding applications. The first is that it gives you the greatest amount of time to observe how users react to your application's user interface and to fine-tune it based on those observations.

You might wonder what real benefit there is to adding a nonfunctional menu item or button to an application in progress, but even a nonfunctional menu item is going to jog your ideas—and those of anyone testing the application—far more than would an entry on a "yet to be completed" list of functions. As your application starts to take shape, you and those testing the application can exchange ideas. And that communication will result in better solutions for implementing and integrating new features than you would

come up with on your own. In short, the design process becomes a collaboration between you and the user, which ensures that the resulting program has indeed been designed for the user.

The second big advantage to this approach is that, in an event-driven environment such as Windows, the user interface is the hub around which your application revolves. In the character-mode world, programmers could build applications that forced the user to conform to a tree-structured menu system and sequential procedures; the programmer dictated the flow of control. In contrast, in a Windows application the user dictates the flow of control, accessing different parts of the application at will by interacting with the user interface. So you need to have the user interface in place before you can complete the functional underpinnings.

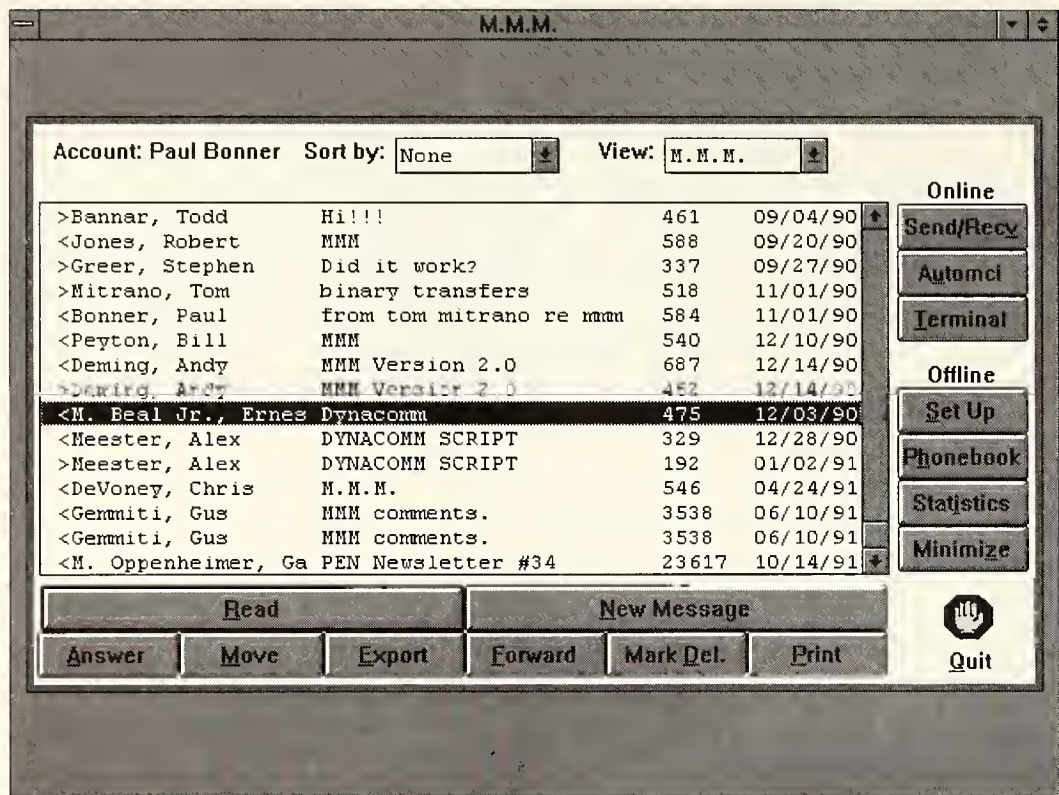
On-the-Job Training

The final major benefit of this approach to application development is that you needn't know how to make the application do everything that it eventually will do before you begin work on it.

Consider, for example, M.M.M., the MCI Mail Manager application described in Chapter 15. When I started work on the M.M.M. project, I knew that I wanted it to offer a broad range of message-handling facilities. For instance, I wanted to be able to create new messages, answer messages, print messages, delete messages, and forward messages. I wanted to be able to access an off-line address book listing the names and MCI account numbers of frequent correspondents. I wanted the application to be able to send and receive messages either automatically (at preset intervals) in the background while I worked with other programs, or instantly, at my command. I also wanted to be able to sort messages by a variety of methods (name, sender or addressee, and date) and to move messages among several different mail folders.

The problem was, I'd never even programmed a log-on routine in DynaComm's script language, let alone a full-fledged application. I knew from my perusals of the DynaComm reference manual that its script language was capable of doing everything I wanted it to, but I didn't know how to make it do anything. So I had a choice. I could study the manual and try to figure out everything I would need to know how to do in advance, or I could dig right in and start programming, implementing new functions as I figured out how. Being a big believer in on-the-job training, I opted for the second approach.

I started by exploring the commands in the DynaComm script language for creating and updating dialog boxes, which soon enabled me to put together M.M.M.'s primary user component, a dialog box that I dubbed the main Mailboxes screen (shown in Figure 8.1).

Figure 8.1M.M.M.'s main
Mailboxes screen

Next, after studying some sample log-on scripts that accompany the DynaComm package, I succeeded in getting the script to log onto MCI and send my user name and password at the appropriate moment. Doing that, I learned the value of the script language's WAIT STRING command, which pauses script execution until a specified string is received from the remote host. For instance, the command

```
WAIT STRING "user name:"
```

would pause script execution until MCI sent its "Please enter your user name:" prompt, and then resume execution with the next command,

```
SEND $Name
```

to send my user name.

Next, I applied that knowledge to the task of teaching the script to recognize the MCI prompt that tells you when there are messages waiting for you, and to respond to that prompt by downloading each message into a separate file. Of course, I needed a way to keep track of the files I was downloading, so I had to learn about DynaComm's database-like tables feature. That, in turn, provided me with the knowledge I needed to create the tables that

would be used to store the names and MCI addresses of people in the script's address book, and so on.

In putting together the address book, meanwhile, I figured out how to use DynaComm's TABLE SORT commands to keep the address book in alphabetical order, a lesson I later applied to the sorting functions on the main Mailboxes screen. Later I tackled the more complex tasks of creating outgoing messages and adding support for sending messages to people on other mail services through MCI's EMS (external mail service) function, and so on.

I took some missteps along the way, and several times I had to go back and rewrite routines when I realized that the methods I'd used initially weren't robust enough (the log-on procedure described above is a good example—I later expanded it to handle various error conditions). Still, I am convinced that tackling a real programming task from the start was the best way to learn the DynaComm script language—and that the same would be true for any other programming language. I doubt whether I would ever have undertaken the project if I had been forced to completely master all the techniques that I might conceivably have needed before starting. Moreover, the lessons I learned (and the mistakes I made) in building the early functions enabled me to make the functions I built later more robust, and suggested many possibilities that I might otherwise have overlooked.

Testing and Debugging

Simultaneously with this iterative program-development method, you should conduct an ongoing process of testing and debugging your application.

The first step in the testing process is to ensure that every aspect of the application works to your own satisfaction. Once you've decided that the application bug free, it is time to put it in the hands of other users, who are bound to uncover more problems in five minutes than you could find in five years of testing it yourself.

Developer Testing

The first rule of application testing is obvious: You have to test every new feature you add to your application to make sure it works as expected. So, for example, if clicking the button labeled "Exit" is supposed to end your application, the first test you've got to make is to ensure that it actually does so.

Expect the Unexpected

Next, you should try to trip up your application by subjecting it to unusual or unexpected conditions. Consider the Read button on M.M.M.'s main Mailboxes screen (Figure 8.1). When the user clicks this button, the script is supposed to react by loading the message under the selection bar in the central

list box into DynaComm's memo editor. But what happens if there are no messages in the current mail folder when the user clicks the Read button? (It makes no sense, of course, that the user would try to read a message when none exists, but rest assured that eventually someone will try to do just that.)

DynaComm reacts in an unfriendly manner to instructions to edit a non-existent file, posting a Task Error box and halting script execution—a rather harsh reaction to an innocuous user error. So to prevent that you have to anticipate the error and incorporate code that checks to make sure the cursor is pointing to an actual message before the script attempts to process the Read command.

Don't Assume It Ain't Broke

The third step in testing your application is to ensure that your fixes and new additions don't foul up functions that used to work. In an iterative program-development process you'll often find yourself tinkering with existing record structures or modifying existing variables to use them in new ways. But in the course of doing so, it is easy to overlook the fact that the little change you just made completely undermines an existing routine that expects the variable whose value you just modified to remain unmodified.

Because of this, you should perform a thorough retest of your previously tested code every time you incorporate a new routine into your program, or every time you modify an existing one.

Covering Every Base

As you proceed with your tests, there are some standard pitfalls you should look out for. These problems and traps have been known to trip up even the most accomplished Windows programmers, so it is imperative that you conduct these tests for every application you develop.

As you conduct these tests, don't just look for problems that crash your application or damage data. Those are the most serious problems, of course, but they're not the only ones you should fix before deciding that your application is ready for prime time. Look for situations that slow down your application, or make it less convenient to use than you anticipated. These are problems that will directly impact the usability of your application—and they should be solved before you decide the application is finished.

Mouse and Keyboard Operation

You should test every aspect of your application both with and without a mouse, asking yourself these questions:

- Is every feature accessible from the keyboard?
- Is that access as convenient as it should be?

- Is there a keyboard shortcut for every menu item and every dialog box control?

If the answer to any of these questions is no, you should go back and add better keyboard support to your application.

Standard and 386 Enhanced Modes

You should also test your complete application in both Standard mode and 386 Enhanced mode. (Fortunately, the removal of Real mode from Windows 3.1 eliminated the need to also test in that antiquated variant of Windows.)

As a Windows user, you might be under the impression that as far as Windows-based applications are concerned, Standard mode and 386 Enhanced mode are identical. However, as a Windows developer you'll soon realize the falseness of that assumption. For no obvious reason, things that work flawlessly in one mode won't work in the other. Mode-related problems are especially likely with (but not limited to) applications that interact with local area network software or with DOS-based software. To be safe, however, you should completely test every application you write in both modes.

Different Video Resolutions

If you always run Windows in a single video mode, it's easy to forget that other users of your application will be utilizing other screen resolutions. Windows applications are supposed to be device-independent with regard to screen resolution, but that independence doesn't always come automatically. The seven-point control labels that are perfectly readable in standard VGA mode might be completely illegible in 1024x768 8514/a mode, or the beautiful screen you've built by mixing bitmaps and standard user-interface controls in 800x600 SVGA mode might be scrambled in 640x350 EGA mode. And the striking color display you've built to use on your 256-color monitor might be absolutely uncradable on a notebook PC's 16-gray shade screen.

You should actually start thinking about the different screen resolutions under which your application may run during the application-design phase. For instance, if there is any possibility that your application will be used on an EGA-resolution screen, you shouldn't create a fixed-sized dialog box any larger than 640x350 pixels. (On a VGA screen the minimum size increases to 640x480.) Otherwise, the user won't be able to see the entire dialog box.

It might not be practical to test your application with every screen mode, but test it with as wide a variety of resolutions and color-depths as possible.

TrueType

You should also test your application with TrueType enabled and disabled, and with all but the standard fonts that accompany Windows 3.1 (Arial, Courier New, Times New Roman, Symbol, and WingDing) disabled. Otherwise, you might discover that, without your even being aware of it, your application

is relying on TrueType's presence, and that the text displays that look so good with TrueType's scalable fonts look terrible on a system in which TrueType has been disabled, or on which the specified font doesn't exist.

Similarly, if you're using Adobe Type Manager to provide scalable PostScript fonts for your screen display, test the application with ATM disabled.

Network Operation

If you developed your application on a system that is connected to a local area network, be sure to test it on a stand-alone machine. Or, if you've developed it on a non-networked PC, but expect that it will be used on a network, be sure to test it on the network—even if your application won't be interacting directly with the network.

Low Memory Conditions

If you are working on a fast 386 or 486 with lots of RAM, it is easy to forget that many users are still using less powerful machines with just 1 or 2 megabytes of RAM. On those machines, Windows is constantly straining to fit everything it can into memory. Low memory conditions present a much harsher environment for Windows applications, and thus you should use them to test the robustness of your application. What happens when you try to run your application when there is only 100k of free RAM? Does it die? Does it slow to a crawl? If so, then you should get back to work and try to isolate the problem.

Low System-Resource Conditions

System-resource memory is a block of RAM that Windows devotes to keeping track of windows, menus, icons, and every other distinct element of each application that is running. In Windows 3.0, it was easy to run low on system resources, a condition that led to many an application crash.

Windows 3.1 is, thankfully, far better at managing system resources than its predecessor—partly because it devotes twice the RAM (128k versus 64k) to the task. Nevertheless, it is not impossible to run low on system resources even in Windows 3.1.

Thus, it is important that you test your application in situations in which system-resource memory is very limited—such as those in which Program Manager's About box reports less than 20 percent free system resources. The easiest way to do so is to use the program STRESS.EXE from the Windows Software Developer's Kit to create a low system-resource situation. (STRESS.EXE is also useful for its ability to simulate other situations that can trip up an application, including low memory conditions and low disk-space conditions.)

If you don't have access to the SDK, however, you can create the same situation by opening up several applications that you know are big consumers of system-resource memory. Any application that sports lots of icons is a

good candidate here, but you can test an application's resource appetite by checking the memory-usage report on the Program Manager's About box before and after loading the application. Also, Program Manager itself is a big user of system resources, because every program group and every program item consumes a chunk of that memory. So another way to cause a precipitous drop in system-resource memory is to create several program groups, each containing dozens of program items.

Running in the Background

If your application engages in any background processes (processes that launch or execute automatically without user interaction), you should test it running in the background both as a window covered by another application's window and as a minimized icon. Your goal here is to ensure that your application runs correctly in both states and that it doesn't interfere with or cause a serious performance drain on other applications running in the foreground.

Different Windows Shells

Dozens of commercial Windows shells are available. (These applications replace Program Manager and shut down Windows when they are shut down.) In addition, undoubtedly hundreds more custom shells are in use at various corporate sites. So it is unrealistic to expect that you can test your application with each of them. However, you should ensure that it isn't dependent on Program Manager (or Norton Desktop for Windows or whatever other shell you use) by evaluating its performance with at least a couple of other shell programs.

Different Drive/Directory Combinations

Be sure to test the ability of your application to run from a different drive and a different path than those in which you developed it. Your application should not be dependent on running from a specific drive or directory, and shouldn't include any drive- or directory-specific references. Otherwise, you might find that the application that always runs flawlessly from your C:\DEV\BIN\MYAPP directory doesn't work at all from someone else's E:\UTILS directory.

Nevertheless, it is often convenient (if not especially wise) to include directory-specific references early on in the development process, before all the routines necessary for the application to automatically locate its directory are in place. Unfortunately, it is easy to forget to remove these references later—a problem you won't be aware of until you get the first angry user complaint. The best way to ensure that these references don't exist in your finished application is to run your application from a different drive and a different directory than the one in which you wrote it.

When doing this testing, be sure to *move* your program files to the new directory for testing (after backing them up), rather than simply copying them there. Otherwise, your application might *appear* to pass the drive/directory independence test, when it is actually still using the original files in the original subdirectory.

User Testing

There are two reasons to give your application to the people who will actually end up using it as early as possible in the development process. The first is that users will uncover bugs you never dreamed of. They'll try to open nonexistent files, and click buttons when there no earthly reason for doing so, and try to merge incompatible data, and otherwise test your application's every stitch and seam (commercial-application developers call this *beta testing*).

The other reason is that doing so might produce an application that is not only more bug-free than you could produce alone, but also richer and more valuable. The key to collecting this double benefit is to piggyback a little usability testing onto the bug-catching and function-testing process.

In practice, you'll want to be selective about which users you give the application to for testing. You don't want testers who are going to be so discouraged over bugs in early versions that they get turned off to the application as a whole. So you should select users who are enthusiastic about the project and who you consider hardy enough to survive a few crashes with their good humor intact.

Then watch them work with your application. Look to see where they hesitate and where they are confused, and ask them why. What did they expect the application to do at that point? What would make more sense—what would be a better way to have it work? The user won't always know how to fix the problem, but if nothing else he or she has helped you to identify a problem you might never have discovered on your own.

Next, find out the user's general impressions of the application. What does he or she like about it? What should be changed? What other things should it do?

You should try to start this usability testing process as early as possible in the development cycle. Of course, before you can do this kind of testing you have to have an application that is at least partially functional, but there is no point in waiting until the application is complete before you start. Doing it early allows you to incorporate both users' suggestions and any resulting ideas of your own into the final application. The end result is an application that pleases everyone involved.

Documentation

Documentation is a necessary evil. How many times have you heard people say, “I never read documentation”? They might even believe it, but just try giving them an application without any documentation. Can you hear them screaming?

The fact is that even the most simple and intuitive application requires at least a modicum of documentation. For your homegrown applications, there is no need to produce the 20-pound monsters that major software companies seem to specialize in, but if anyone else but yourself is going to be using the application, you should provide documentation for at least the following three general areas.

Installation

The documentation should specify the files that have to be installed to use your application. Do they need to be on the DOS path? What changes, if any, does your application make to WIN.INI, AUTOEXEC.BAT, or CONFIG.SYS? What files, if any, does your application install in the WINDOWS directory?

Basic Operations

You should provide an overview of the general purpose of the application and detailed instructions on the use of each application screen and dialog box, including explanations of every menu item, button, list box, and other control.

The documentation should also briefly discuss any assumptions you’re making about the skill level or knowledge the user has coming into the application. If, for example, you’ve built an application using Excel’s macro language that expects the user to have a basic knowledge of spreadsheet operations, your documentation should at least tell less experienced users what chapters in the Excel documentation they should review in order to acquire that knowledge. (If you want to get more ambitious, you could include a tutorial on spreadsheet basics in your documentation.)

Troubleshooting

The documentation should also identify the most likely areas in which problems might arise during operation, and suggest solutions to each. These are going to vary considerably from application to application. In a communications application, for instance, they might include modem- or terminal-settings problems, or incorrect network sign-on data, whereas in a WordBASIC form-letter application they might include incompatible data-file types or missing files.

On-Line Help

In a Windows application, on-line help can be even more important than written documentation, so it should be a part of any application that is destined for widespread use.

Fortunately, Windows 3.1 provides a superb Help system, which is available to any Windows application. Unfortunately, developing HLP-format files for use with the Windows Help engine can be a nightmarish proposition. To begin with, you need a copy of Word for Windows or another Windows application capable of producing RTF (Rich Text Format) files. Then you need a copy of the Windows Help compiler, which turns RTF into HLP files. The Help compiler is part of the Windows SDK, and is also included with Turbo Pascal for Windows, the Visual BASIC Professional Toolkit, and many other Windows development tools. Finally, you need a lot of time and patience to build the complex Word for Windows RTF file from which the Help compiler will build the HLP file.

However, there is a better way to create both written and on-line documentation for your applications. Doc-To-Help, from Wextech Systems, will take plain text documents in Word for Windows and simultaneously create formatted manuals and Windows Help files. So you need only create one file in Word for Windows, and Doc-To-Help will do the rest—generating both a standard Windows on-line help system and an indexed and cross-referenced paper manual.

Doc-To-Help is available for \$249 from Wextech Systems, Inc., 60 East 42nd Street, Suite 1733, New York, NY, 10165; (212) 949-9595.

Shrink-Wrap Time

Once the ink dries on the documentation, it's time to break out the shrink-wrap machine and pronounce your application ready to go. You've gone through a long process of defining the application, selecting a development tool, designing a user interface, coding the functional underpinnings, building prototypes, testing and debugging, and finally documenting the application. And, to tell the truth, that's probably not the easiest way to develop an application. You can almost certainly produce a result—of some kind—faster if you just go into a dark room by yourself for a while. However, though you will emerge with a “finished” product that way, it won't be as strong or as usable as one in which you've involved the user from the beginning.

Enough theory—let's party. This concludes the second section of the book. The chapters in the final section present hands-on examples that put the principles described here to work in real programming projects, with step-by-step explanations of the decisions made at every stage of the process. Bon appétit.

Customizing Applications—The Ultimate Notepad

Presenting Data—Who's Who at PC/Computing?

Automating Existing Applications—AutoPrint for Windows

Making Use of Libraries—Recycler

Linking Applications through DDE—Windows Broker

Enhancing Applications—DocMan

Communicating with Host Systems—M. M. M.: the MCI Mail Manager

P A R T

3

The Projects

C H A P T E R

9

Customizing Applications—The Ultimate Notepad

The Birth of a Notion

*Exploring
NOTEPAD.WDF*

*Exploring
NOTEPAD.WBT*

*Wrapping Up The
Ultimate Notepad*

LET'S START OFF THE PROJECTS SECTION OF THE BOOK WITH AN EASY one: an example of how Windows batch languages can be used to customize and enhance other Windows applications. For this project, dubbed The Ultimate Notepad, I used Wilson WindowWare's WinBatch to add a series of new functions to the Windows 3.1 Notepad accessory.

The Birth of a Notion

The origin of this project was simple. I wanted to make Notepad more useful.

It's awfully convenient having a small, fast text editor always available, the way the Notepad accessory is in Windows. The Windows 3.1 version of Notepad can edit any ASCII file up to about 50k in length, and because it is identified as the default editor for TXT and INI files, it is automatically launched whenever you double-click on a file with either of those extensions in File Manager.

Unfortunately, convenience is about all Notepad has going for it. It is missing too many features that are *de rigueur* for a full-featured text processor. It doesn't allow you to merge existing files into the current file, to edit more than one file at a time, or to save a selected part of the current file to a new file. It offers a search capability, but not a replace function. It can't count the number of words in a file, and it refuses to automatically indent a new line of text to the same point as the previous line. It doesn't even provide shortcuts for converting a selected block of text to upper- or lowercase. All of which combine to make Notepad a highly disappointing text processor.

There are, of course, other better-endowed text editors available for Windows if you've given up on Notepad, but they're not perfect either. Few of them are as quick to load or as fast at basic operations as Notepad is—and you have to pay for them (Notepad is free).

It is also possible to edit plain ASCII files in Windows Write or another full-featured word processing application, but you have to go through a file import routine (albeit a brief one with Write), and you're dealing with a program much less limber than Notepad. So a word processor really doesn't do the trick either.

Besides, I like Notepad. Everything it does, it does very well. It simply doesn't do enough. So I decided to try and change that.

Selecting the Development Tool

This project was somewhat atypical of most program development efforts, in that the capabilities of the development tool provided much of the original impetus for the project. In most cases, a development project is need-driven.

and the selection of a development tool is a matter of finding the tool that best suits the need.

In this case, however, things were turned upside down. Although I certainly needed a better text editor than the standard Notepad, I wasn't harboring any hope that I could fashion an improved model out of Notepad itself. As far as I knew then, Windows applications that didn't include a macro language were not customizable. So I made do, sometimes working with Notepad and grumbling about its inadequacies, and sometimes using other text editors.

But then I was introduced to WinBatch, which offers the ability to customize other Windows applications in two ways: by providing hotkey access to application-specific macros written in the batch language, and by allowing you to attach new menu items for those macros to the application's Control menu.

I was looking around for a way to put those capabilities to the test when it occurred to me that perhaps here was a solution to Notepad's inadequacies. It hardly seemed a fair test, though. Batch languages are best for knitting together the advanced capabilities of other applications, and Notepad is so limited that I doubted there was enough material to work with. Still, if there was a way to enhance Notepad, WinBatch was it, so I decided to give it a try.

Setting Objectives

Having decided that this project was a good way to test the limits of what is possible with a batch language, I outlined a tough set of objectives. I wanted to bring Notepad up to par by adding to it all the basic text-editor features it lacked:

- A customizable mask for the File Open dialog box
- The ability to edit more than one file at once
- A merge file facility
- A command to save selected text
- A Replace command to accompany Notepad's search facility
- Automatic indentation
- The ability to convert selected text to upper- or lowercase
- Word and character counting

In truth, I thought this list was pretty unrealistic. I didn't really expect the WinBatch-Notepad combination to prove capable of all those functions. In fact, I probably would have been satisfied if I had succeeded in producing even half of them. Still, I thought the project was worth the effort—if only to see which ones were possible.

To my surprise, the WinBatch-Notepad duo went eight for eight at the plate. After only a few days' effort, I was able to add working versions of each of these functions to Notepad. Some of them—most notably the word count function—are a little slower than I would like, but not so slow as to render them unusable.

Let's look at how The Ultimate Notepad was built.

Exploring NOTEPAD.WDF

The thing that makes WinBatch so useful for customizing other applications is that you can create a series of batch programs or macros specific to the other application, and then identify them as such to WinBatch using a special definition file. This file contains the name of the menu item to be added to the Control menu of the application, a shortcut key for that item, and the name of the macro to be executed when that menu item or shortcut key is selected.

WinBatch actually consists of two executable files: WINMACRO.EXE and WINBATC.H.EXE. WINMACRO.EXE has the task of adding the items to the Control menus of other applications. It also monitors those applications to determine if one of the custom menu items or keyboard shortcuts has been selected. If so, it launches WINBATC.H.EXE and passes it the name of the selected macro. WINBATC.H.EXE is the WinBatch language interpreter, which carries out the macro's instructions.

When you launch WinMacro it simply appears as an icon at the bottom of your screen (shown below) and waits for you to launch another application. When you do so, it checks to see whether a custom menu-definition file exists for that application. If so, WinMacro modifies the application's Control menu by adding to it the items specified in the definition file.



The definition files used by WinMacro have the same name as the executable file they are associated with, but they have the extension .WDF. Thus, the WinMacro definition file for NOTEPAD.EXE is NOTEPAD.WDF.

To effect the modifications that I wanted to make to Notepad, I created the following NOTEPAD.WDF file:

```
;NOTEPAD.WDF
Open File...          \^O:WINBATC.H NOTEPAD.WBT OpenerSub
Open Another...      \^N:WINBATC.H NOTEPAD.WBT OpentwoSub
Insert File...       \^I:WINBATC.H NOTEPAD.WBT MergeSub
```

```

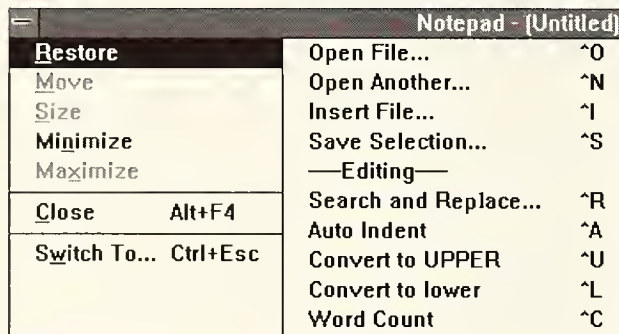
Save Selection...      \^S:WINBATCH NOTEPAD.WBT Savese1Sub
-----Editing-----      :WINBATCH NOTEPAD.WBT DummySub
Search and Replace    \^R:WINBATCH NOTEPAD.WBT ReplaceSub
Auto Indent           \^A:WINBATCH NOTEPAD.WBT AutoIndentSub
Convert to UPPER      \^U:WINBATCH NOTEPAD.WBT CupperSub
Convert to lower      \^L:WINBATCH NOTEPAD.WBT ClowerSub
Word Count            \^C:WINBATCH NOTEPAD.WBT FastcountSub

```

This file produces the menu shown in Figure 9.1.

Figure 9.1

The customized
Control menu for
Notepad



The first line in the file is a comment (as are all program lines preceded by a semicolon in the WinBatch language) that identifies the name of the file. The remaining ten lines specify the custom items that WINMACRO.EXE is to add to Notepad's Control menu.

The first part of each menu item definition contains the text of the new menu item. For instance, the first new menu item will read "Open File...". The backslash that follows this text is used to separate the menu item text from the hotkey for this item (which also appears on the menu). In the case of the first menu item, the hotkey is listed as ^O, which means Ctrl-O. (The caret, ^, is used to represent the Ctrl key throughout the WinBatch language. Similarly, the plus sign (+) represents the Shift key, and the exclamation point (!) represents the Alt key).

Finally, the text following the colon after the shortcut key identifies the application that should be launched when the menu item is selected and the parameters that are to be passed to that application. In the case of the first menu item, the application to be launched is WINBATCH, the first parameter is NOTEPAD.WBT (the name of the file that contains all the custom macros for this project) and the second parameter is OpenerSub, the name of the subroutine in that file that is associated with the first menu item.

The only menu item that doesn't follow this pattern is the one for the separator bar, which is labeled "-----Editing-----". There is no hotkey for this item because the user is not supposed to select it—its purpose is merely to

separate the group of menu items. However, WinMacro does require you to specify an action for this item, so I instructed it to perform the subroutine labeled “DummySub” in the NOTEPAD.WBT file, which consists of a single command, EXIT, that terminates the batch program. In other words, the macro launches and exits immediately, without carrying out any actions.

Next, let’s look at the NOTEPAD.WBT file itself.

Exploring NOTEPAD.WBT

NOTEPAD.WBT is a single batch program that contains ten individual macros, corresponding to the ten menu items I added to Notepad’s Control menu. These appear in NOTEPAD.WBT as individual subroutines.

Introductory Lines

NOTEPAD.WBT begins with a comment line containing the name of the file, followed by eight lines of code that are executed no matter which menu item is selected from the customized Control menu in Notepad.

```
;NOTEPAD.WBT
HomeDir=DIRHOME()
DIRCHANGE(HomeDir)
EnterString=NUM2CHAR(13)
TabString=NUM2CHAR(9)
CRLFString=STRCAT(EnterString,NUM2CHAR(10))
Null=""
OldClip=CLIPGET()
GOTO %Param1%
```

These lines establish a series of variables and directory pointers that are used by most of the macros in NOTEPAD.WBT. First, the DIRHOME() function is used to set the variable HomeDir to point to the directory that contains WinBatch’s executable files. Then the next line, DIRCHANGE(HomeDir) makes HomeDir the current directory. If you’ve installed WINBATCH.EXE and WINMACRO.EXE in your WINDOWS directory alongside NOTEPAD.EXE, this command ensures that any File Open or File Save dialog boxes used by the macro will point to that directory.

The next two lines use the WinBatch NUM2CHAR function to create variables containing special key sequences, such as a horizontal tab character (TabString=NUM2CHAR(9)), and a carriage return character (EnterString=NUM2CHAR(13)). Then the next line combines NUM2CHAR with WinBatch’s STRCAT command (which joins two strings) to create a variable named CRLFString that contains the carriage return–line feed combination.

Finally, the introductory lines assign an empty string to the variable called Null, and assign the current contents of the Clipboard to the variable OldClip. This is here because many of the routines in NOTEPAD.WBT change the contents of the Clipboard. Storing the Clipboard's original contents here in the variable OldClip allows the macro to restore the Clipboard before exiting.

The Subroutine Macros

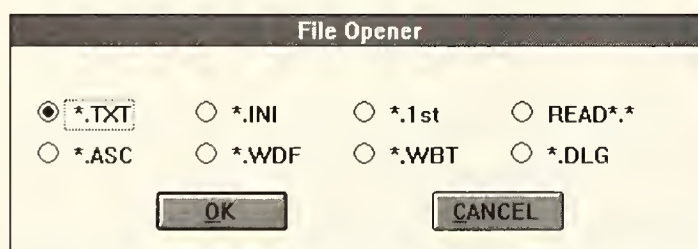
The WinBatch GOTO command is used next, to jump to the subroutine specified on NOTEPAD.WBT's command line (OpenerSub in the case of the first menu item, FastcountSub in the case of the last item, and so on).

The OpenerSub Routine

OpenerSub is used to supply additional choices for the mask that Notepad uses to display files in its File Open dialog box. When you select the File Open command in Notepad, you're normally presented with a list of all the text files in the current directory (*.TXT), and you can use the File Type combo box to select instead a list of all files in the directory (*.*). But what if you just want to select from all the INI files in the directory, or all the README files? Normally, you'd have to type the file specification you wanted into the File Name edit box. But with The Ultimate Notepad, you can pick the file mask you want to use from a customizable list of eight, as shown in Figure 9.2.

Figure 9.2

The File Opener dialog box



You can access this dialog box by selecting the Open File... item on Notepad's Control menu, or by pressing the hotkey Ctrl-O. Then, once you've made your selection, the macro opens Notepad's File Open dialog box and automatically types the file mask that you've specified into the File Name edit field.

The code for this routine looks like this:

```
:OpenerSub
Select1 = "*.TXT"
Select2="*.INI"
Select3="*.1st"
Select4="READ*.*"
Select5="*.ASC"
Select6="*.WDF"
Select7="*.WBT"
Select8="*.DLG"
IF FILEEXIST(STRCAT(HOMEDIR,"MASK.DLG"))== @FALSE THEN
GOTO OpenerWrite
:OpenerDraw
DIALOGBOX("File Opener", "MASK.DLG")
FileString=Select%A%
SENDKEY("!FO")
C=WINGETACTIVE()
IF (C=="Notepad") THEN GOTO Oops
:PutString
CLIPPUT(FileString)
SENDKEY("+{INSERT}{ENTER}")
GOTO Ender
:Oops
W=WINGETACTIVE()
IF W=="Open" THEN GOTO PutString
IF W=="Notepad" THEN GOTO Oops
GOTO Ender
```

The OpenerSub code starts, like every routine in this file, with a label, which is indicated by a single word preceded by a colon. :OpenerSub in this case. This identifies the routine to WinBatch, allowing you to jump to it with a GOTO command.

The next eight lines identify the eight file masks the dialog box will offer and assign them to variables Select1 through Select8. Then the next line checks to see whether a file called MASK.DLG exists in the directory pointed to by the variable HomeDir.

To create a custom dialog box in WinBatch, you store instructions for drawing the box in a special template file. NOTEPAD.WBT creates these template files automatically the first time they are needed. So if MASK.DLG is not found in HomeDir, the macro jumps to the OpenerWrite routine, which draws the box and then jumps back to the next line in the program, labeled "OpenerDraw." (OpenerWrite and two other dialog box-creation routines are discussed at the end of this chapter.)

Note that if, after having run this routine once, you modify the list of file masks it offers, you'll need to delete MASK.DLG from your WINDOWS directory in order to force the macro to recreate the dialog box using the new file masks the next time it is run.

WinBatch's program flow-control commands are limited to a GOTO statement and an IF-THEN statement, so it doesn't support subroutines, per se. However, you can create a section of code that will act like a subroutine by using a GOTO command to jump to the code segment's label, and then at the end of the routine jump back to another label on the line following the original GOTO command, as is done here with the OpenerDraw label.

The next line in the routine uses the DIALOGBOX command to draw the dialog box described in MASK.DLG and title it "File Opener". This command pauses script execution until the user makes a selection from the dialog box. The choice made by the user will be assigned to the variable A, since that is the variable used to define the selection in the dialog box template. So the next line takes advantage of WinBatch's variable-substitution command to assign the contents of the selected file mask to the variable FileString. (In WinBatch, anytime you place the name of a variable between percentage signs, the contents of that variable are inserted into its place at runtime. So if A=5, then Select%A% will be interpreted as Select5.)

Next the macro uses the SENDKEY command to send the standard File Open command (Alt-FO) to Notepad, and then the WINGETACTIVE() command to assign the title of the active window to the variable C. If the File Open dialog box is the active window, C will contain "Open". However, if the text in the current file has not yet been saved, Notepad will have posted a message box titled "Notepad", which informs you of that and offers you the chance either to cancel the File Open operation or to proceed with or without saving your text. So if the title of the current window is "Notepad", the macro knows that the File Open dialog box is not active, and jumps to the Oops subroutine to await the user's response to the error message from Notepad.

Meanwhile, if the File Open box is active, execution continues at the line following the label :PutString. The macro places the file mask to be used onto the Clipboard using the CLIPPUT command, and then sends the keystrokes Shift-Ins Enter to Notepad's File Open dialog box. These commands paste the contents of the Clipboard (the selected file mask) into the File Name edit box, and force Notepad to update its file listing to display only files matching the specified file mask, just as if you had typed the mask yourself and then pressed the Enter key.

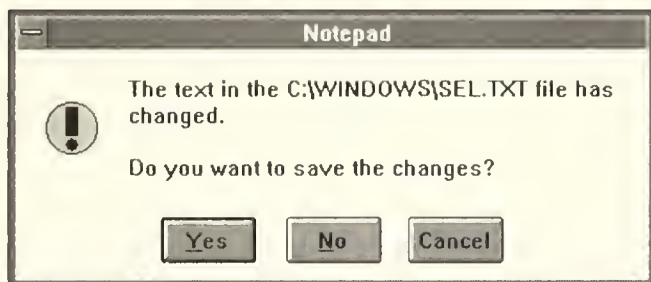
Having completed its task, the macro jumps to a routine called Ender, which is used to terminate each of the routines in NOTEPAD.WBT. Ender restores the original contents of the Clipboard before exiting.

The remaining commands in OpenerSub—those that follow the :Oops label—are performed only if Notepad responded to the original !FO

command with a message box warning that the current file has not been saved, as shown in Figure 9.3. The Oops subroutine is designed to pause script execution until you indicate whether or not you want to save the current file, or until you select the Cancel button. So it keeps checking the title of the active window. If the title is “Open”, then you have evidently already signaled your intention to either save or not save the file, so control jumps back to the PutString routine and the script picks up where it left off.

Figure 9.3

Notepad's warning message that the current file has not been saved



Meanwhile, if the current window title is “Notepad”, then you haven’t responded to the dialog box yet, so it jumps back to the beginning of the Oops routine and checks the title again. If neither of those conditions is true, then you must have selected the Cancel button (in which case the title would be “Notepad -- *filename*”, where *filename* is the name of the current file).

That concludes the OpenerSub routine.

The OpenTwoSub Routine

The next target on my list of goals for improving Notepad was the ability to open more than one file at a time and copy text back and forth between files. Unfortunately, Notepad is strictly limited to working with a single file at a time, and no amount of clever batch file coding is going to change that. Nevertheless, I achieved a degree of success with a simple seven-line subroutine that takes advantage of the fact that you can run multiple copies of Notepad at once to create a close approximation of a two-windowed Notepad. This routine simply opens a second instance of Notepad and stacks the two Notepad windows so that they fill the screen (as shown in Figure 9.4), allowing easy copying of data back and forth between files.

The code for OpenTwoSub looks like this:

```
:OpenTwoSub
ThisWin=WINGETACTIVE()
WINSHOW(ThisWin)
WINPLACE(0,0,1000,500,ThisWin)
RUN("NOTEPAD.EXE",Null)
```

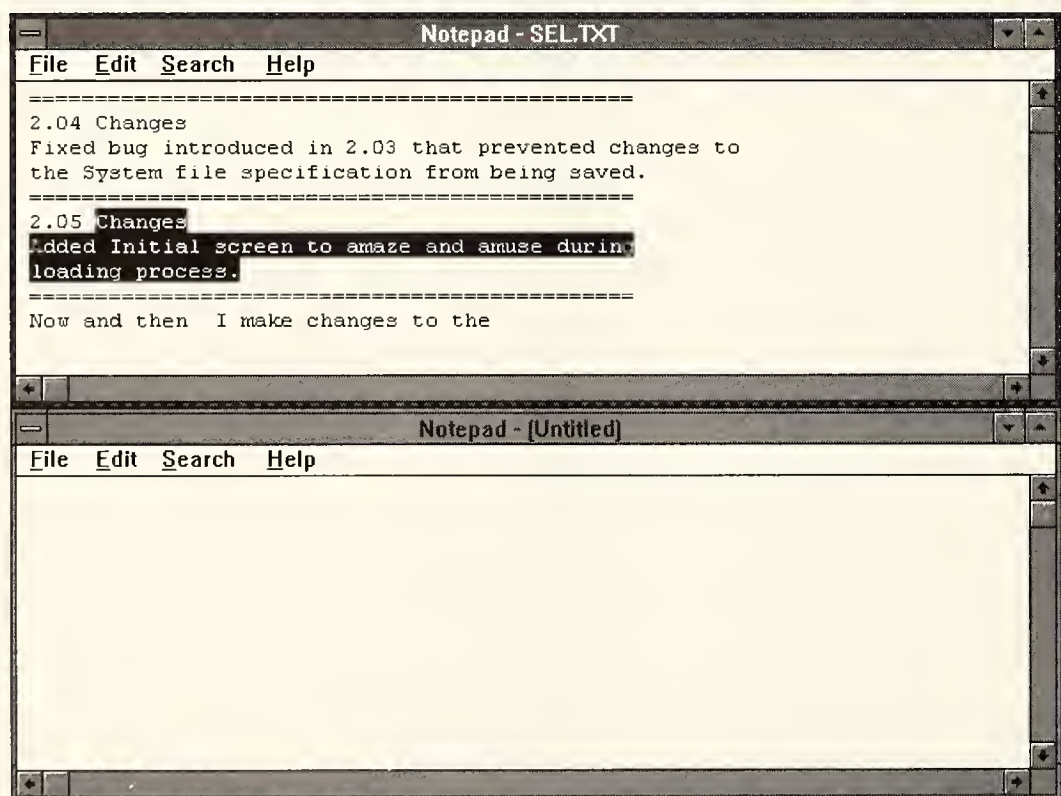
```

ThatWin=WINGETACTIVE()
WINPLACE(0,500,1000,1000,ThatWin)
GOTO Ender

```

The first command in `OpenTwoSub` uses `WINGETACTIVE()` to obtain the title of the active Notepad window. Next, the `WINSHOW` command is applied to the current Notepad window. I used `WINSHOW` here because it automatically restores maximized windows to their normal size, allowing the next command, `WINPLACE`, to change the window's size and location. Otherwise, if the window is in a maximized state, `WINPLACE` will have no effect on it. Then the macro issues the `WINPLACE` command to resize the current window so that it starts at the top-left corner of the screen and extends all the way across the screen and halfway down it.

Figure 9.4
`OpenTwoSub`
 stacks a pair of
 Notepad windows



The `WINPLACE` command views the screen as composed of 1,000 horizontal units and 1,000 vertical units, and accepts two pairs of coordinates: one for the upper-left corner and the other for the lower-right corner (in X,Y order). So the command `WINPLACE 0, 0, 1000, 500`, tells WinBatch to arrange the window starting at the upper-left corner of the screen (point 0, 0) and extending all the way across (1,000 units) and halfway down (500 units).

Next the script runs a second copy of Notepad (passing it a Null parameter instead of a file name), uses `WINGETACTIVE()` to obtain its title, and then `WINPLACE` to size it so that it fills the bottom half of the screen. Finally, the macro jumps to the Ender routine.

The MergeSub Routine

The next routine, MergeSub, provides the ability to merge other text files into the current file. It is called when you select the Insert File... item from Notepad's Control menu.

My experience with this routine illustrates one of the biggest time-wasting traps you can encounter as a programmer. I had decided that the way to implement this routine was to use WinBatch's `FILEREAD` command to read the insertion file line by line, copying each line to the Clipboard and then pasting it into the file. It was such an obvious solution that I never looked for another one—even though no matter how much I tweaked it, the routine was too slow to be useful. The fastest version I could come up with still took nearly two seconds per kilobyte of data in the insertion file. So it would take over 36 seconds to insert a 20k file into the current Notepad document—pretty awful performance in my opinion.

Nevertheless, it seemed so evident that this was the correct method that I wasted countless hours trying to squeeze a little more performance out of it, rather than looking around for a new method. In fact, I got to the point of describing that method for this chapter when it suddenly hit me, "That's dumb. Why don't I just run another copy of Notepad, load the file there and copy it to the Clipboard, then shut down the new Notepad and paste the Clipboard contents into the original Notepad?"

Bingo. The new method works like a dream. It will insert a file of any size into your current Notepad document in about four seconds flat. (Of course, the total size of your current file and the file being inserted cannot exceed Notepad's 50k limit.)

The moral of this story is that sometimes, after having spent hour upon hour trying to patch and tweak a section of code to make it do what you want it to, the best thing you can do is tear it up and start off fresh with a blank sheet of paper. If the obvious solution doesn't work, don't kill yourself trying to make it work. Instead, look for another "obvious" solution.

The new and improved code for MergeSub looks like this:

```
:MergeSub
A = "*.TXT"
IF FILEEXIST(STRCAT(HOMEDIR,"INSERT.DLG"))== @FALSE THEN
  GOTO InsertWrite
:InsertDraw
DIALOGBOX("Insert File","Insert.Dlg")
EXCLUSIVE(@ON)
```

```

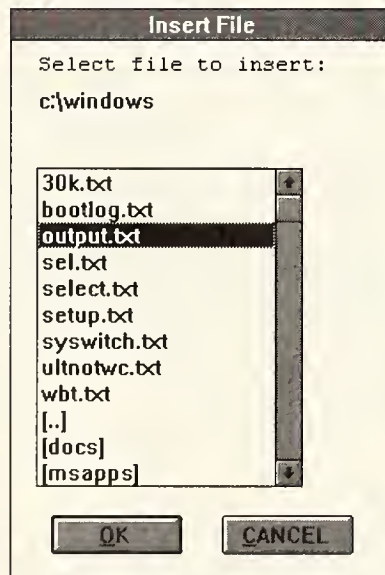
IGNOREINPUT(@True)
RUN("NOTEPAD.EXE",A)
SENDKEY("!EA")
SENDKEY("^{|INSERT}")
SENDKEY("!FX")
SENDKEY("+{|INSERT}")
IGNOREINPUT(@False)
EXCLUSIVE(@OFF)
GOTO Ender

```

The routine begins by assigning the value *.TXT to the variable A, which will be used to set the file mask used by the dialog box from which you'll select the file you want to insert. Next it determines whether a template file for that dialog box (INSERT.DLG) already exists, and if not, it jumps to the InsertWrite routine to create it. It then draws that dialog box (shown in Figure 9.5), and from it obtains the name of the insertion file.

Figure 9.5

The Insert File dialog box



WinBatch automatically creates the OK and CANCEL buttons on the dialog box, and cancels execution of the script if you select the CANCEL button, so there is no need to write code to process the result of the dialog box. Instead, the variable A is automatically set to whatever file was selected in the dialog box's list box.

Next, the macro sets two switches that affect the operation of WinBatch: EXCLUSIVE and IGNOREINPUT. By turning EXCLUSIVE on, the macro instructs WinBatch not to yield control to other Windows applications. This maximizes the performance of the macro, but can cause problems

with background operations, particularly network or communications tasks. (So you may want to remove this line if the macro appears to interfere with background tasks on your system.)

The second switch, IGNOREINPUT(@TRUE), tells WinBatch to ignore keyboard and mouse input until it encounters an IGNOREINPUT(@FALSE) command. This command is necessary in the macro because WinBatch pastes the contents of the insertion file at whatever insertion point is active. So if, in the middle of this macro's operation, you moved the cursor in Notepad or activated another application, the inserted text wouldn't end up where you wanted it. IGNOREINPUT prevents this by freezing the cursor until the macro is done.

You should take extreme care in how you use the IGNOREINPUT switch. Once it has been activated, there is no way to interrupt the macro, so if you have made a coding mistake that sends your macro into an endless loop, there is no way to stop it except by rebooting your PC. The moral is that you shouldn't use this switch until you are certain that the rest of your code is bulletproof.

The next line of the routine executes a new copy of Notepad, passing it the name of the insertion file as a command-line parameter (thus automatically loading the insertion file into the new copy of Notepad). Then it sends the new copy of Notepad the command Alt-EA (Edit, Select All) to select the entire text of the insertion file, and copies it to the Clipboard by sending the Ctrl-Ins command. Then it quits the new copy of Notepad. This causes Windows to reactivate the last active program, which was the original copy of Notepad, to which the macro sends the Shift-Ins (Paste) command, pasting the contents of the insertion file into the current Notepad file.

The macro concludes by turning off the IGNOREINPUT and EXCLUSIVE switches, and then jumping to the Ender routine. All in all it is a simple and fast solution to the problem of inserting a file in Notepad, and it illustrates an important point—in a multitasking environment such as Windows, there is no need to do everything in program code. Let another program (or, in this case, another copy of the same program) do the work for you, if you can.

The SaveselSub Routine

The next routine, SaveselSub, is used to save a block of text from the current file to disk as a separate file. To use it, you would simply highlight the text you want to save, and then press Ctrl-S or pick the Save Selection item from the Notepad Control menu.

```
:SaveselSub
Continue=1
CLIPPUT(Null)
Default=STRCAT(DIRGET(),"SELECT.TXT")
SENDKEY("^IINSERT")
```



```

a=CLIPGET()
IF a==Null THEN GOTO Empty
:CheckFile
Default=ASKLINE("Save Selection","Save to filename:",Default)
Exists=FILEEXIST(Default)
IF Exists==1 THEN Continue = ASKYESNO("File Already Exists","%Default% already
exists. Okay to overwrite?")
IF Continue==0 THEN GOTO CheckFile
hFP=FILEOPEN(Default,"Write")
FILEWRITE(hFP,a)
FILECLOSE(hFP)
GOTO Ender
:Empty
MESSAGE("Save Selection","You must select text first!")
GOTO Ender

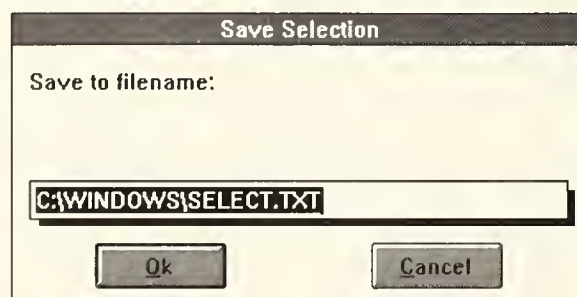
```

The `SaveselSub` routine begins by initializing the variable `Continue` with a value of 1. Then it clears the Clipboard and initializes the variable `Default`, used as a default file name for the new file, by concatenating the name of the current directory and `SELECT.TXT` using the `STRCAT` command. Next it sends a `Ctrl-Ins` command to copy the current selection to the Clipboard, and then checks to see if the Clipboard is empty. If so, it jumps to the `Empty` subroutine.

Otherwise, it uses the `ASKLINE` command, which prompts the user for a single line of input, to determine a name for the file in which the selected text should be stored, suggesting the file name contained in the variable `Default` as a default possibility, and assigning the user's response back to the variable `Default`. The `ASKLINE` prompt is shown in Figure 9.6.

Figure 9.6

The prompt used to obtain a file name for the selected text

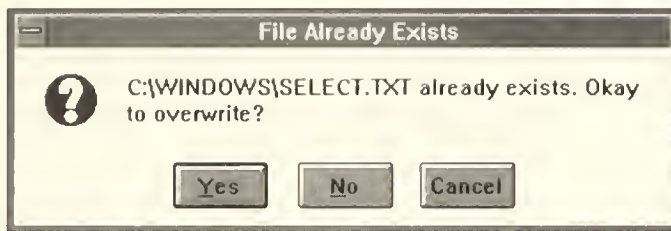


Next, the routine determines whether the file specified by `Default` already exists. If so, it warns the user and asks whether it should overwrite the existing file, as shown in Figure 9.7.

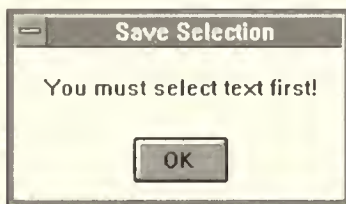
If the user indicates that the file should not be overwritten, the routine jumps back to the `Checkfile` subroutine to elicit a new name for the file. Otherwise, it opens the file specified by `Default` for writing, writes the contents of the Clipboard to that file, and then closes it and jumps to the `Ender` routine.

Figure 9.7

The "File Already Exists" warning



The Empty routine, meanwhile, is called only if there was no text selected in the current file when the Save Selection command was issued. It simply posts a message to that effect (shown below) and then jumps to the Ender routine.



The DummySub Routine

DummySub is called if the user selects the separator bar in the custom Control menu. It simply issues the Exit command.

```
: DummySub
EXIT
```

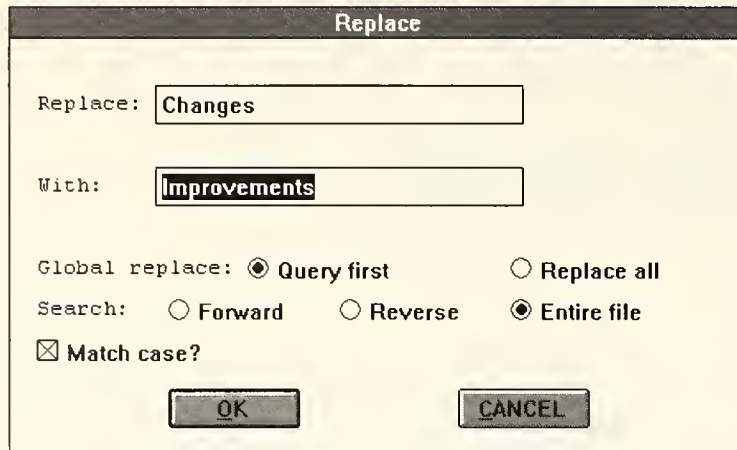
The ReplaceSub Routine

ReplaceSub adds a replace function to Notepad, allowing you to search for a specified character, word, or phrase and replace it with whatever text you specify.

The centerpiece of the ReplaceSub routine is the rather complex dialog box shown in Figure 9.8. The template for the Replace dialog box specifies that the variable A represents the search string, B represents the replacement string, C indicates whether the macro should query the user each time the search string is found (C=1) or automatically replace all instances of the search string (C=2), D indicates the direction of the search (forward from the insertion point if D=1, backwards if D=2, or throughout the entire file if D=3), and E indicates whether the search operation should be case-sensitive (E=1) or not (E=0).

Figure 9.8

The Replace dialog box



The code for the ReplaceSub routine begins by initializing the values of variables A through E, and then determines whether the dialog box REPLACE.DLG already exists. If not, it jumps to the ReplaceWrite subroutine. Otherwise, it draws the dialog box on screen and continues as shown here:

```
:ReplaceSub
a=NULL
b=NULL
c=1
d=1
e=0
IF FILEEXIST(STRCAT(HOMEDIR,"REPLACE.DLG"))== @FALSE THEN GOTO ReplaceWrite
:ReplaceDraw
DIALOGBOX("Replace","REPLACE.DLG")
IF A==NULL THEN EXIT
IF B==NULL THEN EXIT
CLIPPUT(A)
IF D==3 THEN SENDKEY("^{|HOME|}")
SENDKEY("{|sf+{|INSERT|}")
IF D==1 THEN SENDKEY("{|D|}")
IF D==2 THEN SENDKEY("{|U|}")
IF D==3 THEN SENDKEY("{|D|}")
SENDKEY("~")
B=STRTRIM(B)
CLIPPUT(B)
GOTO CheckFind
:FindNext
SENDKEY("{|f3|}")
:CheckFind
DaWin=WINGETACTIVE()
IF DaWin=="Notepad" THEN GOTO Notfound
IF DaWin=="Find" THEN SENDKEY("{|ESC|}")
WinActivate("Notepad")
IF E==1 THEN GOTO Checkcase
:Ask
```



```

If C==2 THEN GOTO Replnext
DELAY(1)
Change=ASKYESNO("This one","Change this one?")
IF Change==0 THEN GOTO FindNext
WINACTIVATE("Notepad")
:Replnext
SENDKEY("+{INSERT}")
GOTO FindNext
:Checkcase
SENDKEY("^^{INSERT}")
Case=CLIPGET()
CLIPPUT(B)
IF Case==A THEN GOTO Ask
GOTO FindNext
:Notfound
SENDKEY("~")
GOTO Ender

```

Once the user has filled out the dialog box and pressed Enter to close it, the routine checks to make sure that search and replacement strings were specified. If either field is empty, the routine exits. Otherwise, it places the search string on the Clipboard. Next, if the user has said to replace A with B throughout the entire file, it sends a Ctrl-Home command to Notepad to move the cursor to the top of the file. Then it sends the command Alt-SF Shift-Ins to Notepad, opening the standard Find dialog box and pasting the search string into its edit field. Then it sets the direction switches in the Find dialog box, sending an Alt-D (down) if the user specified a forward search or a global search, or an Alt-U (up) if the user specified a backward search, and then it sends an Enter (represented in WinBatch by the tilde) to close the Find dialog box and initiate the search.

Next, the routine copies the replacement string (B) to the Clipboard and jumps to the CheckFind routine. CheckFind begins by assigning the caption of the current window to the variable DaWin, using the WINGETACTIVE() command. If the current Window is titled "Notepad", then Notepad has posted a message saying that the search string wasn't found, as shown in Figure 9.9, so the macro jumps to the Notfound routine.

Meanwhile, if the window caption is "Find" then the search string has been found and Notepad has left the Find dialog box on screen so that the user can use its Find Next button to search for the next instance of the search string, as shown in Figure 9.10. When Find Next is selected, the macro responds by sending an Escape character, which closes the Find dialog box.

After closing the Find dialog box, the macro issues the WINACTIVATE command to reactivate Notepad, and then checks the value of the variable E to determine if the search should be case-sensitive. You might expect that it could make use of the Match Case option on Notepad's Find dialog box for this purpose, but it can't, because there is no way for WinBatch to determine whether the Match Case check box is checked when Notepad opens the Find

dialog box. (Notepad keeps track of whether you checked Match Case the previous time the Find command was used, and opens up the dialog box using the previous setting.) WinBatch could easily toggle the Match Case check box to the opposite of its current state, but since it doesn't know what that state is, doing so wouldn't do it much good. So instead, the routine does its case checking using WinBatch commands in the Checkcase subroutine.

Figure 9.9
Notepad's search string not found message

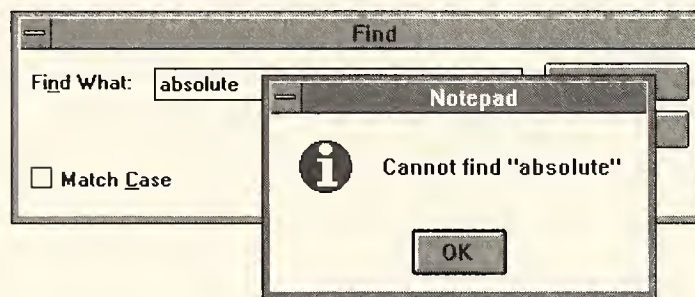
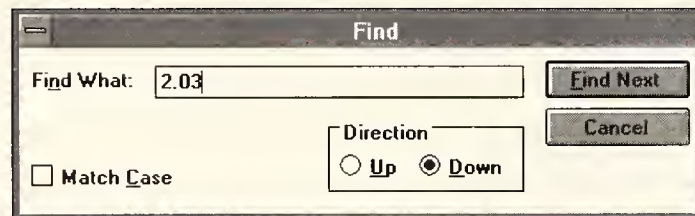


Figure 9.10
Notepad's Find dialog box.



Note that although this method works better than having the macro try to guess whether the Match Case box is checked, it still relies on the user not having left that box in a checked state in the previous Find operation. If you discover the macro is not finding text that you know it should, check the status of the Match Case check box on the Notepad Find dialog box, and deselect it if it is checked.

If the search is not to be case-sensitive, the next line checks the value of the variable C, which determines whether all instances of the search string are to be replaced automatically or if the user wants to approve or cancel each change. If C is equal to 2 (indicating automatic replacement) the macro jumps ahead to the Replnext subroutine. Otherwise, it uses the DELAY command to pause for one second, allowing the user to examine the instance of the search string that Notepad has found and highlighted, and then uses the ASKYESNO function to determine if the current instance is to be changed. If the answer is no (Change=0) the macro jumps back to the FindNext label to find the next instance of the search string. FindNext simply sends an F3, the shortcut key for Notepad's Find Next function, to Notepad, and then goes on to the CheckFind routine.

Meanwhile, if the user indicates that the found text is to be replaced, the macro continues with the Replnext subroutine, which sends the Shift-Ins command to overwrite the highlighted search text with the replacement string, and then jumps back to FindNext.

The Checkcase subroutine, which comes next in the listing, is called when a case-sensitive search is required. It copies the highlighted search text to the Clipboard, and then assigns it to the variable Case. Then it copies the replacement text back to the Clipboard, and uses WinBatch's string-comparison capability to determine if the found text is equal to the search text—which would indicate that both the contents and the case of the two strings match. If so, it jumps back to the Ask routine to continue with processing that instance of the search text. Otherwise, it jumps to FindNext to find the next instance of the search text.

Finally, the NotFound routine is called when Notepad can locate no more instances of the search string. It sends an Escape character to close the Notepad message box, and then jumps to the Ender routine, thus ending the macro.

The AutoIndentSub Routine

The next routine in NOTEPAD.WBT is AutoIndentSub. This routine is designed to automatically indent a new line of text using the same number of tabs as begin the current line.

This capability is useful for several purposes, including creating indented lists, indenting loops and subroutines in program code, and creating outlines. The macro, which can be launched by pressing Ctrl-A, simply counts the number of tabs at the beginning of the current line and then starts the next line with the same number of tabs.

```
:AutoIndentSub
CLIPPUT(Null)
SENDKEY("{END}+{HOME}^{INSERT}")
A=CLIPGET()
A=STRCAT(A,"1")
SendString=STRCAT("{END}",EnterString)
C=ITEMCOUNT(A,TABSTRING)-1
:CountTabs
IF C==0 THEN GOTO DoneTab
SendString=STRCAT(SendString, TabString)
C=C-1
GOTO CountTabs
:DoneTab
SENDKEY(SENDSTRING)
GOTO Ender
```


The routine starts by clearing the Clipboard. Then it sends the key sequence End, Shift-Home, Ctrl-Ins, which first moves the cursor to the right edge of the current line of text, then moves it to the left edge, highlighting everything in between, and finally copies the highlighted line to the Clipboard. Then it assigns the contents of the Clipboard to the variable A, and next appends a single character to that variable. (I used the character 1, but any nonspace, nontab character would do.) Without this, the macro counts one too few tabs on lines that contain no characters but tabs.

Next the macro creates a string called SendString, consisting of an END command (which moves the cursor to the end of the current line) and the contents of EnterString (to start a new line). Then it counts the number of tabs in the string assigned to variable A, using the ITEMCOUNT function, and assigns that result to the variable C.

Next, the macro enters the CountTabs loop. The first line of the loop examines the value of C. If it is equal to 0, the macro jumps to the DoneTab subroutine. Otherwise, it appends a tab to SendString, decrements the value of C, and jumps back to the beginning of CountTabs.

This rather awkward procedure is a workaround for the lack of a FOR-NEXT loop command in WinBatch. It simply decrements the value of C after each tab is added until C is equal to 0. In a language such as BASIC, which includes a FOR-NEXT structure, this routine would be rendered in a much more easily understood manner, as follows:

```
FOR X=1 TO C
SENDSTRING=SENDSTRING+TabString
NEXT X
```

In the WinBatch code, once C is equal to 0 there are no more tabs to be added, so the macro jumps to DoneTab, which sends SendString to Notepad (moving the cursor to the end of the current line, sending the Enter character, and then sending however many tabs the macro has added to SendString). Then it jumps to the Ender routine, signaling the end of the routine.

The CopperSub Routine

The next two routines, CopperSub and ClowerSub, are used to convert selected text in the current Notepad file to upper- or lowercase. (As a side-note, these were the only two routines I knew WinBatch was capable of before I started this project; they are inspired by similar examples in the WinBatch manual.)

CupperSub's code looks like this:

```
:CupperSub
CLIPPUT(Null)
SENDKEY("^ {INSERT}")
CLIPPUT(STRUPPER(CLIPGET()))
SENDKEY("+ {INSERT}")
GOTO Ender
```

CupperSub begins by clearing the Clipboard, and then places the selected text on the Clipboard. It then combines the STRUPPER function (which converts a string to uppercase) and the CLIPGET() command to replace the current contents of the Clipboard with an all-capitals version of those contents. Then it pastes the text back into Notepad and jumps to the Ender routine.

The ClowerSub Routine

ClowerSub is identical to CupperSub, except that it uses the STRLOWER function to convert the Clipboard's contents to lowercase:

```
:ClowerSub
CLIPPUT(Null)
SENDKEY("^ {INSERT}")
CLIPPUT(STRLOWER(CLIPGET()))
SENDKEY("+ {INSERT}")
GOTO Ender
```

The FastcountSub Routine

The final major routine in NOTEPAD.WBT is FastcountSub, which is used to obtain word and character counts for the current Notepad file. It is the most complicated of the routines listed here, not least because the process of defining what constitutes a word is more complex than you might think.

When I started work on this routine, I assumed that counting words would be easy. I thought of a word as any block of characters separated from its neighbors by a space on either side. But that definition proved surprisingly inadequate even for straight text, and laughably off-base for text that includes programming code or other nonstandard elements.

For instance, using that definition, each of the following items would constitute one word:

decisions--not

reality...unless

read-only

```
EXCLUSIVE(@ON)
```

```
CLIPPUT(STRLOWER(CLIPGET()))
```

Even worse, this *list* of five items (each appearing on its own line with no spaces before or after it) would as a whole be recognized as a single word by a word-count algorithm that looked only for preceding and following spaces.

So I was forced to come up with a broader method of counting words, one that recognized that characters such as brackets, parentheses, ellipsis points, and hyphens should all be recognized as delimiters between words, as should tabs, carriage returns, and line feeds.

The other complicating factors were that WinBatch's string-manipulation functions fail if you try to work with strings longer than about 10k in length, and that although it offers routines such as STRSUB and STRINDEX, which allow you to search for characters within a string, they are tortoiselike in performance. I knew this routine would never function with the speed of one written in assembly language, but I wanted it to be usable.

After much experimentation, I came up with the following solution. It's still not perfect, but it delivers a very close to accurate (within 5 percent on most files) count of words and characters, and does so with reasonable speed (counting 7,600 words in a 48k file in about 25 seconds on my 386/33). Here's what it looks like:

```
:FastcountSub
EXCLUSIVE(@On)
Space=" "
Words=0
Chars=0
SENDKEY("^ {HOME}")
:GetNext
SENDKEY("+ {PGDN 3} + {END}")
SENDKEY("^ {INSERT}")
SENDKEY("{RIGHT}")
A=CLIPGET()
IF A==NULL THEN GOTO FINAL
CLIPPUT(NULL)
STRREPLACE(A,")",Space)
STRREPLACE(A,"}",Space)
STRREPLACE(A,"]",Space)
STRREPLACE(A,"(",Space)
STRREPLACE(A,"{",Space)
STRREPLACE(A,"[",Space)
STRREPLACE(A,":",Space)
STRREPLACE(A,"...",Space)
```



```

STRREPLACE(A,";",Space)
STRREPLACE(A,"-",Space)
STRREPLACE(A,"/",Space)
STRREPLACE(A,CRLFString,Space)
STRREPLACE(A,TabString,Space)
Chars=Chars+ STRLEN(A)
Words=Words+ITEMCOUNT(A,Space)
GOTO GetNext
:Final
SENDKEY( "^{HOME}")
EXCLUSIVE(@Off)
BEEP
MESSAGE("Word Count","%Chars% characters in %Words%
words.")
GOTO Ender

```

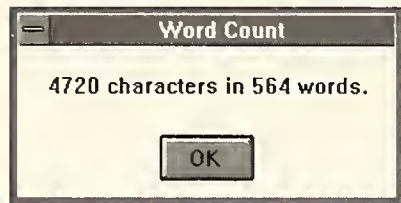
The routine begins by setting WinBatch's EXCLUSIVE switch to on, then defines the variable Space as a space character and initializes the variables Words and Chars to 0. Next it sends a Ctrl-Home command, to instruct Notepad to move the cursor to the top of the file, and then enters the GetNext loop.

GetNext starts by highlighting three screen-pages of text: issuing three Shift-PgDn commands and a Shift-End command to move to the end of the last line of the third page. Then it copies the selected text to the Clipboard, and sends a Right Arrow keystroke to move the cursor to the first character of the next line, removing the highlighting from the current selection. Next it assigns the contents of the Clipboard to a variable named A, and checks to see if A is empty. If so, then it has reached the end of the file, and so it jumps to the Final subroutine.

Otherwise, it clears the Clipboard and issues 13 STRREPLACE commands in a row, each of which replaces all instances of a possible word delimiter (including opening and closing parentheses and brackets, hyphens, ellipsis points, tabs, and carriage returns) with a space. Then it counts the number of characters and the number of words in string A. It uses the WinBatch STRLEN command to obtain a count of the number of characters in the string, and the ITEMCOUNT command to obtain the word count. ITEMCOUNT returns a value equal to the number of items (other than spaces) in string A that are separated from one another by spaces. Then it adds that result to the current value of Words and loops back to get the next three screen-pages of text.

When the macro reaches the end of the file (indicated by the variable A being empty) it jumps to the final routine, which returns the cursor to the top of the file, deactivates WinBatch's EXCLUSIVE switch, beeps the PC speaker to wake up the user, and opens a message box that delivers the word

and character counts it has obtained, as shown below. Then after the user closes the message box, the macro concludes by jumping to the Ender routine.



The Ender Routine

The Ender routine, referred to so often above, consists of just two lines:

```
:Ender
CLIPPUT(OldClip)
EXIT
```

The first line in Ender restores the original contents of the Clipboard, and the second stops execution of the macro. There are two advantages to placing these commands here, rather than at the end of each macro. The first is that it ensured that I didn't forget to restore the Clipboard contents after any of the macros that use the Clipboard. The second is that it offers the ability to add other routines here that will be executed at the end of any macro that calls Ender, should the need to do so ever arise.

The Dialog Box Routines

The next three routines are used to create the dialog boxes used by the OpenerSub, InsertSub, and ReplaceSub routines. They follow a similar order and logic, starting with a FILEOPEN command to open the file for the dialog box template, then a series of FILEWRITE commands that write the dialog box template instructions, then a FILECLOSE command, and finally a GOTO command that returns to the calling routine.

The WinBatch template format is simple, if not particularly intuitive. Blank lines in the template file create blank lines in the dialog box, so in the routines that follow, this command denotes a blank line:

```
FILEWRITE(hFP, "")
```

Static text is created simply by entering it into the template. So the command

```
FILEWRITE(hFP, "Select file to insert:")
```

would create the static text string, "Select file to insert:" when the DIALOG-BOX command is used to load the template file.

Radio buttons are created by indicating the variable to be used to store the number of the button that is selected from the button group, followed by a caret, followed by the individual button's number. The following would create a line in the template file to produce four radio buttons, labeled "*.TXT", "*.INI", "*.1ST", and "READ*.*", respectively:

```
FILEWRITE(hFP, "[A^1 *.TXT] [A^2 *.INI] [A^3 *.1ST] [A^4 READ*.*]")
```

File list boxes with accompanying fields in which you can enter a new file specification are created by following a variable that contains the default file specification with a dollar sign to create the edit field. Then a series of lines containing the same variable, followed by a backslash, is used to determine the height of the list box. So the following code would create the file-specification edit field, followed by a blank line, followed by a list box that is three lines high:

```
FILEWRITE(hFP, "[a$                               ]")
FILEWRITE(hFP, "[a$                               ]")
FILEWRITE(hFP, "[a\                               ]")
FILEWRITE(hFP, "[a\                               ]")
FILEWRITE(hFP, "[a\                               ]")
```

Standard edit fields are created by following the variable to which the edit field's contents are to be assigned with a number sign. The following command would create an edit field with the prompt "Replace:" and would assign the contents of the edit field to the variable A:

```
FILEWRITE(hFP, "Replace: [A#                               ]")
```

Finally, check boxes are created by following the variable to which the result of the check box is to be assigned with a plus sign, then the value to be assigned to the variable if the check box is checked, then the text of the check-box prompt. So the command:

```
FILEWRITE(hFP, "[E+1Match case?]")
```

would write a line in the template file to create a check box with the prompt "Match case?". The variable E will be assigned the value 1 if the user checks the check box.

Without further ado, then, here are the listings of the OpenerWrite, InsertWrite, and ReplaceWrite subroutines.

The OpenerWrite Routine

OpenerWrite contains this code:

```
:OpenerWrite
hFP=FILEOPEN(STRCAT(HOMEDIR, "MASK.DLG"), "WRITE")
FILEWRITE(hFP, "")
```



```

FILEWRITE(hFP,"[A^1 *.TXT] [A^2 *.INI] [A^3 *.1ST] [A^4 READ*.*)"
FILEWRITE(hFP,"[A^5 *.ASC] [A^6 *.WDF] [A^7 *.WBT] [A^8 *.DLG]")
FILEWRITE(hFP,"")
FILECLOSE(hFP)
GOTO OpenerDraw

```

You could create the same template file manually (rather than automatically, as is done here) by entering the following text into Notepad or another text editor and saving the file as MASK.DLG (being sure to leave a blank line at the beginning and end of the file to create empty lines in the dialog box):

```

[A^1 *.TXT] [A^2 *.INI] [A^3 *.1ST] [A^4 READ*.*)
[A^5 *.ASC] [A^6 *.WDF] [A^7 *.WBT] [A^8 *.DLG]

```

The InsertWrite Routine

InsertWrite contains the following code:

```

:InsertWrite
hFP=FILEOPEN(STRCAT(HOMEDIR,"INSERT.DLG"),"WRITE")
FILEWRITE(hFP,"Select file to insert:")
FILEWRITE(hFP,"[a$                ]")
FILEWRITE(hFP,"")
FILEWRITE(hFP,"[a\                ]")
FILEWRITE(hFP,"[a\                ]")
FILEWRITE(hFP,"[a\                ]")
FILEWRITE(hFP,"[a\                ]")
FILEWRITE(hFP,"[a\                ]")
FILEWRITE(hFP,"[a\                ]")
FILEWRITE(hFP,"[a\                ]")
FILEWRITE(hFP,"[a\                ]")
FILEWRITE(hFP,"[a\                ]")
FILECLOSE(hFP)
GOTO InsertDraw

```

The ReplaceWrite Routine

Finally, here's the ReplaceWrite routine:

```

:ReplaceWrite
hFP=FILEOPEN(STRCAT(HOMEDIR,"REPLACE.DLG"),"WRITE")
FILEWRITE(hFP,"")
FILEWRITE(hFP,"Replace: [a#                ]")
FILEWRITE(hFP,"")
FILEWRITE(hFP,"With: [b#                ]")
FILEWRITE(hFP,"")
FILEWRITE(hFP,"Global replace: [c^1Query first] [c^2Replace all]")
FILEWRITE(hFP,"Search: [d^1Forward] [d^2Reverse] [d^3Entire file]")
FILEWRITE(hFP,"[e+1Match case?]")
FILECLOSE(hFP)
GOTO ReplaceDraw

```

Wrapping Up The Ultimate Notepad

That concludes discussion of The Ultimate Notepad project. You'll find all the code for the project on the disk that accompanies this book. Feel free to experiment with it and adapt it to your own needs. You might, for instance, start by modifying the MergeSub routine to call the file mask-selection dialog box used by OpenerSub, rather than having the Insert File dialog box default to offering a selection of TXT files.

Although The Ultimate Notepad adds some truly useful features to the Windows Notepad, it is in the end limited by both Notepad's bare function set and the limitations of the WinBatch language. In the following chapters you'll see how programming efforts of similar scope, tackled with more capable tools, can yield more impressive results.

C H A P T E R

10

**Presenting Data—
Who's Who at
PC/Computing?**

*In the Beginning
Was Confusion*

*Application-Design
Issues in Plus*

*Exploring the
Application*

*Wrapping Up the
Who's Who Application*

THIS CHAPTER'S PROJECT, WHO'S WHO AT PC/COMPUTING, WAS DEVELOPED in preparation for that company's move to spanking new quarters in the Mystic Center office complex in the Boston suburb of Medford, Massachusetts. This interactive system was designed to provide a variety of information about each PC/Computing employee, including his or her title, telephone extension, office location, and place on the magazine's organizational chart. Although the original intent was to use it internally, as the project progressed plans were also made to distribute it externally as a marketing promotion.

As it turns out, however, the application was never widely distributed either internally or externally. In a curious twist that will make sense if you've ever faced the prospect of spending your days in Medford, the powers that be announced shortly before the move that two months after arriving in Medford the magazine's offices would move yet again—this time to Foster City, California. In the excitement that followed, the Who's Who project was permanently back-burnered.

Nevertheless, this project remains a good example of how easily you can develop an attractive and practical interactive information system using only a minimal amount of program code, given the proper Windows development tools.

In the Beginning Was Confusion

Assuming that you don't succumb to toxic fumes from the freshly laid carpeting, the worst thing about a move to a new office is the confusion. If you're lucky, you can find your own office, but that's about it. Where's the conference room? Where's the fax machine? Where's the bathroom?

If anything, I'm more prone to this confusion than most. And, in what may be a related malady, I've also been known to forget the names, titles, and areas of responsibility of people I see and work with every day. So it was purely out of self-interest that as the move to our new offices in Medford approached, I started thinking about an application I'd seen demonstrated on a Sun workstation several months before.

The Sun application linked an office floor plan with individual employee records. I thought the concepts embodied in that application could be expanded to serve as a guide for locating not only people but resources such as laser printers, conference rooms, and coffee machines in the new office. The resulting Windows application could also provide a picture of both the overall organizational structure of PC/Computing and the areas of responsibility of each individual employee.

The functional goals for the Who's Who application developed out of those initial musings. I wanted to start with a floor plan displaying the layout of the new offices, and then link each office on the plan to a personnel

information sheet that would provide vital statistics about the occupant of that office—including name, title, job description, and telephone extension. I also wanted to be able to conduct free-form searches through the database, and to locate each individual on an organization chart.

Despite its obvious uses, I knew this application was destined to fall into the “nice to have” rather than “need to have” category. So its operation had to be completely intuitive; otherwise, it was unlikely many people would take the trouble to use it. For the same reason, the application had to be fun to use. So I wanted to make it colorful and graphic, with scanned photographs of each employee and attractive representations of the floor plan and organizational chart.

Choosing the Tool

Remarkably, considering that this application's functional requirements would have overwhelmed all but the most powerful DOS-based application-development tools, any number of Windows tools could have handled this project. It certainly was within the reach of any of the Windows BASICs, or of Turbo Pascal for Windows. But I also could have pulled it off using a database development tool such as Superbase 4 or ObjectVision or even dbFast/Win. It might even have been possible to build it using the macro language for Lotus's Ami Pro word processor. All of those languages offer the element critical to making this application work: the ability to link graphic objects (such as the depiction of an office on a floor plan) to other program screens.

However, in selecting a program-development tool for this application I followed my own advice from Chapter 4 and tried to identify the right tool for the job. This meant using the tool that would require the least amount of effort and smallest amount of program code to do the job. Which in this case added up to it being a perfect project for a Hypercard-like tool such as Asymmetrix's Toolbook or Spinnaker Plus. Both are tailored for this kind of job—one in which you want to present a relatively small amount of data in a graphic manner, with links between disparate data items and simplified support for responding to user-interface events such as mouse moves and button clicks. Unlike all of the other Windows development tools that would have allowed me to build this application, Toolbook and Plus could do most of the work for me, allowing me to spend more time thinking about design details and less pondering the subtle nuances of Windows's message queue.

My preliminary evaluation of Toolbook and Plus suggested that they were fairly equally matched. Their command languages offer similar capabilities, and both provide adequate (albeit far from blazing) performance. The decision between them would have come down to a coin toss except for one thing: Spinnaker also offers a Macintosh version of Plus, whereas Toolbook is available for the PC only. Had this application ever gone into wide circulation

at PC/Computing, it would have needed to be available to the art and production staffs, who work on Macintosh computers, as well as to the rest of the editorial, marketing, and sales staffs, who all use Windows-capable PCs. So the availability of a Macintosh version tipped the scales in Plus's favor.

Application-Design Issues in Plus

Spinnaker Plus makes plain what some other Windows development tools only suggest: For all practical purposes, the interface is the application. With Plus you have no choice but to start off the development process by building a user interface for your application, because every line of program code is associated with a user action. You cannot create program code unless you have already created the user-interface element it is to be associated with.

That restriction might sound a little confining, since it would seem to eliminate the possibility of general purpose subroutines that can be called by several top-level routines. However, the user interfaces constructed by Plus are structured in such a way that you can still write global procedures, even though they must be structured as responses to user actions.

Stacks

In Plus, as in the Macintosh application Hypercard, which inspired it, applications are referred to as *stacks*. Each stack is composed of one or more *cards*—or individual screens—each of which is associated with a *background*, which may be shared with other cards. User-interface objects such as buttons and bitmaps may appear on either the card or the background. In the former case they are specific to the card upon which they appear; in the latter they will appear on every card sharing that background.

Plus responds to user actions through a hierarchical series of *handlers*. For instance, when you press the left mouse button while the cursor is on a button, Plus interprets the message it receives from Windows as a result of that action as MOUSEDOWN, and responds by looking at the code associated with that button for an ON MOUSEDOWN handler. Thus, if the following code was associated with a button, Plus would respond by beeping the PC's speaker five times:

```
ON MOUSEDOWN  
  BEEP 5  
END MOUSEDOWN
```

However, if there is no ON MOUSEDOWN code associated with the button (or if the button's code included a PASS MOUSEDOWN statement), Plus would then look at the code associated with the card to see if it contained an ON MOUSEDOWN routine, and if so would execute it.

If the card didn't contain, or passed, the ON MOUSEDOWN statement, Plus would look at the code associated with the background. And if it failed to find an ON MOUSEDOWN there, or if the background procedure contained a PASS MOUSEDOWN statement, Plus would look next to the code associated with the stack.

For instance, if the code listed above was associated with the stack, and none of the other elements on the card or background intercepted the MOUSEDOWN first, Plus would beep the PC's speaker five times anytime you pressed the left mouse button, no matter what card or background was visible or what user interface element was under the mouse pointer (making for a very annoying application).

Finally, if none of the four possible locations in the current stack (the button, card, background, or the stack itself) contained an ON MOUSEDOWN procedure, Plus would continue to look for one further up its message-passing hierarchy, first looking in any library scripts that are defined for the current stack, then in the Home stack, and finally in the Plus application itself. If it can't find a handler for the MOUSEDOWN message in any of those locations, it ignores the message, taking no action.

Thus, as you build an application in Plus, one of the key decisions you have to make over and over is where to put a segment of code. If the action you want Plus to carry out is specific to a single user-interface element, then you should associate the code with that element. But if it is applicable to every element on a particular card, on a particular background, or in the entire stack, then you should associate it with the card or background or stack, as the case may be.

Drawing the Interface

Since every line of code in Plus is associated with a user action, before you can write any code you have to create the elements with which the user will interact.

Like many Windows development tools, Plus provides a palette of standard user-interface elements from which you can choose. To create a button, for instance, you would select the Button tool and draw the button on the screen, as shown in Figure 10.1.

Next, you would double-click on the button you've just drawn, to open its Info dialog box, as shown in Figure 10.2. From this dialog box you can modify the button's appearance, identify an icon that it is to display (from Plus's array of predefined icons), link it to another card (so that when the user pushes the button Plus displays that card), or modify its script.

When you select the button labeled "Script" on the Button Info dialog box, Plus opens the button's Script window, which holds all the scripts associated with that button. So, if you have already written ON MOUSEDOWN and ON MOUSEUP procedures for the button, both would appear in its script window, as shown in Figure 10.3.

Figure 10.1

A newly drawn pushbutton

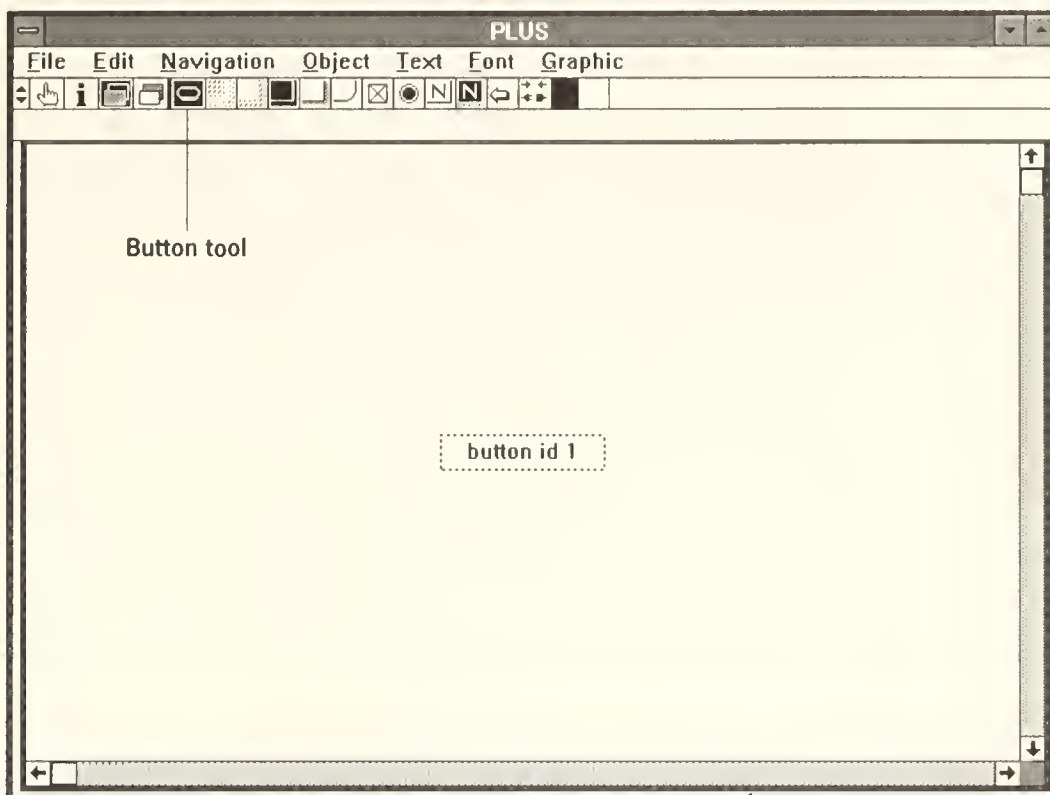


Figure 10.2

The Button Info dialog box

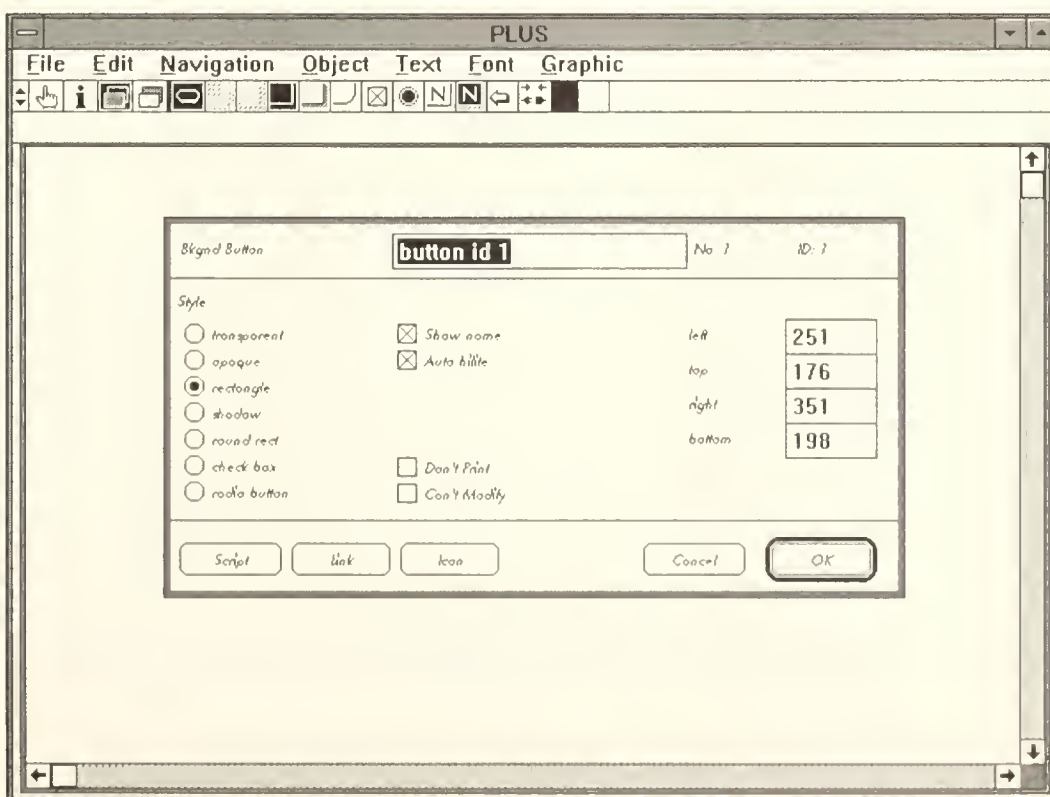
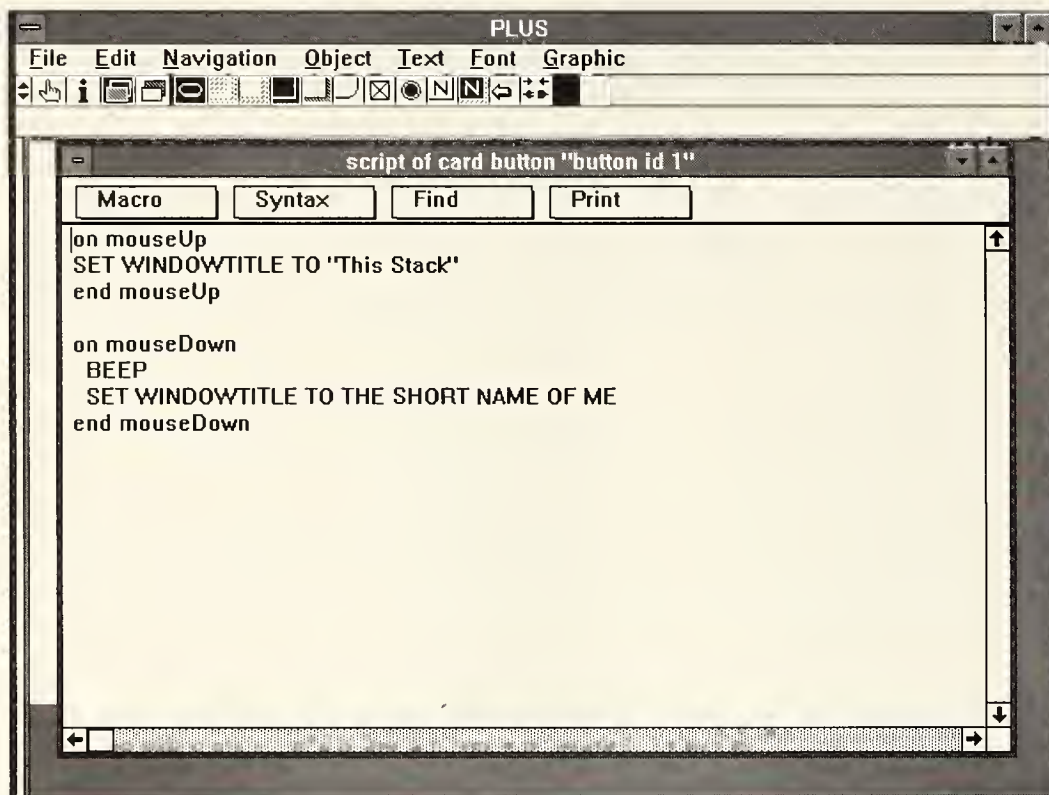


Figure 10.3

The Script window for a button



Plus's Limitations

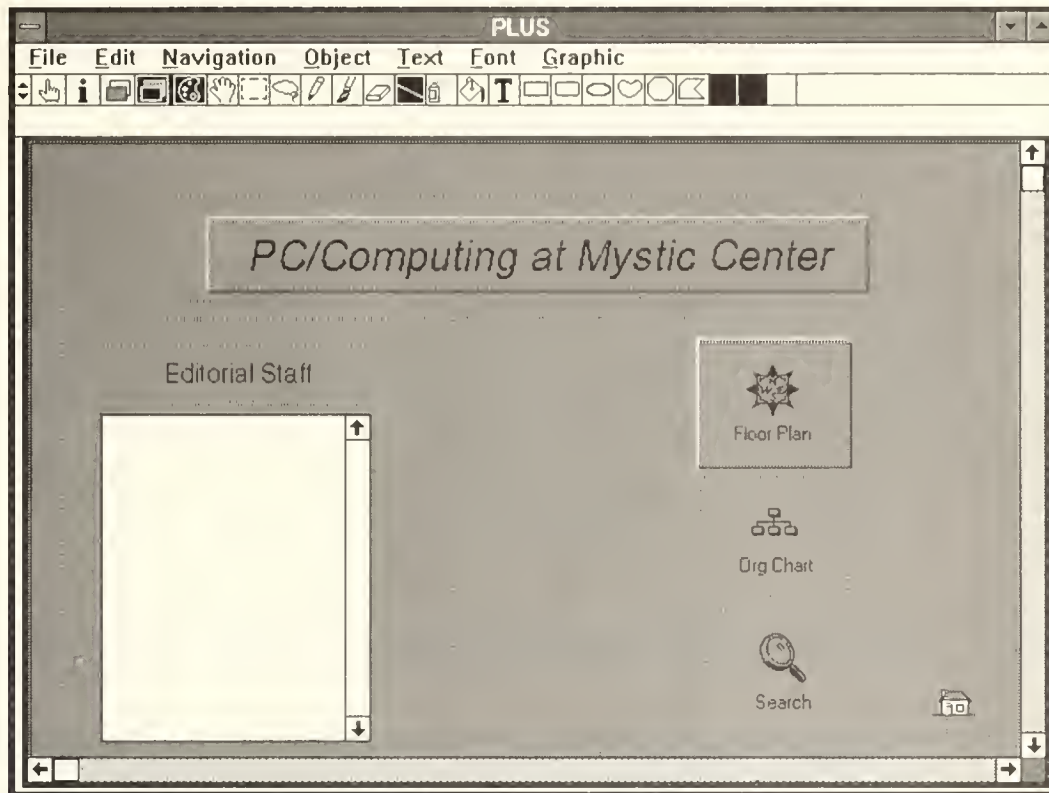
Plus was originally developed on the Macintosh as a superset of Hypercard, and its array of standard user-interface elements betrays this heritage. Missing are such standard Windows elements as drop-down list boxes and group boxes, as well as the enhanced three-dimensional controls offered by many Windows development tools.

However, you can simulate many of these elements through judicious use of Plus's painting and drawing tools, which provide the ability to create bitmapped or vector-based graphic objects on a card or background. For instance, I created the illusion of three-dimensional frames on the opening screen of the Who's Who application by first creating a light gray paint object (a bitmapped graphic object), which covers the entire card, and then adding draw objects (vector-graphic objects) with black lines along their bottom and right edges and white lines along the top and left edges, as shown in Figure 10.4.

Paint and draw objects in Plus are not static as they are in many programs. Instead, you can associate scripts with them, just as you can with buttons, list boxes, and edit fields.

Figure 10.4

Creating three-dimensional frames



Exploring the Application

The Who's Who application is built around four basic cards, or program screens:

- The Opening screen presents an alphabetical list of all employees. Each name is linked to the employee's personnel card, and buttons allow you to initiate a free-form search, jump to the Organization Chart or Floor Plan, or return to Plus's Home card (the opening Plus screen).
- The Floor Plan screen is where each office or cubicle is linked to its occupant's personnel card. This screen also includes buttons for jumping to the Opening screen or the Organization Chart, and for initiating a search.
- The Personnel Card screen is a template that is copied for each employee. It features space for the employee's photo; database fields for the employee's name, title, telephone extension, and department; a multiline text box that holds a description of the employee's areas of responsibility; and buttons for initiating a search and jumping to the Organization Chart, Floor Plan, or Opening screen.

- The Organization Chart screen is a hierarchical chart that utilizes standard pushbuttons to represent each employee. These buttons are linked to the employee personnel cards. Additional buttons on the Organization Chart screen provide links to the Floor Plan and Opening screens, and allow the user to initiate a free-form search.

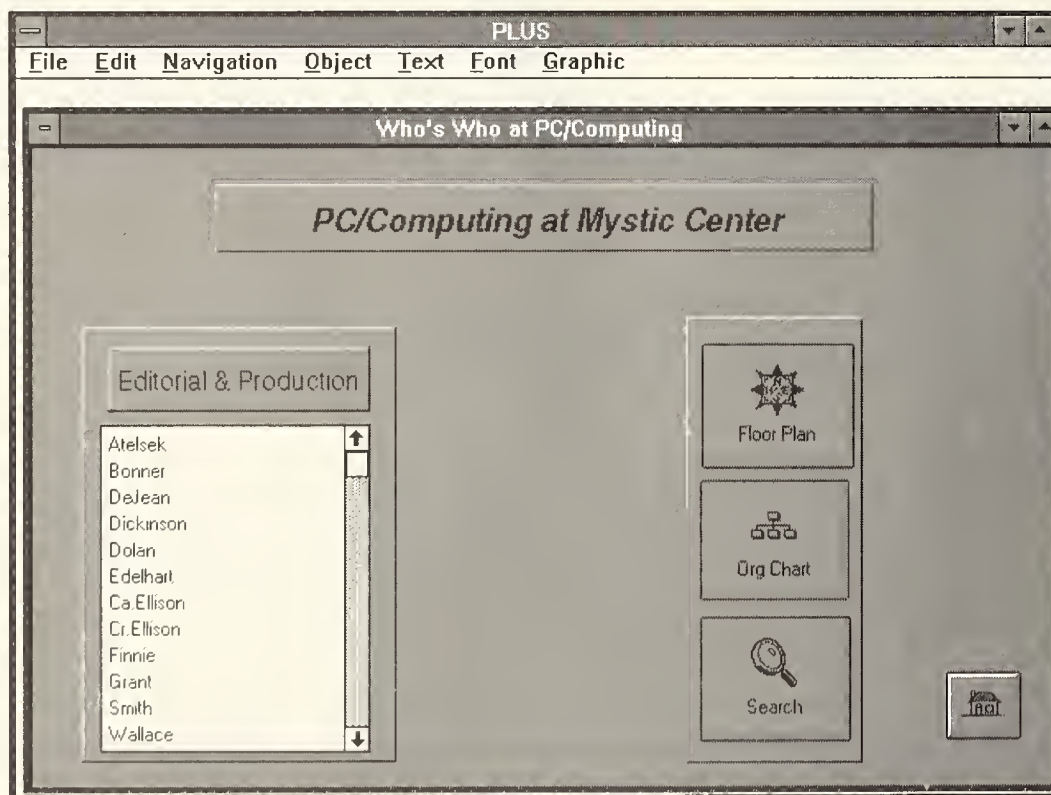
Let's look at each of these in detail.

The Opening Screen

Who's Who at PC/Computing's Opening screen, shown in Figure 10.5, greets you by presenting several options for locating the information you seek. If you know the last name of the person you're looking for, you can select his or her name from the list box at the left of the screen. Doing so will take the application to that individual's personnel information card.

Figure 10.5

The Who's Who application's Opening screen



If, on the other hand, you want to know the physical location of someone's office or of a departmental resource, you can jump to the Floor Plan screen by clicking on the corresponding button—identified by a compass icon.

If you want to see what position someone occupies on the magazine's organization chart, you can click the button labeled "Org Chart", or if you

want to search for individuals on the basis of their title or department, or a certain phrase appearing in their job description, you can click on the button labeled "Search".

Finally, if you've finished using the Who's Who application, you can return to Plus's Home stack (its opening menu) by clicking the button depicting a house.

Now let's look at the scripts that power this screen.

The OpenStack Script

The first script executed as the Who's Who application is launched is associated with the ON OPENSTACK handler. This is a stack-level handler that responds to the OPENSTACK message Plus generates when it launches a stack. Thus, you could think of it as sort of an AUTOEXEC.BAT program for the stack.

The ON OPENSTACK script looks like this:

```
ON OPENSTACK
LIBRARY "PTOOLS.STA"
WINDOWTOCARDSize
HIDE TOOLBOX
GLOBAL High
PUT "" INTO High
LOCK RECENT
SET THE AUTOSAVE TO 0
END OPENSTACK
```

The script starts by loading Plus's Power Tools library—a set of functions not built into the standard Plus application but available to any stack that loads the library PTOOLS.STA. Next the script uses the WINDOWTOCARDSize command from that library to adjust the size of the Plus window to match that of the opening card in the Who's Who stack.

The next command, HIDE TOOLBOX, hides the palette of development tools that normally appears at the top of the Plus window. Next the script establishes a global variable called High and initializes it as an empty string. (The PUT "" INTO High is the equivalent of a High="" statement in BASIC.)

Then, the script sets a pair of standard Plus toggle switches. Plus normally maintains a series of thumbnail images of each card that you view, allowing you to jump to a card you've already viewed by picking its image from a screen of "recent" cards. In addition, Plus normally saves the current stack to disk at regular intervals. However, both of these facilities slow Plus's performance, so in order to get the fastest possible performance from the Who's Who application, the script issues the commands LOCK RECENT (turning off the thumbnail feature) and SET THE AUTOSAVE TO 0 (turning off the AUTOSAVE feature).

Finally, the script issues the END OPENSTACK command, concluding processing of the OPENSTACK message.

The Home Button Script

The Home stack acts as sort of an opening menu for Plus; from it you can modify various program settings and access other stacks. The Home Button in Who's Who is used to shut down the Who's Who application and return to the Home stack. Its script looks like this:

```
ON MOUSEUP
GO Home
END MOUSEUP
```

Here, the script responds to the ON MOUSEUP message, which is sent when you release the left mouse button over the object to which the script is attached, by issuing the simple command, GO Home.

In addition to the MOUSEUP and MOUSEDOWN handlers, Plus also interprets a variety of other mouse-related events, including among others MOUSEENTER (which occurs when the mouse pointer touches the attached object), MOUSELEAVE (which occurs whenever the mouse pointer is removed from the object), MOUSESTILLDOWN (sent repeatedly while the left mouse button is held down), and MOUSEWITHIN (sent repeatedly while the mouse cursor rests on the object). This wide variety of mouse-related events offers you great flexibility in responding to different user actions, as will be seen in some of the routines that follow.

The next routine is attached to the Floor Plan button on the main screen.

The Floor Plan Button Script

The script for the Floor Plan button is only slightly more complex than that for the Home button.

```
ON MOUSEUP
SET CURSOR TO 4
LOCK SCREEN
GO TO CARD AAAB
UNLOCK SCREEN WITH VISUAL EFFECT SCROLL RIGHT
SET WINDOWTITLE TO "Who's Who?"
END MOUSEUP
```

The Floor Plan button code responds to the MOUSEUP event (sent when the user releases the left mouse button over the Floor Plan icon) by first setting the cursor to shape 4. Shape 4 looks like a wristwatch, and is used here to indicate that the application is busy, in the same way that most Windows applications use the hourglass. Among the other standard cursor shapes in Plus are the arrow, for item selection (shape 0), and the I-beam cursor, for text entry (shape 1).

Next the script locks the screen, which prevents Plus from updating it until the UNLOCK command is issued (or until the current script ends). Then the command GO TO CARD AAAB is issued, telling Plus to draw the Floor Plan screen in memory. Because the screen is locked, Plus doesn't display the Floor Plan screen as it is being drawn—that would slow the drawing operation. Instead, it draws the screen completely in memory, and then displays it and makes it the active card when the script executes the command UNLOCK SCREEN WITH VISUAL EFFECT SCROLL RIGHT.

The VISUAL EFFECT SCROLL RIGHT part of that command tells Plus to scroll the image of the new card over that of the current card, so that it looks like the new card is sliding from left to right onto the screen. Plus supports a variety of visual effects that provide interesting transformations from screen to screen: the SCROLL (or WIPE) RIGHT, LEFT, UP, and DOWN commands; IRIS (or ZOOM) OPEN and IRIS CLOSE (in which the new card either appears first at the center of the old card and then zooms open from there, or appears at the periphery of the new card and then takes over the screen from the outside edge in); and DISSOLVE (in which the current screen seems to disintegrate as it changes into the new screen).

The final command in the Floor Plan script sets the title bar of the Floor Plan window to “Who's Who?”

Card-Naming Conventions

You might be wondering how the Floor Plan became card AAAB.

Every object in Plus has a name. When you create a new button or other object, Plus names it automatically. For instance, a button drawn on a card's background might be given the name Background Button 4. These names are editable, and in most cases you'll want to give them names that will be more evocative of their actual use. For instance, I renamed the buttons for the Floor Plan and Organization Chart, “Floor Plan” and “Org Chart”.

Plus also names new cards automatically, giving them names like Card 5 or Card 34. So how did the Floor Plan become card AAAB, rather than Card 4 or Card 2 or whatever Plus had originally named it?

As you'll see below, the script automatically renames new personnel cards to reflect the last name of the person about whom they present information. For instance, the card with my employee information is named Bonner. I wanted the script to automatically keep these personnel cards in alphabetical order, so that the user could employ Plus's standard navigation commands to go to the next card (Ctrl-3) or previous card (Ctrl-2) alphabetically. The only practical way to do this, however, was to use the CARDSORT command to alphabetically sort the entire stack of cards (in ascending order by card name) every time a new card is added to the stack. The problem with that was that a card named Opening Screen would be unlikely to end up as the first card after a sort procedure. Instead, it would end up in the middle of

the pack, mixed in with the personnel cards, Floor Plan, Organization Chart, and others.

To solve this problem, I named the Opening Card AAAA, the Floor Plan AAAB, the template card from which all new personnel cards are created AAAC, and the Organization Chart ZZZ_OrgChart. This ensures that—unless PC/Computing hires someone whose last name begins with more than three A's or Z's, the Opening Screen will always be the first card, the Floor Plan will be the second, the template card will come third, followed by the personnel cards, and the Organization Chart will always be the last card.

The Organization Chart Button Script

As you might expect, the Organization Chart button script opens the Organization Chart. It uses the following code:

```
ON MOUSEUP
SET CURSOR TO 4
LOCK SCREEN
GO TO LAST
UNLOCK SCREEN WITH VISUAL EFFECT SCROLL LEFT
SET WINDOWTITLE TO "Organization Chart"
END MOUSEUP
```

As you can see, this code is very similar to that of the Floor Plan button, varying only in that it substitutes the keyword LAST for a card name in its GO TO command (thus telling Plus to show the last card in the stack) and in the visual effect with which it unlocks the screen.

The Organization Chart buttons that appear on the personnel cards and the Floor Plan screen feature nearly identical scripts. They vary from this one only in the visual effects used to reveal the Organization Chart screen. The use of different transformation effects adds some visual interest to the application, but doesn't affect its function.

Next we'll look at the script for the Search button on the Opening Screen.

The Search Button Script

This routine makes use of the Plus ASK function, which requests a single line of input from the user and assigns that input to a system variable known as IT.

```
ON MOUSEUP
GLOBAL WHATTOFIND
ASK "Enter search string:"
IF IT IS EMPIY THEN EXIT MOUSEUP ELSE PUT IT INTO WhatToFind
LOCK SCREEN
GO NEYT
FIND WhatToFind
IF THE RESULT IS NOT EMPTY THEN
```



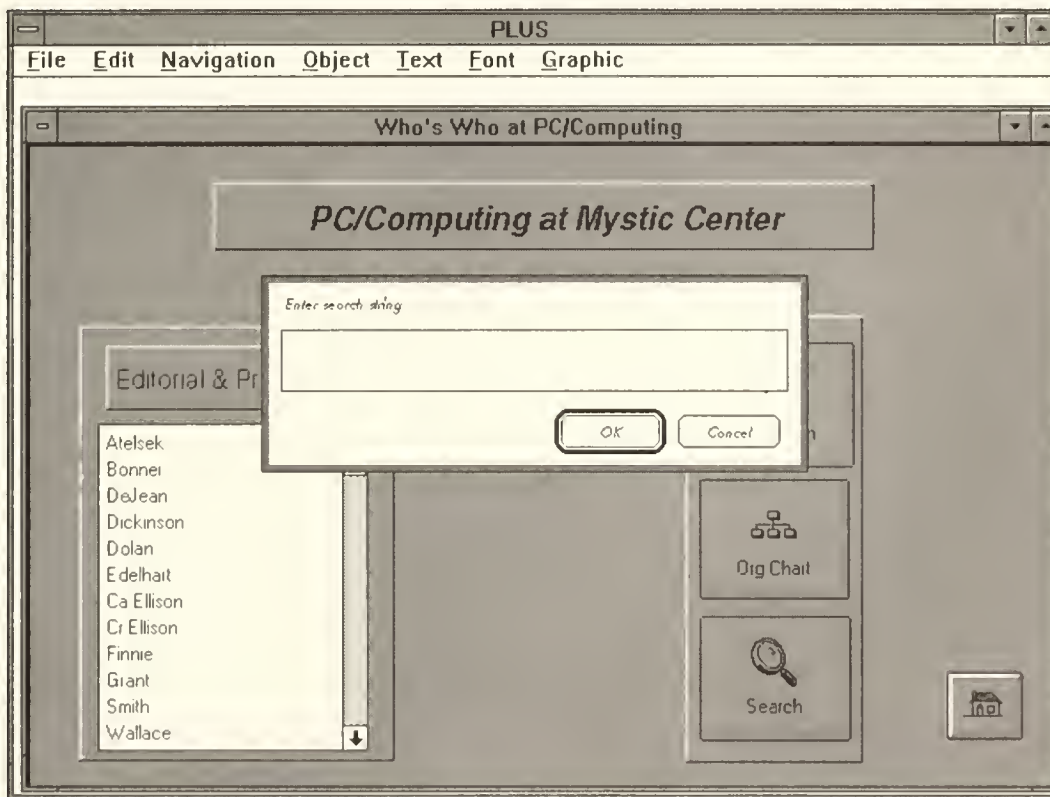
```

BEEP
ANSWER "Couldn't find " & WhatToFind
GO PREVIOUS
EXIT MOUSEUP
END IF
END MOUSEUP

```

The script begins by creating a global variable called WhatToFind. Next the script opens an input box and prompts the user to enter a search string, as shown in Figure 10.6.

Figure 10.6
The Search
function's input box



The result of that input box (the text the user enters) is automatically assigned to the system variable IT. So the script examines the contents of IT and if it is empty (which would be the case if the user pressed the Enter key without entering any data, or canceled the operation by pressing the Esc key or selecting the Cancel button), the script exits the MOUSEUP handler routine. Otherwise, it assigns the value of IT to WhatToFind.

Next the script locks the screen, selects the next card, and then issues the Plus FIND command, instructing Plus to find the next card field that contains the text in WhatToFind. (I decided to start the process by going to the next card because otherwise Plus would find WhatToFind on the current card if it exists there, which is generally not what the user has in mind.)

Plus uses a system variable called `RESULT` to store error messages. This allows your application to examine the value of `RESULT` to determine whether or not the previous operation was successful. If Plus finds a match for `WhatToFind`, it automatically makes the card on which the match was found the current card, and the variable `RESULT` will be empty. Otherwise, `RESULT` will contain the error message “card not found”. So the script tests the results of the Search operation by examining the contents of `RESULT`.

If `RESULT` is not empty, then the Search operation must have been unsuccessful. So the script beeps the PC's speaker, displays the message “Couldn't find” followed by the contents of `WhatToFind`, makes the previous card (the card the operation started with) the current card, and exits the `MOUSEUP` procedure.

If, on the other hand, `RESULT` is empty, then the Search operation was successful, and the card on which the search text was found has automatically been made the current card. So the script simply issues the `END MOUSEUP` command, automatically unlocking the screen and making the new current card visible.

The Search buttons that appear on the Personnel Card, Floor Plan, and Organization Chart screens utilize scripts identical to this one.

The final script associated with the Opening screen is attached to the Employee List list box.

The Employee List List-Box Script

The script attached to the Employee List list box is the most complex in the Who's Who application. Nevertheless, it is easy to follow once you understand its basic function.

The list of employees that appears on the Opening screen is a background field called Employee List. When you click on one of the names in the list, the script jumps to the card for that employee. If no personnel card exists for that employee, the script creates one automatically.

This enables you to create a personnel card for any employee simply by adding a name to the employee list (by typing the person's last name into the list box in Plus Card Edit mode) and then selecting that name from the list box. Of course, you'll also want to amend the Floor Plan and Organization Chart to reflect the employee's position on each, (and you'll have to keep the list in order manually, since there's no sort function for it). However, the ability to create personnel cards on the fly frees you from having to create a card for each employee before you can begin using the application.

Here's the script:

```
ON MOUSEDOWN
GLOBAL WhichCard
PUT CLICKLINE() INTO WhichCard
PUT THE SHORT NAME OF ME INTO WhichField
SELECT LINE WhichCard, WhichField, BG
```

```
PUT FIRST WORD OF THE SELECTION INTO WhichCard
IF WhichCard IS EMPTY THEN EXIT MOUSEDOWN
SET CURSOR TO 4
LOCK SCREEN
GO TO CARD WhichCard
IF THE RESULT IS NOT EMPTY THEN
GO TO CARD AAAC
DOMENU COPY CARD
DOMENU PASTE CARD
SET THE NAME OF THIS CARD TO WhichCard
PUT WhichCard INTO BG DBFIELD Name
SORT ASCENDING BY SHORT NAME OF CARD
GO TO CARD WhichCard
END IF
UNLOCK SCREEN WITH VISUAL EFFECT WIPE RIGHT
SET WINDOWTITLE TO WhichCard
END MOUSEDOWN
```

The script begins by establishing the global variable `WhichCard` and assigning to it the value returned by the `CLICKLINE` function (the line number of the item selected in the list box). Then it uses the command `PUT THE SHORT NAME OF ME` to assign the list box's name to the variable `WhichField`. (In a scheme that might have pleased T. S. Eliot—or at least one of his cats—every item in `Plus` has both a short name and a long name. The short name of this list box is `Employee List`. Its long name, which also identifies the card, background, and stack in which the list box is found, is `BKGND FIELD "Employee List" OF CARD ID 4199 OF BKGND ID 3758 OF STACK "WHONEW.STA"`.)

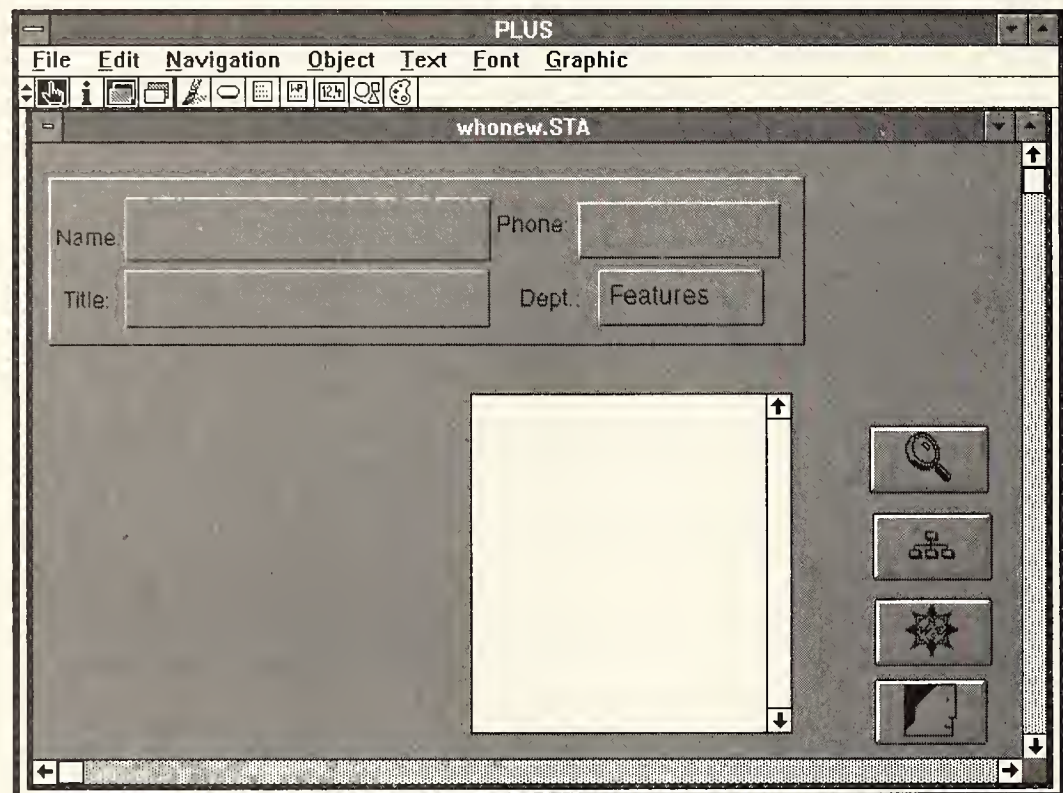
The next line, `SELECT LINE...`, is used to highlight the line the user clicked in the list box. It tells `Plus` to select the entire line indicated by the variable `WhichCard` in the field `WhichField` on the background of the current card.

Next the script assigns the first word of the highlighted line—in this case the last name of the selected employee—to the variable `WhichCard`, and then checks to see if `WhichCard` is empty, which it would be only if the user had clicked on an empty line. If so, the script exits the `MOUSEDOWN` function. Otherwise, it creates the busy cursor, locks the screen, and issues the command `GO TO CARD WhichCard`, instructing `Plus` to jump to the selected employee's personnel card.

Next, the script checks the value of the system variable `RESULT`. If `RESULT` is not empty (it contains an error message), then the script surmises that the card it just instructed `Plus` to go to does not exist. Consequently, it sets out to create the card, first by issuing the command to go to the `Personnel Card Template` (card `AAAC`). Once that card, shown in Figure 10.7,

has been made current, the script issues the command `DOMENU COPY CARD` to execute the Plus menu command `COPY CARD`, which copies the current card, followed by `DOMENU PASTE CARD`, which pastes the copy back into the stack, creating a new card identical to the Personnel Card Template card.

Figure 10.7
The Personnel Card
Template card



Next, the script uses the contents of the variable `WhichCard` (the selected employee's name) to name the new card, and assigns that same text to the field on the card labeled "Name", in which the employee's name is to be displayed.

Having created the new card, the script uses the Plus `SORT` command to sort the stack by card name, thus ensuring that the personnel cards remain in alphabetical order. Then it reissues the command `GO TO WhichCard`, to make the newly created card current (since the `SORT` command might have made another card current), thus ending the process of creating a new card.

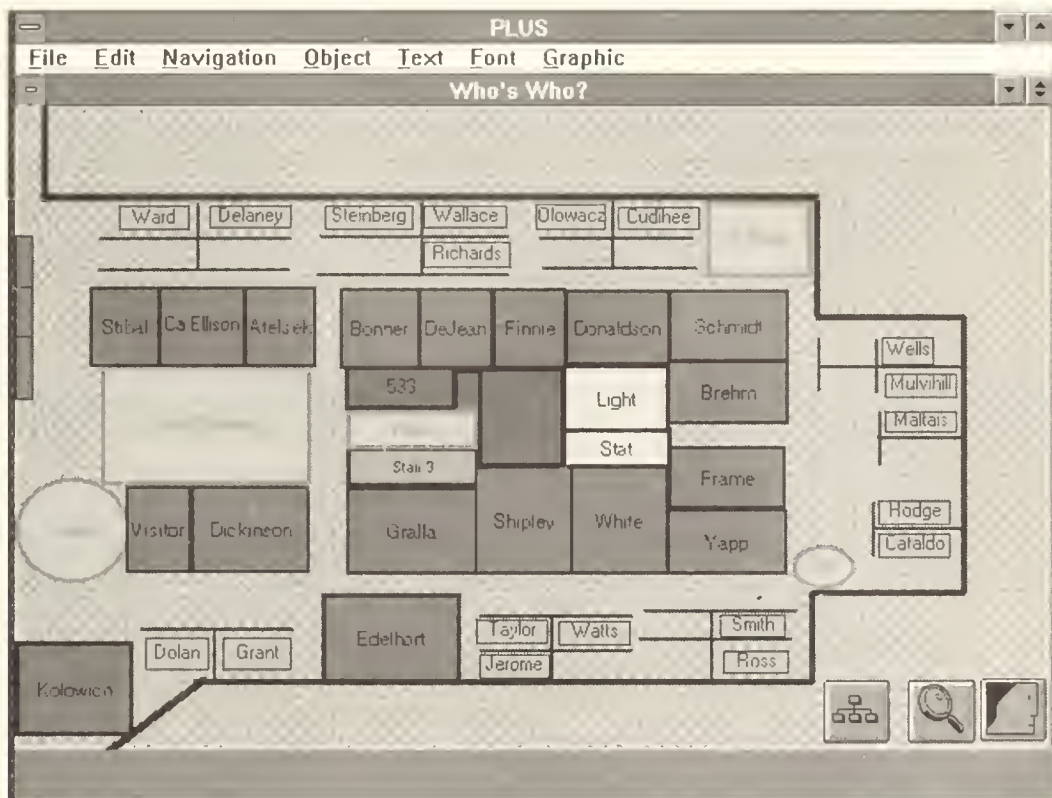
Finally the script unlocks the screen with the `WIPE RIGHT` visual effect, and sets the window title of the now-visible employee card to the employee's name, before ending the `MOUSEDOWN` procedure.

That concludes the scripts associated with the Opening screen of Who's Who. Next we'll examine the Floor Plan screen.

The Floor Plan Screen

The Floor Plan screen, shown in Figure 10.8, shows a map of PC/Computing's Medford offices.

Figure 10.8
The Floor Plan
screen



Aside from the three buttons at the lower-right corner of the screen, all the elements on the Floor Plan, including the building outline, cubicles, and offices, are individual draw objects.

I designed the Floor Plan screen to help the user identify the occupant of each office and as an aid to locating offices and resources. Hence, each office on the Floor Plan screen responds to the `MOUSEENTER` message by identifying its occupant's name and title on the Floor Plan screen's title bar. Conference rooms and other facilities on the Floor Plan likewise respond to the `MOUSEENTER` message by identifying their purpose. And as soon as the mouse is moved away from an object, the window title reverts to "Who's Who at PC/C?".

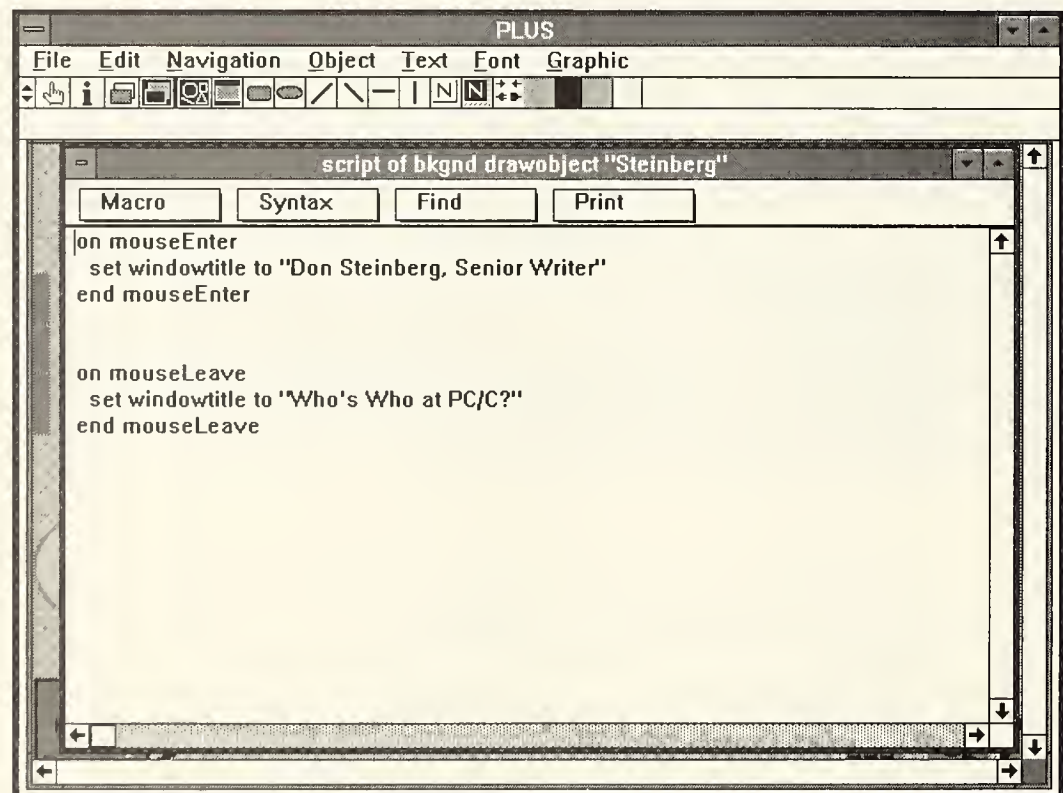
When you click on an office, it becomes highlighted on the screen, and then the script displays the occupant's personnel card. Again, as on the Opening screen, if no personnel card exists for that individual the script creates it automatically.

Although almost all the background draw objects on the map have a script attached to them, I actually only needed to create three completely unique draw objects from scratch: one for completely “inanimate” objects, such as cubicle and exterior walls; one for objects such as conference rooms, which can identify themselves on the title bar but are not linked to a personnel card; and one for offices and cubicles, which are linked to their occupant's personnel card.

To create additional offices and facilities, I merely copied the appropriate object and pasted it into position, and then amended its script. For instance, to create an additional office, I copied an existing office, pasted it into place, and then changed the name of the new object and its MOUSE-ENTER script to reflect the name of the occupant of that office. Figure 10.9 shows the script-editing window used to make these changes.

Figure 10.9

Editing a script in Plus



The Floor Plan Background Scripts

The code that actually links offices and cubicle occupants to the personnel cards resides not within the individual draw objects that represent those locations, but rather in the ON MOUSEUP handler for the Floor Plan's background.


```
ON MOUSEUP
SET CURSOR TO 4
LOCK SCREEN
PUT THE SHORT NAME OF THE TARGET INTO ChosenOne
IF ChosenOne IS EMPTY THEN EXIT MOUSEUP
GO TO CARD ChosenOne
IF THE RESULT IS NOT EMPTY THEN
GO TO CARD AAAC
DOMENU COPY CARD
DOMENU PASTE CARD
SET THE NAME OF THIS CARD TO ChosenOne
PUT ChosenOne INTO BG DBFIELD NAME
SORT ASCENDING BY SHORT NAME OF CARD
GO TO CARD ChosenOne
END IF
UNLOCK SCREEN WITH VISUAL EFFECT ZOOM OPEN
SET WINDOWTITLE TO ChosenOne
END MOUSEUP
```

This script probably looks familiar, if you remember the script for the Employee List list box on the Opening screen, because it performs pretty much the same actions in pretty much the same way.

The primary difference between this and the Employee List script is that this script makes use of the system variable TARGET, which Plus uses to identify the object that received the message the script is handling. Thus, if you released the mouse button over my office, the short name of TARGET would be Bonner, even though the current script is attached to the background rather than to the object Bonner. (The background MOUSEUP script is performed because the office buttons don't have a MOUSEUP procedure.)

The script assigns the short name of TARGET to the variable ChosenOne, and then issues the command GO TO CARD ChosenOne to activate the selected individual's personnel card. If that card does not exist, the script creates it, using the same method as the Employee List procedure on the Opening screen.

Keeping Track of Highlighting

In addition to the MOUSEUP script, the Floor Plan background has a MOUSEDOWN script, used to ensure that only the office the user has clicked on is highlighted.

This is necessary because when you select the Floor Plan button from a personnel card, that button's script creates a variable called High, stores the short name of the current personnel card there, and then turns highlighting on for the corresponding office or cubicle on the Floor Plan (so that the color of the office or cubicle is reversed). I wanted that highlighting to turn off as

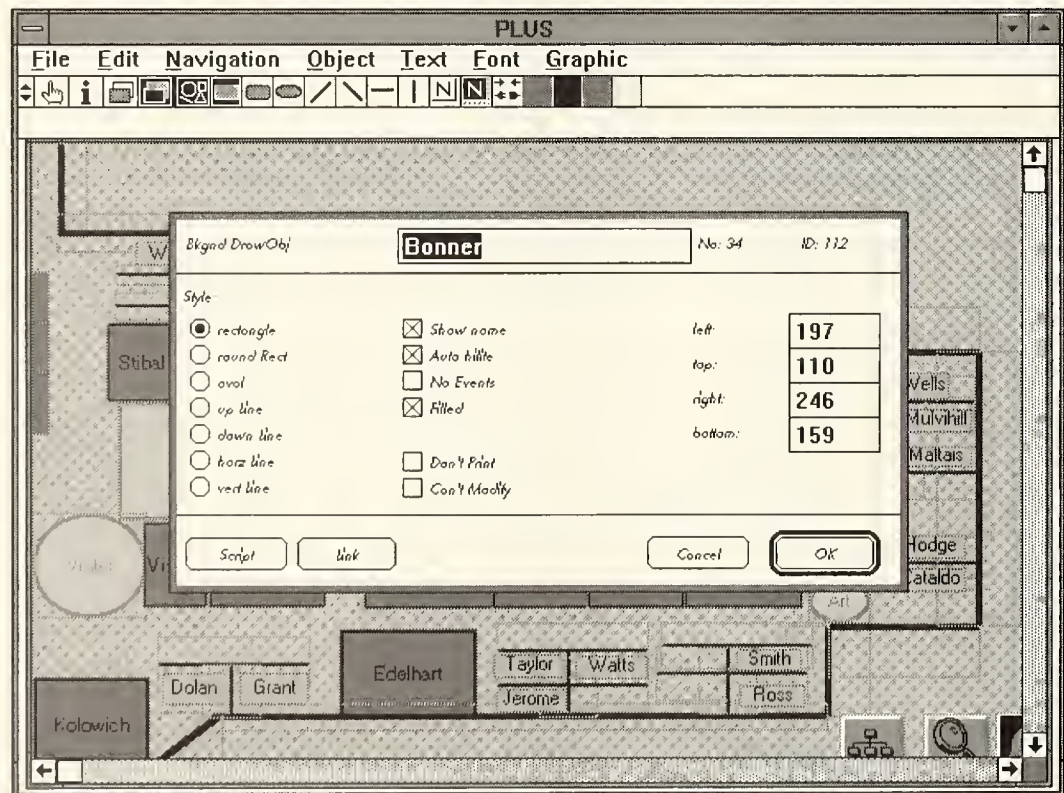
soon as the user begins to click on another office or cubicle, so I included the following MOUSEDOWN handler on the Floor Plan's background:

```
ON MOUSEDOWN
GLOBAL High
IF High IS NOT EMPTY THEN SET HIGHLIGHT OF BG DRAWOBJECT High TO FALSE
END MOUSEDOWN
```

Thus, if High contains the name of a draw object (that is, High is not empty), the script turns off highlighting for that object. Meanwhile, because I had set the AUTOHIGHLIGHT property to TRUE for office and cubicle objects using the Object Properties dialog box, shown in Figure 10.10, highlighting will be turned on automatically for whatever office the user has selected with the mouse once the mouse button is released.

Figure 10.10

The Object Properties dialog box



Now let's look at the scripts for office and cubicle objects.

The Office Object Scripts

Each office and cubicle object has two simple scripts: one that changes the Floor Plan's window title to identify the occupant of the office or cubicle whenever the mouse pointer is over that object, and one that changes the

window title back to "Who's Who at PC/C?" once the mouse pointer leaves that object. The first script looks like this:

```
ON MOUSEENTER
SET WINDOWTITLE TO "Paul Bonner, Senior Editor"
END MOUSEENTER
```

This script must be modified for each office to reflect the name and title of its occupant.

The second script is identical for each office and cubicle:

```
ON MOUSELEAVE
SET WINDOWTITLE TO "Who's Who at PC/C?"
END MOUSELEAVE
```

Facilities Scripts

The scripts for conference rooms and other facilities that identify themselves on the title bar are similar to those for offices and cubicles. In fact, their MOUSELEAVE handlers are identical to that listed above, and their MOUSEENTER handlers differ only in that they identify the purpose of the facility, rather than an individual's name and title, as follows:

```
ON MOUSEENTER
SET WINDOWTITLE TO "Conference/Training Room"
END MOUSEENTER
```

In addition, however, these objects have empty scripts for handling the MOUSEUP and MOUSEDOWN messages:

```
ON MOUSEUP
END MOUSEUP

ON MOUSEDOWN
END MOUSEDOWN
```

These empty handlers intercept the MOUSEUP and MOUSEDOWN messages for these objects and discard them, preventing them from ever reaching the background-level scripts for those events. Thus, clicking the mouse on a conference room does not change the current highlighting or make Plus attempt to jump to a personnel card for that room.

Inanimate-Object Event Handling

I didn't want Plus to respond at all to mouse events involving inanimate objects, such as the outside walls on the Floor Plan, so I used the Object Properties dialog box to set the No Events property for these objects to True. Doing so tells Plus to ignore any events involving these objects.

Floor Plan Screen Buttons

In addition to all the draw objects that make up the office map, the Floor Plan screen includes three standard buttons: Search, Org Chart, and First. As previously noted, the Search and Org Chart buttons are identical to those on the Opening screen card, except for the visual effects used with the Org Chart button's UNLOCK SCREEN command (the Floor Plan version of this script uses the ZOOM OPEN effect).

The button called First, which has the icon of a face in profile, is designed to take you back to the Opening screen of the application. Its script looks like this:

```
ON MOUSEUP
SET CURSOR TO 4
LOCK SCREEN
GO TO FIRST
UNLOCK SCREEN WITH VISUAL EFFECT SCROLL LEFT
SET WINDOWTITLE TO "Who's Who?"
END MOUSEUP
```

This script simply turns on the busy cursor, locks the screen, and then activates the Opening screen (the first card in the stack, hence the keyword FIRST), unlocks it with a SCROLL LEFT effect, and sets the window title to "Who's Who?"

That's it for the Floor Plan screen. Now let's look at the Personnel Card screen.

The Personnel Card Screen

Each personnel card contains four background database fields (one each for the employee's name, title, telephone number, and department), plus a scrolling text field that can hold extended biographical data, and a paint object that can display a scanned photograph of the employee. In addition, each personnel card has four buttons, one to launch the search function and three others to jump to the Organization Chart, Opening screen, and Floor Plan. Figure 10.11 shows a typical personnel card.

As discussed above, the script creates new personnel cards automatically as it needs them, by copying the Personnel Card template and pasting a new card into the stack. These newly created cards are more or less blank. The name field holds the last name of the employee, and the department field holds its default value, Features, as shown in Figure 10.12.

To fill in a newly created card, you would simply type the necessary data into the Name, Title, and Phone database fields, and the biographical information field. The Dept. (department) field, however, won't allow you to type data into it directly. Instead, it reacts to any keypress by opening a pop-up menu on the screen, from which you can select the employee's department, as shown in Figure 10.13.

Figure 10.11

A Who's Who personnel information card

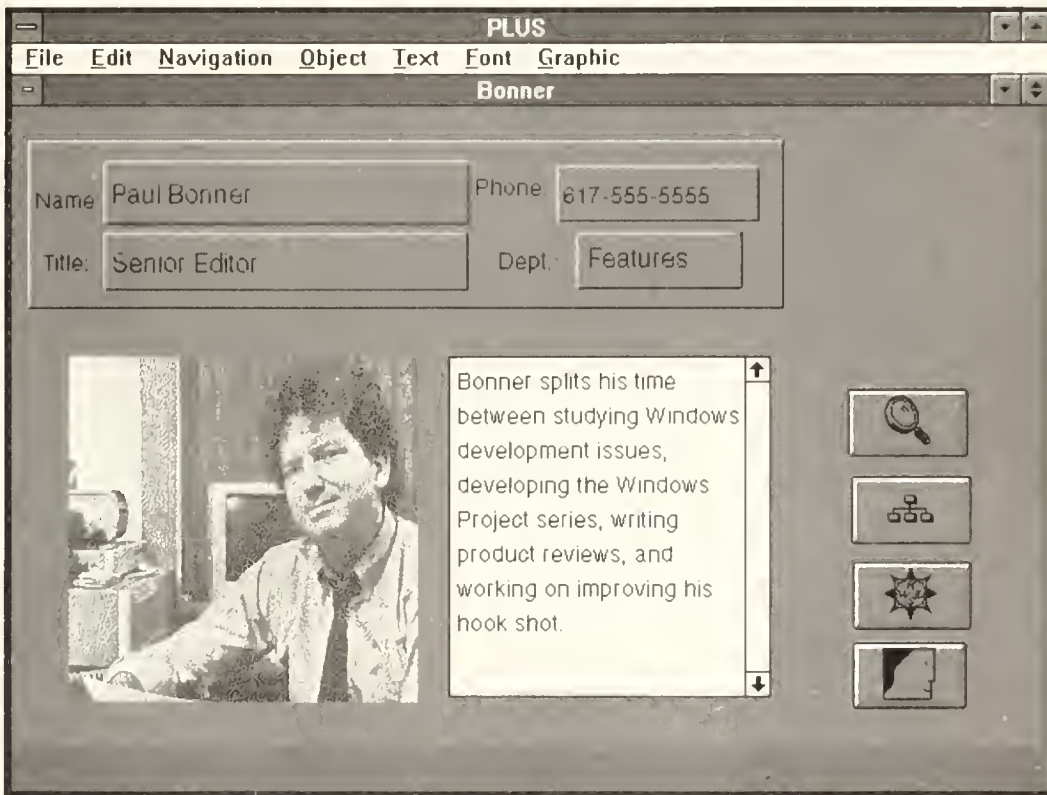


Figure 10.12

A newly created personnel card

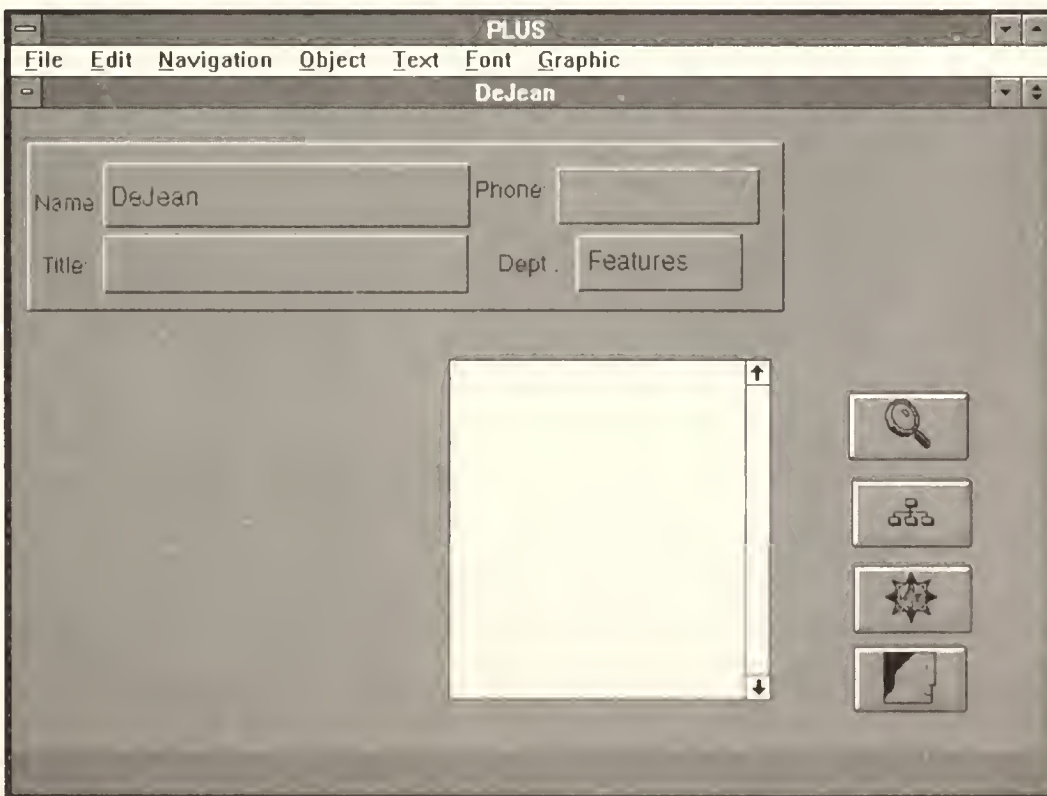
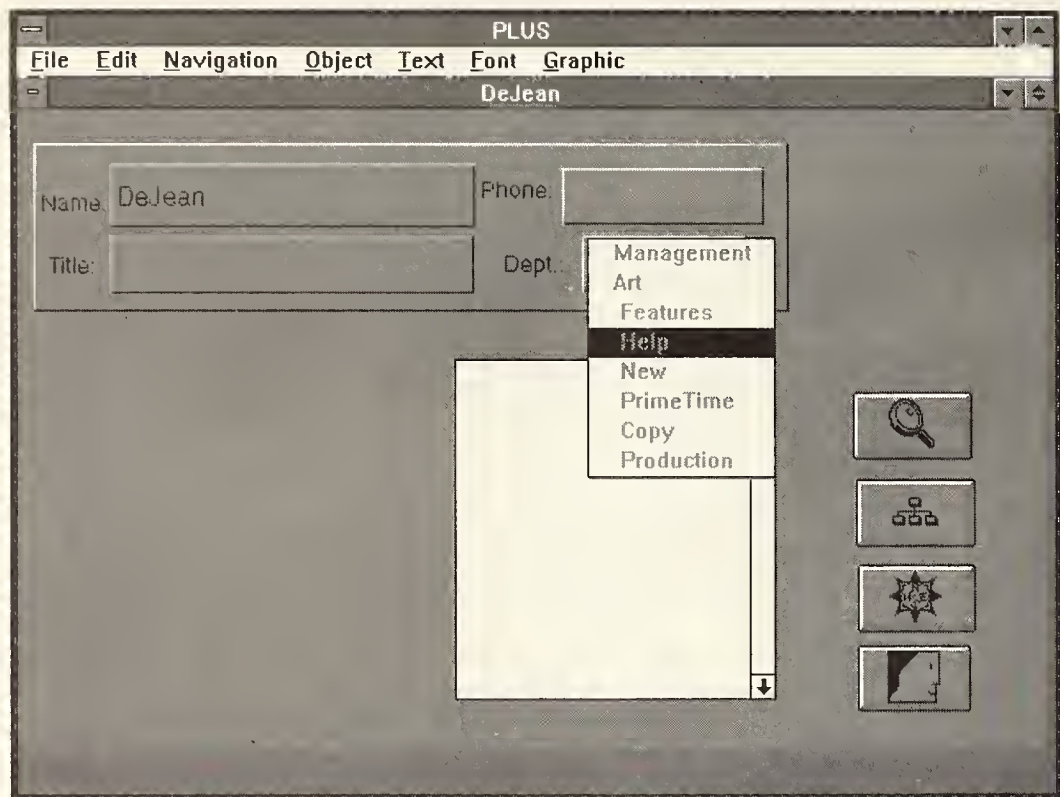


Figure 10.13

The pop-up
Departments menu



Providing a pop-up list such as this allows you to ensure that all entries for a given department will be the same—which simplifies processes such as searching for people by department. Otherwise, one Features writer's department might be labeled "Features", another "Feat. Dept.", and another "Feat".

The paint object field for the scanned photograph is also empty when Plus creates the card. To add a scanned photo to the field, you can either select the Import Picture option on the Object Information dialog box, or simply select the card in background editing mode and paste a graphic that you have previously captured to the Clipboard into it. (I found the second method the more reliable of the two. I used a hand scanner to create a 16-color scanned image of each employee photograph, and then used the Windows Clipboard to paste it onto the card.)

The Department Field Script

The Personnel Card screen has only two unique scripts: the script for its Floor Plan button and the script for the department field. Let's look at the department field script first:

```
ON KEYPRESSED
  PUT " Management; Art; Features; Help; New; Prime Time;
    Copy; Production" INTO DepartmentList
```



```

PUT THE TOPLEFT OF ME INTO ThePos
PUT 1 INTO ItemNumber
PUT POPUPMUSELECT (ThePos, DepartmentList, ItemNumber)
  INTO N
IF N=0 THEN EXIT KEYPRESSED
IF N=1 THEN PUT "Management" INTO ME
IF N=2 THEN PUT "Art" INTO ME
IF N=3 THEN PUT "Features" INTO ME
IF N=4 THEN PUT "Help" INTO ME
IF N=5 THEN PUT "New" INTO ME
IF N=6 THEN PUT "Prime Time" INTO ME
IF N=7 THEN PUT "Copy" INTO ME
IF N=8 THEN PUT "Production" INTO ME
END KEYPRESSED

```

This script, which is launched in response to any keypress in the Departments field, begins by creating a list of all PC/Computing's editorial departments, which it calls `DepartmentList`. Then it assigns the top-left screen coordinates of the department field (`THE TOPLEFT OF ME`) to the variable `ThePos`, and the number 1 to the variable `ItemNumber`.

Next, the script opens a pop-up menu at the coordinates indicated by `ThePos`, using the list of items in `DepartmentList` as menu items, with the first item in the list (`ItemNumber`) highlighted. It then waits for the user to select a menu item. Once a selection has been made, the number of the selected item is assigned to the variable `N`.

Finally, the script fills the department field with a string corresponding to the user's choice. If `N = 1`, for instance, indicating that the user selected the first item on the list, the script places "Management" into the department field.

This segment of code demonstrates both how unconventional and how intuitive the Plus language can be. "PUT THE TOPLEFT OF ME INTO `ThePos`" looks like gibberish to anyone used to more conventional languages, but it actually is pretty simple to understand, if you don't mind its self-conscious cuteness.

The Floor Plan Button Script

The only other unique script on the Personnel Card screen is the `MOUSEUP` handler for the Floor Plan button, which not only jumps to the Floor Plan screen but highlights the office of the person whose personnel card you last worked with.

```

ON MOUSEUP
SET CURSOR TO 4
GLOBAL High
GET THE SHORT NAME OF THE CARD
PUT IT INTO HIGH

```

```

LOCK SCREEN
GO TO CARD AAAB
UNLOCK SCREEN WITH VISUAL EFFECT SCROLL RIGHT
SET WINDOWTITLE TO "Who's Who?"
IF High IS NOT EMPTY THEN SET HILITE OF BG DRAWOBJECT High TO TRUE
END MOUSEUP

```

The MOUSEUP handler sets the busy cursor, then creates the global variable High. Next it obtains the short name of the current card and assigns it to the variable High. Then it locks the screen, jumps to the Floor Plan screen (card AAAB), unlocks the screen with a scroll-right effect, and sets the window title of the Floor Plan screen to “Who’s Who?” Finally it examines the variable High, and if it is not empty (which it would be only if the user pressed the Floor Plan button while viewing the empty Personnel Template card) turns on highlighting for the draw object with the same name.

That brings us to the application’s final screen, the Organization Chart.

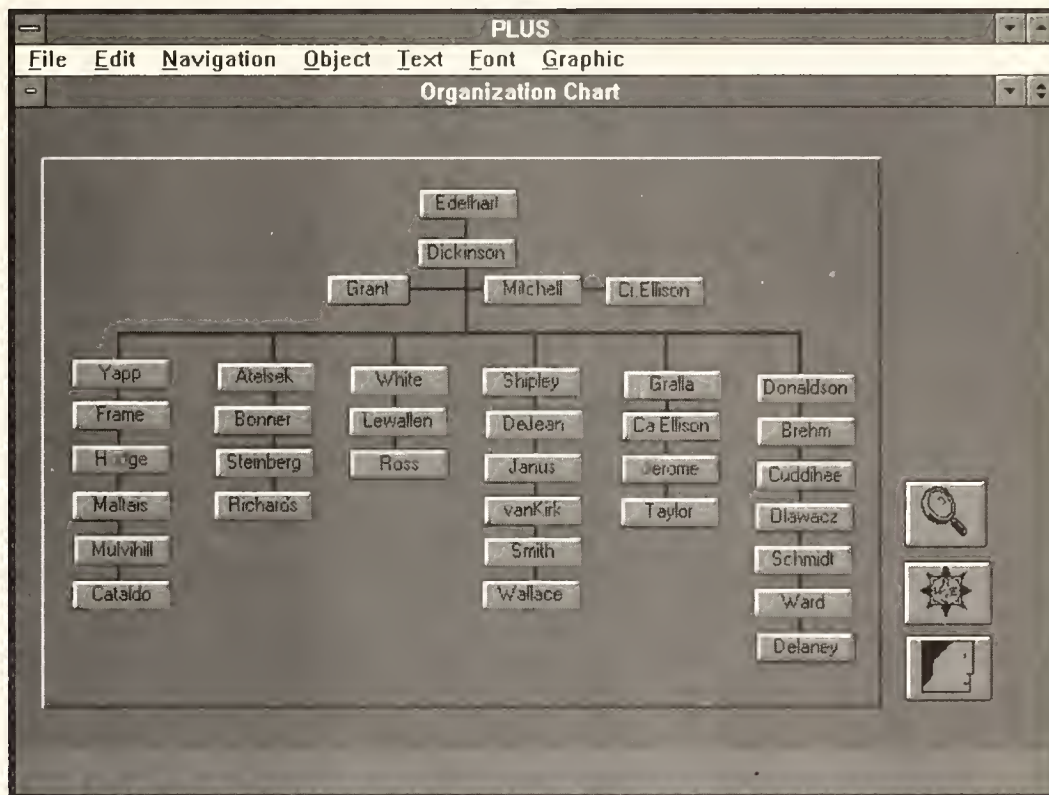
The Organization Chart Screen

Although the Organization Chart screen is visually complex, its operation and scripts are very simple. The screen displays a hierarchical chart of the editorial organization at PC/Computing (circa January, 1991), with each individual represented by a separate pushbutton, as shown in Figure 10.14. When you press a button, Plus jumps to that individual’s personnel card.

In addition, the Organization Chart screen contains buttons for jumping to the Floor Plan and Opening screens, and a button to activate the Search routine. The scripts for the Search and Floor Plan buttons are identical (except for the visual effect on the Floor Plan button’s screen-unlock sequence) to those for the same buttons on the Opening screen. And the Opening Screen button’s script is identical (again, except for the unlock sequence) to that of the corresponding button on the Floor Plan.

I painted the hierarchical organization lines directly onto the Organization Chart’s background using Plus’s paint tools, so they exist only as bit-maps, not as objects.

The buttons representing each member of the editorial staff are identical except for their names. The script that powers those buttons is located at the background level, rather than being contained in each button. I could have attached it to each button’s MOUSEUP procedure, but that would have meant that the script code would be physically reproduced once for every button in the stack’s file—swelling the file size and increasing the amount of time it takes Plus to load the screen. So instead, I placed the script at the background level, where it applies to every personnel button on the organization chart.

Figure 10.14The Organization
Chart screen**The Organization Chart Script**

The Organization Chart script reacts to the MOUSEUP event, and uses the TARGET variable to obtain the name of the button the user pressed:

```

ON MOUSEUP
SET CURSOR TO 4
LOCK SCREEN
PUT THE SHORT NAME OF THE TARGET INTO ChosenOne
IF ChosenOne IS "ZZZ_Org_Chart" THEN
UNLOCK SCREEN
SET WINDOWTITLE TO "Organization Chart"
EXIT MOUSEUP
END IF
GO TO CARD ChosenOne
IF THE RESULT IS NOT EMPTY THEN LOCK SCREEN
GO TO CARD AAAC
DOMENU COPY CARD
DOMENU PASTE CARD
SET THE NAME OF THIS CARD TO ChosenOne
PUT ChosenOne INTO BG DBFIELD Name

```



```
    SORT ASCENDING BY SHORT NAME OF CARD
    GO TO CARD ChosenOne
  END IF
  UNLOCK SCREEN WITH VISUAL EFFECT ZOOM OPEN
  SET WINDOWTITLE TO ChosenOne
  END MOUSEUP
```

By now this probably looks pretty familiar, since it closely resembles the scripts that get you to the personnel cards from the Opening screen or the Floor Plan.

The script uses `TARGET` to obtain the user's selection, and assigns that to the variable `ChosenOne`. Then it determines if `ChosenOne` is equal to `"ZZZ_Org_Chart"`, which happens to be the name of the Organization Chart. (`ChosenOne` would have that value if the user had clicked on the background, rather than on a button.) If so, the script unlocks the screen, restores the window title (which Plus forgets when it encounters a `LOCK` command) and exits the `MOUSEUP` procedure.

Otherwise, the script issues the command `GO TO CARD ChosenOne`. If no such card exists, the script creates it, using the now familiar `COPY CARD, PASTE CARD` sequence to duplicate the Personnel Template card. That concludes discussion of the Organization Chart screen.

Wrapping Up the Who's Who Application

The Who's Who application demonstrates just how easily you can build a data-presentation application in Windows, given the right development tools.

This application obviously doesn't perform any heavy-duty data-analysis functions, but it provides rapid access to a variety of data in forms that would be difficult to present with a non-Windows development tool. Moreover, once you've become proficient with Plus, it would be easy to modify the application to have it gather and maintain a richer set of data. In the meantime, Who's Who at PC/Computing was simple to build, and is enjoyable to use—a winning combination any way you look at it.

C H A P T E R

11

**Automating
Existing
Applications—
AutoPrint
for Windows**

*Designing the
Application*

How AutoPrint Works

*The GETFILE.WBT
Batch File*

*The AUTOPRN.WBT
Batch File*

*Putting the Batch Files
to Work*

The WinBatch Version

*The COPYMAC.WBT
Batch File*

*The AUTOP.WBT
Batch File*

SOMETIMES YOU CAN GET AN AWFUL LOT OF MILEAGE OUT OF A FEW lines of programming code. Take, for instance, AutoPrint for Windows, a batch printing utility designed to automate the process of printing files created by Windows applications. It allows you to specify files for unattended printing, eliminating the need to manually issue the File Open and Print commands for each file you wish to print. It also allows you to specify a time at which the print utility will run each day, so that time-consuming print jobs can be performed overnight or at another time that will not interfere with your PC use. It is powerful enough to print any number of files, created by nearly any Windows program, at any time you specify, and yet requires only a few dozen lines of code.

Printing a document created by a Windows application is tricky business. You need to know all about fonts, bit-mapped graphics, and complex page formatting, and you have to translate that knowledge to your program. That's the kind of task that requires thousands of lines of program code, not dozens, and a knowledge of Windows' internal workings well beyond the reach of most users.

The key to AutoPrint's efficiency is that it doesn't actually print anything itself. Instead it sidesteps those complex formatting issues by having the program that created a document do the actual printing. So, rather than getting involved in the messy job of printer control, AutoPrint simply controls other Windows applications, taking advantage of their ability to print the documents they create. In short, when you run the AutoPrint utility, it automatically identifies the application that created each document you want to print, launches that application, and tells it to print the document.

By virtue of its ability to use the printing facilities of other applications in this way, AutoPrint for Windows is an embodiment of the concept of cut-and-paste programming discussed in Chapter 3. Even though it consists of only a few simple lines of batch language code, it performs a series of highly complex tasks by taking advantage of the capabilities of other applications. The same approach can be used for a wide variety of utility programming tasks in Windows.

The Impetus behind AutoPrint

AutoPrint was born out of my frustration with the way that printing several complex documents under Windows can tie up a computer. Try to print a half-dozen large PageMaker documents, or large documents created in any other Windows program that contain lots of different fonts and large bit-mapped graphics, and you might end up sitting in front of your computer for hours. And you spend most of that time just waiting for your application to send the output from one document to the Windows Print Manager or to your printer before you can give it the commands to load and print the next document. The ridiculous thing about it is that the process is essentially

automatic—as long as your printer doesn't run out of paper all you basically have to do is issue the print command. But despite that, you can't do anything else with your PC, or at least with the application doing the printing, until the print job is complete.

The obvious solution was to build a print queue for Windows. By this I don't mean a replacement for Windows' Print Manager, which saves the data being sent from an application to your printer while a document is being printed, and feeds that information out only as fast as the printer can handle it. Although Print Manager speeds the printing process a little by enabling your application to operate faster than your printer can accept data, the process can still be time consuming because of the time it takes for Windows applications to format documents for printing.

Rather, I wanted AutoPrint to initiate and manage the slow, dull process of printing multiple documents, so that it could take place after I'd gone home for the night, or while I was at lunch or a meeting, not while I sat at the keyboard. So I designed AutoPrint to act as a print queue that would maintain a list of files designated for later printing, and then print them at a more convenient time.

Designing the Application

The requirements for AutoPrint were so simple that it could have been built using nearly any Windows development tool. All the utility really had to do was provide a way for the user to designate the documents to be printed; then it had to interact with other Windows applications to make them print those documents.

One approach would have been to use a tool with which I could design an attractive user interface for the portion of the application used to designate the files the utility should print. In that case, the application would probably have looked something like a modified version of the File Open dialog box provided by Windows 3.1, with drive, directory, and file list boxes that the user could manipulate until the file to be printed was highlighted.

It wouldn't have been difficult to implement the utility in that fashion using a tool such as Turbo Pascal for Windows, Visual BASIC, or Realizer, but it nevertheless seemed unnecessarily complex. What appeared more reasonable was to try to make the process of selecting files for later printing an extension of the standard file management tools I was already using for things like moving and deleting files. That way, I hoped, I would be able to highlight the name of a document in a Windows 3.1 File Manager window, press a hotkey, and instantly have the document added to the AutoPrint list.

To accomplish that, I wanted a tool that put more emphasis on interacting with other applications than on building custom applications from

scratch. Specifically, I needed a Windows batch language, such as WinBatch, Bridge Batch, or Batchworks. The question was, which one?

The Right Tool for the Job

Following the precepts of cut-and-paste programming, I opted to build the initial version of AutoPrint for Windows using Batch Runner, the version of WinBatch that is built into Norton Desktop for Windows. Doing so allowed me to take advantage of Norton Desktop's Scheduler utility to designate a daily time at which all the files in the AutoPrint queue would be printed. It also simplified the process of actually initiating the printing of each document, because, unlike the standard Windows 3.1 File Manager, Norton Desktop has a "smart" print command.

In the Windows 3.1 File Manager, when you highlight a plain ASCII text file (such as one created by the Notepad accessory) and then issue the File Print command, the file will be printed. But if you highlight any other kind of file and issue the File Print command, File Manager merely prints the name of the file.

In contrast, whenever you highlight a file in a Norton Desktop drive window file pane and issue the File Print command, Norton Desktop uses the file association recorded in the Extensions section of WIN.INI to launch the program associated with that file, and then instructs it to print the file.

Thus, using the Norton Desktop batch language eliminated the need to write either scheduling code or the code necessary to launch applications and issue the Print File command.

Once the initial, Norton Desktop version of AutoPrint was complete, however, I expanded the code using the standard version of WinBatch and Windows File Manager to provide the same scheduling and printing functions outside of Norton Desktop. (See the section titled "The WinBatch Version".)

How AutoPrint Works

AutoPrint for Windows is used as follows: Throughout the day you can designate any document file for printing, simply by selecting its name—either in a File Manager window or in the file pane of a Norton Desktop for Windows drive window—and pressing Ctrl-P. Doing so will launch a short batch program called COPYMAC.WBT (or GETFILE.WBT if you're working with Norton Desktop), which adds the file to the list of those to be printed. When the actual AutoPrint batch file is run later, it directs the printing of each file on the list.

The key to AutoPrint's ability to interact with other applications this way is found in the WIN.INI file in your Windows directory. This file, used to store a variety of parameters and option settings for Windows and Windows

applications, contains a section called Extensions, which Windows uses to identify the program associated with a given file extension. This information is used, for example, when you double-click on a document in File Manager. What you're doing is telling Windows to run that document—but documents are not executable files. So Windows looks in the Extensions section of WIN.INI to find out which executable program the file is associated with, that is, what application should be used to “run” documents that have the extension of the one you selected.

For instance, some of the standard entries in the Extensions section of WIN.INI look like this:

```
txt=notepad.exe ^.txt
pcx=pbrush.exe ^.pcx
wri=write.exe ^.wri
```

These entries tell Windows that Notepad is associated with files that have a .TXT extension, Paintbrush (pbrush.exe) is associated with .PCX files, and Write is associated with files that have a .WRI extension. So when you double-click on a document that has a .TXT extension, Windows loads it into Notepad.

Many Windows applications add additional entries to the Extension section of WIN.INI when you install them on your system. For instance, Microsoft Word for Windows adds these lines to the Extensions section:

```
doc=winword.exe ^.doc
dot=winword.exe ^.dot
```

These lines tell Windows to associate files with a .DOC (document) or .DOT (document template) extension with WINWORD.EXE, Word for Windows' executable file.

So, as long as your WIN.INI file contains an entry in its Extensions section that tells Windows which executable program to use with every type of file in your AutoPrint list, AutoPrint won't have any trouble identifying the program to be used in printing each document.

Of course, identifying the program that created a document, successfully launching that program, and instructing it to print the document are entirely different things. However, although AutoPrint is not infallible, it will always be able to complete the first two steps as long as the program it is attempting to launch is located in a directory or subdirectory listed on your DOS path.

Once the application is launched, AutoPrint can almost always convince it to print the designated file, thanks to the widespread standardization of the printing process in Windows applications. In virtually all Windows applications, you can print a document by pressing first Alt-F (to open the File menu), then P (for Print), and then Enter once or twice to close a print

options dialog box or two. AutoPrint takes advantage of that standardization to launch the printing process.

This, by the way, is a good example of why it makes sense for developers of Windows applications to follow the standard Windows application design guidelines whenever possible. Not everybody necessarily loves the File-Print method for printing documents. Perhaps some developers would prefer to use a menu named Output, with options such as Serial Printer, LPT1, and LPT2, in which case the print command might be something like Alt-O S, or Alt-O 1. But you can see how that kind of individualization of a standard process, such as printing, would make it much more difficult to build a utility such as AutoPrint (not to mention that it would be one more thing users would have to learn before they could use the application effectively).

The GETFILE.WBT Batch File

The Norton Desktop version of the AutoPrint utility actually consists of two batch files, GETFILE.WBT and AUTOPRN.WBT. The former is used to add files to the AutoPrint queue, whereas the latter prints the files in the queue.

Norton Desktop for Windows includes a feature known as the Launch List, which takes the form of a custom menu of applications that is added to the Control menu of all your Windows applications. (The Control menu is the one that appears when you click on the gray box in the upper-left corner of a Windows application's main window.) A small utility called the Launch Manager allows you to assign new items to the Launch List menu and to designate hotkeys that can be used to activate those items. I used this utility to assign the hotkey Ctrl-P to the GETFILE batch program, so that it would run whenever the user pressed that key combination.

GETFILE.WBT is designed to be used in conjunction with Norton Desktop's File Manager-like drive windows. To add a file to the AutoPrint queue, you simply highlight the file in the drive window's file pane and then press Ctrl-P to launch the GETFILE batch program.

GETFILE.WBT has a very simple job. It simply has to identify the file selected in the drive window file pane and add it to the AutoPrint queue, after checking to ensure that an association exists for files with its extension.

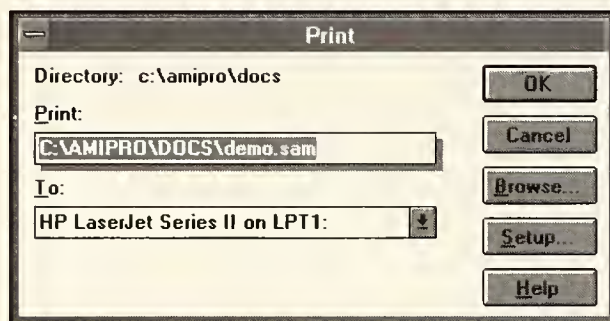
The only difficult part of that process is identifying the selected file. In order to print it later, the AUTOPRN batch file needs the file's full path name, and, unfortunately, it can't actually read the screen. So there had to be some way to determine which file was selected and its full path.

Even that was fairly simple, however. When you issue the File Print command in Norton Desktop, a dialog box entitled "Print" opens. The Print dialog box includes an editable text box listing the name and full path of the file currently selected in the file pane, as shown in Figure 11.1. So GETFILE

issues the File Print command, copies the contents of that text box and adds them to the AutoPrint queue list, and then cancels the File Print dialog box.

Figure 11.1

Norton Desktop's
Print dialog box



GETFILE Dissected

GETFILE.WBT consists of four simple routines, called Start, Main, Wrong-App, and AssocErrJump.

The Start Routine

The Start routine simply initializes three string variables: AutoPrnDir (the directory that holds the Norton Batch Runner program, used to execute batch files), AutoPrnList (which contains the name of the AutoPrint queue list), and TempFile (which holds the name of a temporary file used by the GETFILE routine).

```
:Start
AutoPrnDir=DIRHOME()
AutoPrnList=STRCAT(AutoPrnDir, "AUTOPRN.LST")
TempFile=STRCAT(AutoPrnDir, "TEMPLIST.LST")
```

The DIRHOME function in the first line of the routine returns the name of the directory in which Batch Runner resides. Then the second and third lines use the STRCAT command to concatenate that name with the file names AUTOPRN.LST and TEMPLIST.LST, to define the variables AutoPrnList and TempFile. Thus, if Batch Runner resides in the directory C:\NDW\, the variable AutoPrnList will equal C:\NDW\AUTOPRN.LST after the execution of the second line in the routine.

The Main Routine

The next routine, Main, is the workhorse of the GETFILE.WBT batch file. This routine begins by making sure that the cursor is in a Norton Desktop drive window file pane and that a single file has been selected. It then copies the name of that file to a variable called FileToPrint. Next, it checks that an association has been defined for files with the extension of the selected file.

Then it adds the file's full name and path to the list of files to be printed. And finally, it unloads itself from memory.

Here's how the routine begins:

```
:Main
SENDKEY("!FP")
IF WINGETACTIVE() <>"Print" THEN GOTO WrongApp
```

The first line uses SENDKEY to issue the File Print command by sending the keystrokes Alt-F P to Norton Desktop. (Alt is represented in a SENDKEY command by an exclamation point character (!); similarly, Ctrl is represented by a caret (^) and Shift by a plus sign (+).)

After the first line of the Main routine sends the File Print command, the second uses the batch language's WINGETACTIVE title to obtain the title of the current active window or dialog box. If the cursor is in the file pane of a Norton Desktop drive window, sending the File Print command should open a dialog box labeled "Print". If the active window isn't entitled Print, then the cursor must not have been in a Norton Desktop drive window file pane, so execution jumps to the error routine called WrongApp (described below).

Of course, all this test does is ensure that some application opened a dialog box labeled "Print" in response to the SENDKEY command. So the routine continues by checking to ensure that a file name is highlighted in the Print dialog box, as follows:

```
CLIPPUT( "")
SENDKEY("^{|INSERT}")
SENDKEY("{ESCAPE}")
FileToPrint=CLIPGET()
IF FileToPrint="" THEN GOTO WrongApp
Period=STRSCAN(FileToPrint,".",1,@FWDSCAN)
IF Period==0 THEN GOTO WrongApp
```

First the Windows Clipboard is emptied by placing an empty string onto it using the CLIPPUT command. Then SENDKEY is used to send the standard Windows shortcut for the Copy command (Control-Insert) to copy the contents of the edit box that displays the document's full name and path.

Next, the routine sends an Escape character to cancel the dialog box, and then assigns the contents of the Clipboard to the variable FileToPrint. The script then examines the contents of FileToPrint to ensure that the variable contains a valid file name. It starts by checking to make sure that FileToPrint isn't an empty string. If it is empty, that means there was no file name to copy in the dialog box's edit field, so it jumps to the WrongApp error routine. (One way this error could occur is if more than one file name is highlighted at the time you press Ctrl-P.)

Next, the routine scans the string to detect the presence of a period, such as that used to separate a file's name and its extension. If the string doesn't contain a period, then either a directory name (rather than a file name) was highlighted, or the cursor wasn't in a Norton window file pane at the time the routine was activated. In any case, without a file name to process, the routine cannot continue.

If a file name was obtained from the Print dialog box, the routine goes on like this:

```
Ext=FILEEXTENSION(FileToPrint)
AssocExists=INIREAD("Extensions",Ext,@FALSE)
If AssocExists==@FALSE THEN GOTO AssocErrJump
```

It uses the FILEEXTENSION function to obtain the file's extension. Then it uses the INIREAD command to determine if an association exists for files with that extension. The INIREAD command automatically searches the section of WIN.INI specified by the first item in its parameter list. If it finds the second item specified in the parameter list there, it returns the value @TRUE. Otherwise, it returns @FALSE.

If the specified document's extension is not found in the Extensions list, the batch program jumps to a second error routine, AssocErrJump (see below).

If the specified file's extension is found in the Extensions section of WIN.INI, GETFILE knows that AutoPrint will be able to print the file later. So it adds the file's complete name and path to the queue list with these commands:

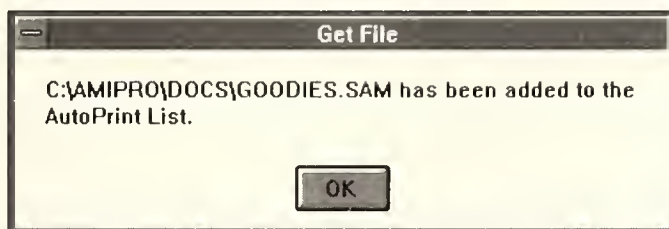
```
FHandle=FILEOPEN(TempFile,"WRITE")
FILEWRITE(FHandle,FileToPrint)
FILECLOSE(FHandle)
FILEAPPEND(TempFile,AutoPrnList)
FILEDELETE(TempFile)
FileUpper=STRUPPER(FileToPrint)
MESSAGE("Get File", "%FileUpper% has been added to the
AutoPrint queue.")
EXIT
```

The first command in the preceding block opens a temporary file, using the name assigned earlier to the variable TempFile. Next, it writes the name of the selected file (FileToPrint) to that file and then closes the file. Then it appends that temporary file to the end of the file identified by the variable AutoPrnList (creating the latter file if it does not already exist). It then deletes the temporary file, converts the name of the file to be printed to uppercase, and displays a message saying that the file has been added to the queue, as shown in Figure 11.2. Finally, it exits.

The reason for this somewhat clumsy use of a temporary file is that the Norton Desktop batch language doesn't allow you to position the cursor within a file before writing to it. Thus, if the routine simply opened the print queue list and wrote the name of the file to be printed there, each time you ran the batch program it would overwrite the existing contents of the queue. This method preserves the previous contents of the file, with only a small cost in terms of elegance.

Figure 11.2

The Get File dialog box



The final two routines in GETFILE.WBT handle common errors that might arise during use of the batch program.

The WrongApp Routine

The first of these routines, WrongApp, is called when any of three possible error conditions occurs:

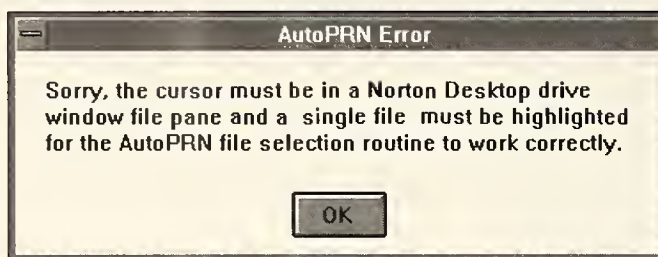
- There was no Norton Desktop drive window active at the time the batch file was launched
- The cursor wasn't in the file pane window
- The cursor was highlighting either a subdirectory name or more than one file (neither condition can be handled by GETFILE.WBT)

In the event that any of these errors occurs, the batch program closes the open dialog box and displays an error message, as shown in Figure 11.3. Once the user has clicked on the OK button in the error message box, the program exits.

```
:WrongApp
SENDKEY("{ESC}")
MESSAGE("AutoPRN Error", "Sorry, the cursor must be in a
Norton Desktop drive window file pane and a single file
must be highlighted for the AutoPRN file selection
routine to work correctly.")
EXIT
```


Figure 11.3

The error message displayed by the WrongApp routine



The AssocErr Jump Routine

The final routine in GETFILE.WBT is called if the extension of the file that the user has selected for printing is not listed in the Extensions section of the WIN.INI file. AutoPrint for Windows can't print such files, so rather than adding them to the AutoPrint queue, GETFILE displays an error message telling the user to use Norton Desktop's Associate command to fix the problem by identifying the program that should be associated with that kind of document.

```
:AssocErrJump
SENDKEY("{ESC}")
MESSAGE("AutoPRN Error", "Sorry, no association exists for
  this file, so it won't be printable. Use File Associate to
  correct the problem.")
EXIT
```

The AUTOPRN.WBT Batch File

AUTOPRN.WBT is the second batch program used by the Norton Desktop version of AutoPrint for Windows. Its function is also very simple. It sends the name of each document listed in the AutoPrint queue to Norton Desktop's File Print command. Then, when each file in the queue has been printed, it deletes the queue list and then exits.

The only tricky part of this is the timing. It's easy enough to send the command to print the first file in the queue, but once that's been done the batch program has to pause while Norton Desktop runs the application that will do the actual printing, directs it to print the file, and then closes the application down again. Only then should the batch program send the File Print command for the next file in the queue.

If you were to print several files in a row yourself using the File Print command, you would rely on visual cues to know when Norton Desktop was ready to accept another file to print: The cursor would change from an hourglass (indicating that Windows is busy) to an arrow, and the icon representing the program doing the printing would disappear from the screen.

Unfortunately, the batch language can't read the screen that way. However, it can count open windows, and then use the number it counts to determine whether the application used to print the current document is still active, or if it has been unloaded from memory, in which case Norton Desktop should be ready to print the next file in the queue.

AUTOPRN Dissected

AUTOPRN.WBT consists of five routines: Start, GetFile, PrintFile, TitleCheck, and ExitRoutine.

The Start Routine

The Start routine begins, as shown below, by initializing variables pointing to the Batch Runner directory (AutoPrnDir), the file containing the print list (AutoPrnList), and a variable that contains the horizontal tab character (HTab).

```
:Start
AutoPrnDir=DIRHOME()
AutoPrnList=STRCAT(AutoPrnDir, "AUTOPRN.LST")
HTab=NUM2CHAR(9)
```

Next, the routine checks to make sure the queue file specified in the variable AutoPrnList actually exists. If there is no queue file, that means that there are no files to print, so the program exits. Otherwise, it opens the file for reading.

```
IF FileExist("%AutoPrnList%")==@FALSE THEN EXIT
FHandle=FILEOPEN("%AutoPrnList%","READ")
```

Then, the Start routine activates Norton Desktop and closes any open Norton Desktop windows using the Close All command on Norton Desktop's Windows menu.

```
WINACTIVATE("Norton Desktop")
SENDKEY("!WA")
```

After that, the routine reads the WIN.INI file twice: first to identify the current default printer, and then to determine if the printer setup specifies that the Print Manager should be used to spool printer output. If so, it launches Print Manager, so that it will be included in the initial count of open windows that AUTOPRN.WBT uses to determine whether it is time to send the print command for the next file.

```
Printer=INIREAD("Windows","device","")
Spooler=INIREAD("Windows","Spooler","yes")
IF Spooler=="yes" THEN RUN("PRINTMAN.EXE"," ")
```

Finally, the Start routine uses the WINITEMIZE command to obtain a list of all open windows, and then the ITEM COUNT command to count that list:

```
Apps=WINITEMIZE()
AppCount=ITEMCOUNT(Apps,HTab)
```

The WINITEMIZE command automatically separates each member of a list with a horizontal tab character, so the second parameter to the ITEM-COUNT command tells Batch Runner to use a tab as the delimiter in counting the members of that list.

The GetFile Routine

The next routine, GetFile, consists of only two lines.

```
:GetFile
FileToPrint=FILEREAD(FHandle)
IF FileToPrint=="*EOF*" THEN GOTO ExitRoutine
```

The first line uses the FILEREAD command, which reads a single line of a file at a time, to get the next entry in the AUTOPRN.LST file. Then it checks to see if the end-of-file marker has been reached, and if so, it jumps to the Exit routine. Otherwise, it proceeds on to the PrintFile routine, which does the actual work of sending the commands to print the file.

The PrintFile Routine

The PrintFile routine starts by activating Norton Desktop for Windows and sending the File Print command. That opens the Print dialog box, to which the batch file sends the name of the file to print. Then it sends a Tab character to advance the cursor to the next field in the dialog box, a drop-down list box used to select the printer to be used. The routine automatically selects the first printer on the list, and then tabs over to the OK button and presses it by sending an Enter character. This closes the dialog box, instructing Norton Desktop to execute the print command. The PrintFile routine concludes by delaying for five seconds before it proceeds to the TitleCheck routine.

```
:PrintFile
WINACTIVATE("Norton Desktop")
SENDKEY("!FP")
SENDKEY("%FileToPrint%")
SENDKEY("{TAB}")
SENDKEY("!!{DOWN}{DOWN}")
SENDKEY("{TAB}{ENTER}")
DELAY(5)
```


The TitleCheck Routine

The TitleCheck routine is used to monitor the progress of the preceding print command. It consists of a brief loop that the program executes over and over until Norton Desktop is ready to print another file.

The key to this routine is the fact that Norton Desktop automatically closes down any programs launched using the File Print command after the print job for which they were launched has been completed. This means that between jobs the count of active applications should be the same as the initial count obtained during the Start routine.

```
:TitleCheck
ActiveWin=WINGETACTIVE()
IF ActiveWin=="Warning" THEN EXIT
IF ActiveWin=="Print Manager" THEN EXIT
IF ActiveWin=="Norton Desktop" THEN GOTO GetFile
NewApps=WINITEMIZE()
NewAppCount=ITEMCOUNT(NewApps,HTab)
IF NewAppCount==AppCount THEN WINACTIVATE("Norton Desktop")
GOTO TitleCheck
```

This routine begins by obtaining the title of the active window. During a print job, this will generally be the name of the program that created the file being printed. For instance, if you're printing a file with a .WRI extension, the title will be "Microsoft Write."

The second line of the routine checks to see whether the title starts with the word "Warning", which would indicate that Norton Desktop has detected a print error. The batch file responds to this by exiting, leaving the error message on screen for you to read when you return. The next line performs almost the same function, checking this time for an error message from Print Manager.

Next the routine checks to see whether Norton Desktop is the active application. If so, that means that the current print task has been completed, and so the routine jumps to the GetFile routine to start the next print job.

Otherwise, the routine obtains a new list of open windows, and counts that list. If a print job is active, the new count will be higher than the original window count, because the application doing the actual printing will be among those that are counted. If not, then the last print job must have finished, so the command to activate Norton Desktop is issued. If a job is active, then it loops back to the beginning of the routine and once again starts checking the title of the active window.

The ExitRoutine Routine

The final routine in the batch file, ExitRoutine, is performed after all the files in AutoPrnList have been printed. It begins with an INIWRITE command, which restores the default printer configuration in the WIN.INI file.

(This is necessary because the Norton Desktop File Print command tends to deactivate the default printer after it finishes printing a file.) Then it closes the AUTOPRN.LST file, deletes it, and exits.

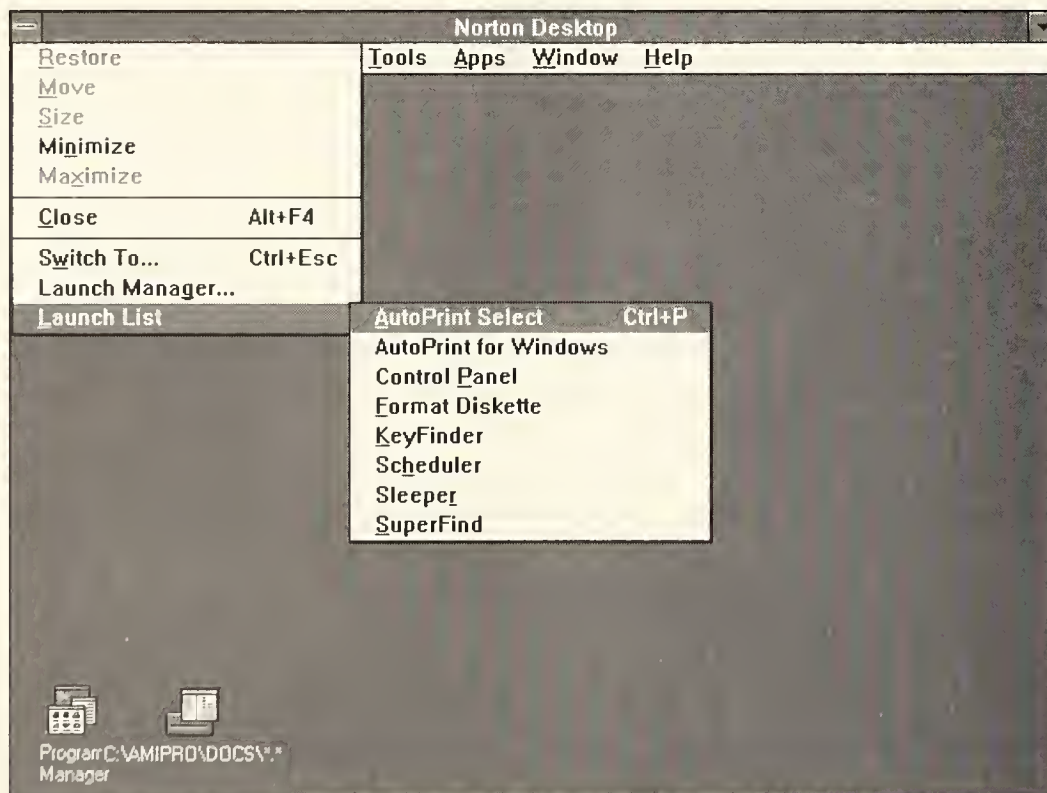
```
:ExitRoutine
INIWRITE("Windows","device",Printer)
FILECLOSE(FHandle)
FILEDELETE(AutoPrnList)
EXIT
```

Putting the Batch Files to Work

Once the two batch files were complete, all that remained to be done to finish the Norton Desktop version of AutoPrint was to add two menu items to the Launch List for launching GETFILE.WBT and AUTOPRN.WBT (as shown in Figure 11.4), and to schedule AUTOPRN.WBT to run each day.

Figure 11.4

The Launch List menu



The first new menu item, labeled “AutoPrint Select”, was given the following command line:

```
C:\NDW\BATCHRUN.EXE GETFILE.WBT
```

This links it to the GETFILE batch file. The option was also assigned the hotkey Ctrl-P, so that GETFILE would run whenever that key combination was pressed.

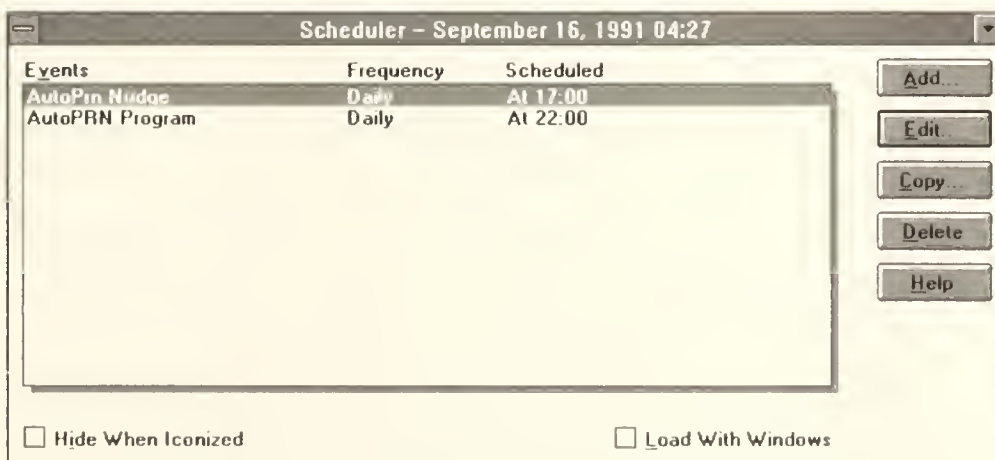
I also added an item to the Launch List menu for launching the AUTO-PRN batch file, labeling it “AutoPrint for Windows” and giving it this command line:

```
C:\NDW\BATCHRUN.EXE AUTOPRN.WBT
```

I didn’t assign a hotkey to this item since it doesn’t require the same level of instant access as the GETFILE.WBT batch program.

The final step in setting up the Norton Desktop version of AutoPrint for Windows was to add a pair of items to Norton Desktop’s Scheduler. The Scheduler allows you to specify events—either a message to be displayed or a program to execute—that you want to have occur at a specified time or at regular intervals, as shown in Figure 11.5. If you set up Scheduler to automatically run whenever Windows is launched, you can be sure that the events you’ve specified will occur as scheduled.

Figure 11.5
The Norton
Desktop Scheduler

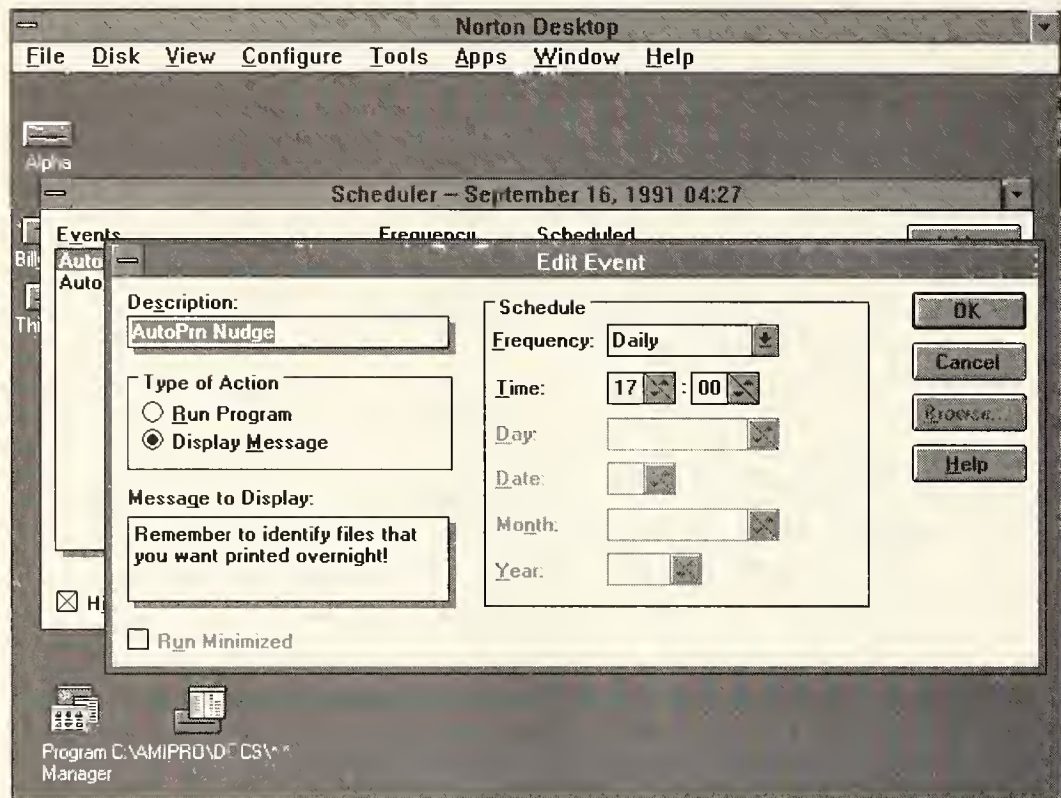


The first item added to the Scheduler was a reminder message: “Remember to identify files that you want printed overnight!”, shown in Figure 11.6. I scheduled it to pop up on my screen every day at 5:00 p.m.

Finally, I scheduled AUTOPRN.WBT to automatically run nightly at 10:00. So every night at 10:00 p.m. the AUTOPRN batch file launches, checks to see if a print queue list exists, and if so, prints the files on it.

Figure 11.6

Creating the daily AutoPrint nudge



Having the AUTOPRN.WBT file execute every day like this ensures that any files you've added to the print queue will automatically be printed. Of course, if you want to print them at some other time, you simply select the AutoPrint item from the Launch List to immediately run the AUTOPRN.WBT batch program.

The WinBatch Version

No sooner had I wrapped up the Norton Desktop version of AutoPrint for Windows than I realized that, as handy as some of Norton Desktop's features were in AutoPrint, they weren't absolutely essential. Using the capabilities of the WinBatch language alone, it would be possible to duplicate the Scheduler and Launch List capabilities used by the AutoPrint utility, and to launch and issue print commands to the applications that would print the documents listed in the print queue.

Like the Norton Desktop version, the WinBatch version of AutoPrint consists of two batch files: COPYMAC.WBT and AUTOP.WBT.

I was able to reuse many of the routines from the Norton Desktop version of AutoPrint in the WinBatch version. However, I did have to expand them significantly to support several new functions, including built-in

scheduling. Several changes were also necessary to reflect differences between Norton Desktop's drive windows and the Windows 3.1 File Manager.

Macro Files

The WinBatch equivalent of Norton Desktop's Launch List is WinMacro, a small utility that allows you to add menu items used to launch batch files to the Control menu of any Windows application. This facility is used in The Ultimate Notepad project, detailed in Chapter 9.

Whenever WINMACRO.EXE is running, it watches to see what other applications you load into memory, and adds any menu items that you've specified to their Control menus. These Control menu assignments are made through files with a .WDF extension. Thus to add a menu item to File Manager's Control menu, you would define the menu item in a file called WINFILE.WDF—File Manager's executable file is called WINFILE.EXE.

To get the ball rolling with the WinBatch version of AutoPrint, I created a WINFILE.WDF file that contained the following single line:

```
Add file to AutoPRN queue      \^P:WINBATCH.EXE COPYMAC.WBT
```

This adds a menu item labeled "Add file to AutoPRN queue" to File Manager's Control menu, as shown in Figure 11.7, and assigns it the hotkey Ctrl-P. Whenever you press the hotkey or select the menu item, the batch file COPYMAC.WBT is executed.

The COPYMAC.WBT Batch File

The COPYMAC.WBT batch program is the equivalent of the GETFILE.WBT batch program used by the Norton Desktop version of AutoPrint. However, there is a fundamental difference in the way the two operate.

GETFILE.WBT takes advantage of the fact that Norton Desktop's Print command displays the full path of the file it is about to print, which allows GETFILE to copy the file's name and path into the AutoPrint queue list. Unfortunately, the same command issued in Windows 3.1's File Manager merely displays the name of the file in the editable text field, not the full path. Thus, there is no way for WinBatch to determine the file's path, and thus no way for it to find the file again when it's time for it to be printed.

To solve this difficulty, COPYMAC issues a File Copy command, and copies the file that is being designated for later printing into a subdirectory called C:\AUTOPRN. Then, when the AUTOP.WBT batch program is executed later, it prints and deletes each file that it finds in the C:\AUTOPRN directory. This method is slower and most costly in terms of disk space—although the disk space used to store the files to be printed is reclaimed once they have been printed.

Figure 11.7

Modifying File Manager's Control menu



COPYMAC Dissected

COPYMAC.WBT consists of three routines: Main, WrongApp, and AssocErrJump.

The Main Routine

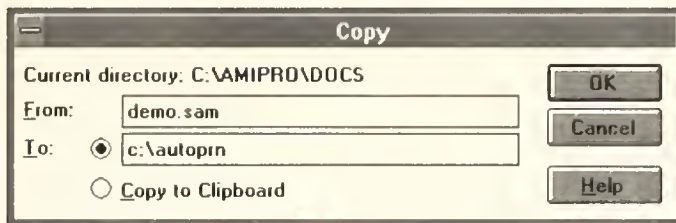
The Main routine starts by emptying the Clipboard, and then issues the File Copy command. It then checks the title of the current window, which should be Copy. If it is not, then the routine is not working, so it jumps to the WrongApp error routine.

```
:Main
CLIPPUT( "")
SENDKEY("!FC")
A=WINGETACTIVE()
IF A<>"Copy" THEN GOTO WrongApp
```

File Manager's Copy dialog box includes two editable text fields. When the dialog box opens, the first field contains the name of the file that was highlighted in the File Manager window when the Copy command was issued. The second field, which is used to designate the destination for the copied files, is either blank or contains the name of the last copy destination

if the Copy command has already been used in the current Windows session, as shown in Figure 11.8. The cursor is located in the second edit field.

Figure 11.8
File Manager's
Copy dialog box



At this point the batch program must examine the file name in the first edit field to ensure that it is valid, so the program sends a Shift-Tab key-stroke to move the cursor up to that field, copies its contents to the Clipboard, and then assigns them to the variable File, as shown in the first three lines below.

```
SENDKEY("+{TAB}")
SENDKEY("^{|INSERT}")
File=CLIPGET()
I=STRSCAN(File,"\",1,@FWDSCAN)
IF I<>0 THEN GOTO WrongApp
File=STRTRIM(File)
Ext=FILEEXTENSION(File)
AssocExists=INIREAD("Extensions",Ext,"@TRUE")
If AssocExists==@FALSE THEN GOTO AssocErrJump
```

The batch program then uses the STRSCAN command to determine if the variable File contains a backslash character. If it does, then the field contains a directory name rather than a file name, so the batch program jumps to the WrongApp error routine. Otherwise, it trims any excess spaces off the file name and then checks that the document's file extension is listed in the Extensions section of the WIN.INI file. If not, it jumps to the AssocErrJump error routine.

Next, COPYMAC.WBT sends a Tab character to advance the cursor to the destination field, and then sends C:\AUTOPRN to the destination field and closes the dialog box, initiating the file copy by simulating a press of the Enter key.

```
SENDKEY("{TAB}")
SENDKEY("C:\AUTOPRN")
SENDKEY("{ENTER}")
EXIT
```

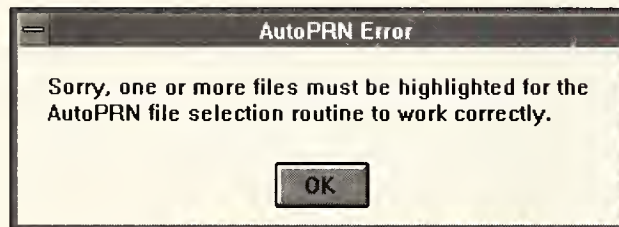
The WrongApp Routine

The next routine in COPYMAC.WBT is a general-purpose error routine called WrongApp. It simply closes the Copy dialog box and displays an error message (shown in Figure 11.9), if for any reason COPYMAC.WBT was not able to process the file or files that were highlighted in File Manager when it was run.

```
:WrongApp
SENDKEY("{ESC}")
SENDKEY("{ESC}")
MESSAGE("AutoPRN Error", "Sorry, one or more files must be
  highlighted for the AutoPRN file selection routine to
  work correctly.")
EXIT
```

Figure 11.9

The WrongApp error dialog box



The AssocErrJump Routine

The AssocErrJump routine is called if the highlighted file's extension isn't listed in the WIN.INI file. It's very straightforward.

```
:AssocErrJump
SENDKEY("{ESC}")
SENDKEY("{ESC}")
MESSAGE("AutoPRN Error", "Sorry, no association exists for
  this file, so it won't be printable. Use File Associate
  to correct the problem.")
EXIT
```

The AUTOP.WBT Batch File

AUTOP.WBT is the WinBatch version of AUTOPRN.WBT. The most obvious difference between the two is the addition of a time-checking routine to AUTOP.WBT, which allows you to specify the time you want files in the AUTOPRN directory to be printed.

When AUTOP.WBT is run, it displays a dialog box (shown in Figure 11.10) that contains an edit field in which you can enter the time to print the queued files. The default is to print the files immediately. But if you specify a later time, the dialog box disappears, and AUTOP also disappears—except for its icon at the bottom of the screen—until that time arrives.

Figure 11.10

The START.DLG dialog box



Alternatively, when you run AUTOP.WBT, you can include a command line parameter specifying the starting time. The starting time must be specified in 24-hour format. So if you wanted to run AUTOP.WBT at 10:00 p.m., you could issue the File Run command in Program Manager or File Manager, and give it the following command line:

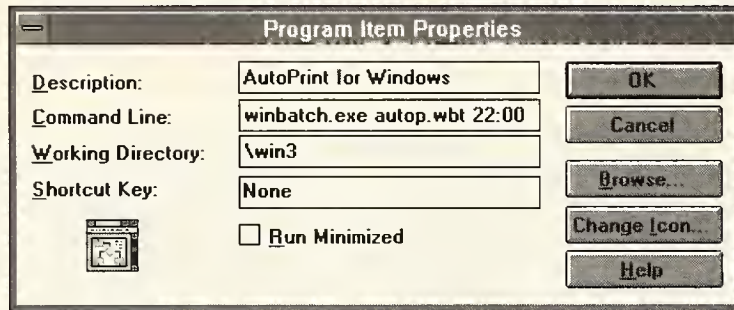
```
WINBATCH.EXE AUTOP.WBT 22:00
```

Similarly, if you wanted the print queue emptied at 10:00 p.m. every day, you could create a Program Manager icon with that same command line, as shown in Figure 11.11, and place it in Program Manager's StartUp folder. In that case, AUTOP would run whenever you launched Windows, and would present you with the START.DLG dialog box suggesting 22:00 as the starting time. You could press Enter to accept that time, in which case any files in the AUTOPRN directory would be printed at 10:00 p.m. Alternatively, you

could enter a different starting time, or you could type **NOW** to begin printing immediately.

Figure 11.11

Creating a Program Manager icon for AutoPrint



AUTOP Dissected

AUTOP.WBT consists of nine separate routines: Start, ConfirmTime, TimeCheck, StartNow, PrintFileLoop, GetWindTitle, EndLoop, WentFine, and NextFile.

The Start Routine

The Start routine begins by changing the name of the batch program's icon (which is always visible at the bottom of the screen while the batch program is running) from its default of "WBT -- AUTOP.WBT" to "AutoPrint for Windows".

```
:Start
WINTITLE("WBT","AutoPrint for Windows")
if Param0>0 THEN Start=Param1
if Param0==0 THEN Start="NOW"
```

Then it checks the value stored in the global variable Param0, which contains a count of the number of command line parameters that were passed to the batch program. If at least one command line parameter was passed, it assigns the contents of the first one to the variable Start. Otherwise it assigns "NOW" to Start.

The Confirm Time Routine

Next, the program displays a dialog box showing the value it has assigned to Start and asking you to confirm the starting time. You can edit this value using the form hh:mm in 24-hour time format. Here's that code:

```
WATCH
:ConfirmTime
DIALOGBOX("AutoPrint for Windows","START.DLG")
```

The dialog box used by AUTOP.WBT is defined in the file START.DLG, as follows:

```
Start print job at: [Start#   ]
(Enter time in 24 hour format as hh:mm, or type
NOW to begin immediately.)
```

This dialog box definition tells WinBatch to display the prompt “Start print job at:” followed by an editable text box containing the value of the variable Start and, two lines beneath that, another text message that explains the form in which a starting time should be entered. WinBatch automatically adds OK and Cancel buttons to the dialog box definition. Pressing the Cancel button automatically aborts the script.

Once you click the OK button to close the dialog box, the ConfirmTime routine examines the starting time that you’ve specified. If Start contains the value “NOW”, the program jumps to the StartNow routine. Otherwise, it tests the value of Start to ensure that it is a valid time.

```
IF STRUPPER(Start)=="NOW" THEN GOTO StartNow
Colon=STRSCAN(Start,":",1,@FWDSCAN)
IF Colon==0 then Start=STRCAT(Start,":00")
IF Colon==2 THEN Start=STRCAT("0",Start)
IF STRLEN(Start)<5 THEN GOTO ConfirmTime
StHour=STRSUB(Start,1,2)
StMin=STRSUB(Start,4,2)
IF ISNUMBER(StHour)==@FALSE THEN GOTO ConfirmTime
IF ISNUMBER(StMin)==@FALSE THEN GOTO ConfirmTime
IF StHour>23 THEN GOTO ConfirmTime
IF StHour<0 THEN GOTO ConfirmTime
IF StMin>59 THEN GOTO ConfirmTime
IF StMin<0 THEN GOTO ConfirmTime
IF STRSUB(StHour,1,1)=="0" THEN StHour=STRSUB(StHour,2,1)
WINTITLE("AutoPrint","AutoPrint for Windows--Waiting for
%StHour%:%StMin%")
```

The STRSCAN function is used to locate the colon (:) character within the contents of the variable Start. If Start doesn’t contain a colon, the script adds a colon and a pair of zeros to the end of the current contents of Start, on the assumption that you’ve simply entered 18, for example, when you meant 18:00. If, on the other hand, it finds a colon in the second position, it tacks a zero onto the beginning of Start, in an attempt to change the time to 24-hour format; 9:00 becomes 09:00, for instance.

Next, the routine checks the length of Start. If it is less than five characters long, it cannot contain a valid time, so the program jumps back to the beginning of the ConfirmTime routine and redisplay the dialog box so that

you can reenter the Start time. Otherwise, it assigns the first two characters of Start to the variable StHour, and the fourth and fifth characters to StMin. It then uses the ISNUMBER function to determine if both variables contain integers. If either variable fails that test, the time entered must not be valid, so the script jumps back and redisplay the dialog box.

The script then checks that the values of StHour and StMin fall within valid ranges, that is, the specified starting hour is from 0 to 23, and the specified starting minute is from 0 to 59. Once again, if any of those tests fail, the routine jumps back and redisplay the dialog box.

Finally, the routine strips off the preceding zero that it added to the starting hour (if a single-digit hour was specified), and then changes the title of the program's icon to reflect the time at which the print job will begin.

The TimeCheck Routine

The next routine, TimeCheck, is a simple loop that checks the current time every 15 seconds to determine if the designated starting time has arrived. It uses the DATETIME function to obtain the current time, and then the STRSUB function to break the value returned by DATETIME down into the current hour and current minute. (DATETIME supplies the date and time in a single long string, such as "Wed 10/02/91 2:26:07 PM".) Then it converts the current hour to 24-hour format and compares the current hour and current minute to the values of StHour and StMin. Here's what it looks like:

```
:TimeCheck
DELAY(15)
CurrTime=DATETIME()
CurrHour=STRSUB(CurrTime,14,2)
CurrHour=STRTRIM(CurrHour)
CurrMin=STRSUB(CurrTime,17,2)
CurrAMPM=STRSUB(CurrTime,23,2)
IF CurrAMPM=="PM" THEN CurrHour=CurrHour+12
IF CurrHour==24 THEN CurrHour=0
IF CurrHour<>StHour THEN GOTO TimeCheck
IF CurrMin<StMin THEN GOTO TimeCheck
```

The StartNow Routine

If the designated starting time has been reached, execution of the program continues past the end of the TimeCheck routine to the StartNow routine, which initializes a series of variables: HTab (used to parse the list of open windows), AutoPrnDir (the directory in which queued files are stored), Files (a list of the files in AutoPrnDir), FileCount (the number of files in AutoPrnDir), and Counter (used to keep track of the file being printed). The routine then launches Print Manager, if it is set up to run, and then counts the number of open windows.


```

:StartNow
HTab=NUM2CHAR(9)
AutoPrnDir="C:\AUTOPRN\"
Files=FILEITEMIZE("%autoprndir%*.*)"
FileCount=ITEMCOUNT("%Files%"," ")
IF FileCount==0 THEN EXIT
Counter=1
Spooler=INIREAD("Windows","Spooler","yes")
IF Spooler=="yes" THEN RUN("PRINTMAN.EXE","")
Apps=WINITEMIZE()
AppCount=ITEMCOUNT(Apps,HTab)

```

The PrintFileLoop Routine

The PrintFileLoop routine is the next to execute. It attempts to run each document in the AUTOPRN directory. When you “run” a document for which an associated executable file is specified in WIN.INI, Windows launches the executable file and instructs it to load the document. Then, once the document has been loaded into the application program, the PrintFileLoop routine issues the standard File Print command to the program, waits one second, and then sends a carriage return (if necessary) to close the Print dialog box. (Some programs require this, some don't.)

```

:PrintFileLoop
FileToPrint=ITEMEXTRACT(Counter,Files," ")
RUN("%AutoPrnDir%%FileToPrint%","")
DELAY(5)
App=WINGETACTIVE()
SENDKEY("!FP")
DELAY(1)
IF WINGETACTIVE()<>App THEN SENDKEY("~")
DELAY(1)

```

The GetWindTitle Routine

Next, the GetWindTitle routine repeatedly checks the name of the current active window. While printing is under way this window will generally be a print status dialog box of some kind. (Again, this varies from program to program.) The routine loops until the print status dialog box closes and the application's main window is active again.

```

:GetWindTitle
CurrApp=WINGETACTIVE()
IF CurrApp==App THEN GOTO EndLoop
DELAY(1)
GOTO GetWindTitle

```

The EndLoop Routine

The next routine, EndLoop, sends the standard File Exit command to close down the program that just printed the document. If the program doesn't close immediately, it sends a pair of N's to close any dialog boxes of the save-before-closing ilk that the program may have created as part of its shutdown process.

```
:EndLoop
SENDKEY("!FX")
DELAY(1)
IF WINEXIST(App) THEN SENDKEY("N")
DELAY(1)
IF WINEXIST(App) THEN SENDKEY("N")
WINWAITCLOSE(App)
GOTO WentFine
```

The routine pauses, using the WINWAITCLOSE() command, until the application is completely shut down before proceeding to the WentFine routine.

The WentFine Routine

WentFine deletes the file that has just been printed from the AUTOPRN directory (the original file will remain in the directory from which COPY-MAC.WBT copied it) and then jumps to the NextFile routine.

```
:WentFine
FILEDELETE("%AutoPrnDir%%FileToPrint%")
GOTO NextFile
```

The NextFile Routine

The NextFile routine adds 1 to the current value of the variable Counter, and then checks to see whether the value of Counter is now higher than the total number of files to print. If so, its job is done, so it exits. If not, it jumps back to the PrintFileLoop routine and prints the next file.

```
:NextFile
Counter=Counter+1
IF Counter>FileCount THEN EXIT
GOTO PrintFileLoop
```

This concludes the AutoPrint for Windows application. However, the techniques used here for launching applications, automating their operations, and scheduling events should be of use in any number of your own Windows development projects.

CHAPTER

12

Making Use of Libraries—Recycler

Basic Operations

*Defining Functional
Requirements*

*The Application
Framework*

*Exploring
GLOBAL.BAS*

Exploring FORM1.FRM

*Exploring
DRAGDROP.BAS*

Wrapping Up Recycler

THIS PROJECT, RECYCLER, DEMONSTRATES HOW EASILY YOU CAN EXTEND the power of Windows 3.1 development tools by taking advantage of dynamic link libraries. The Recycler application acts as a recycling-bin for disk files. When you no longer want a file, you drag it from the Windows File Manager into the recycling bin. Then later, when you empty the bin, the disk space the file occupied is recycled—or made available to you once again. But in the meantime, if you decide you can't live without one or more of these files, you can restore them any time before you empty the bin.

This might sound faintly reminiscent of the Macintosh Trash application, and in truth it does bear a certain resemblance to that classic. However, in keeping with the green-conscious '90s, the emphasis here is on recycling, rather than contributing to overcrowded landfills or hard disks.

Basic Operations

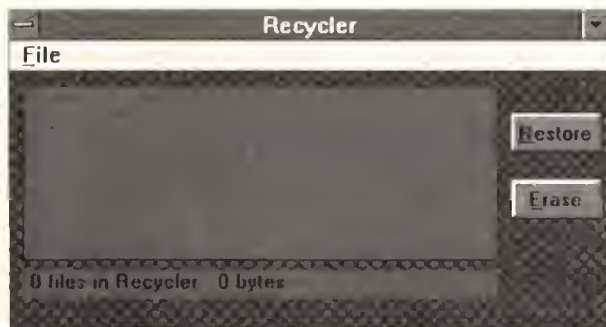
When you launch Recycler, it appears as an icon (shown below) at the bottom of your screen.



You can click on the Recycler icon to open its window, but doing so won't be of much interest until you've dragged files into the Recycler; before that the window merely displays an empty list box, as shown in Figure 12.1.

Figure 12.1

Recycler's window when the recycling bin is empty



Things get more interesting when you open up the Windows 3.1 File Manager, highlight a few files, and drag them onto the Recycler's icon. Several things then happen at once. The Recycler icon changes to display an overflowing bin, and its label changes to display the number of files that you've dragged onto it and their total size in kilobytes, as shown below.

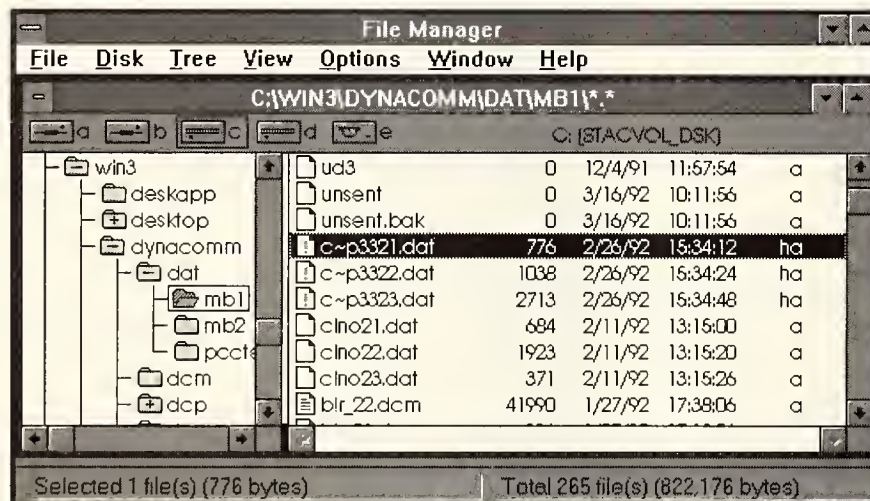
Moreover, the files you've dragged onto the Recycler disappear from the File Manager's file listing.



Recycler doesn't actually delete the files that you've dragged onto it (or even move them for that matter); it simply turns on the DOS Hidden attribute for each file. This makes the files disappear from the File Manager's file listing, unless you have turned on the Hidden/System Files option on File Manager's View By File Type dialog box. If you have set this option, the files in the recycling bin will appear in File Manager's listing with an exclamation mark on the small icon that precedes each file's name, as shown in Figure 12.2.

Figure 12.2

File Manager will list files in the recycling bin only if you have turned on its Show Hidden/System Files option

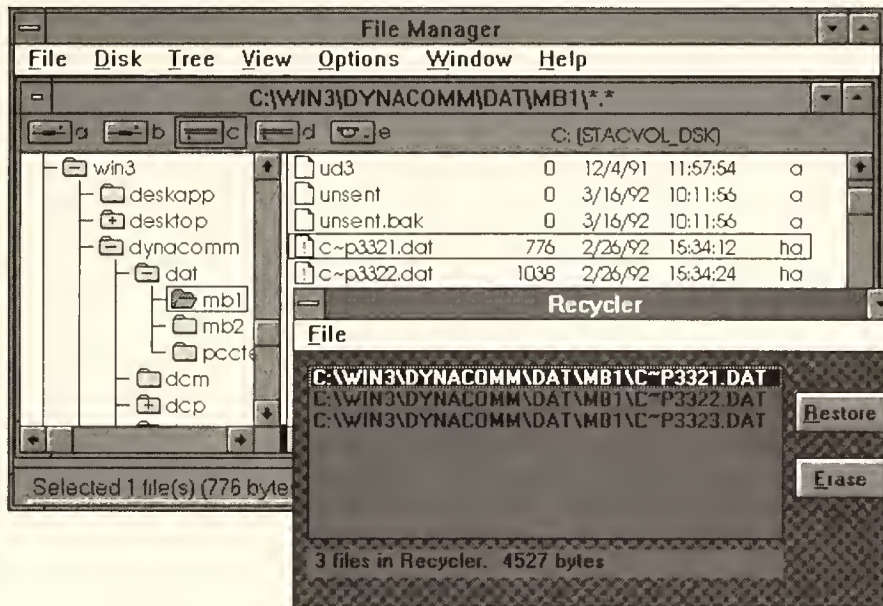


In addition to removing the files from the standard File Manager listing, hiding them in this manner makes them invisible both to most common file operations at the DOS command line (including the COPY and DELETE commands) and to the standard Windows File Open dialog box. So if there are 18 TXT files in a directory, and you drag 16 of them into the Recycler, a COPY *.* command at the DOS command line will only copy the two unhidden files, and the File Open dialog box in Notepad (or any other application that works with TXT files) will list only those two files.

When you open the Recycler window after you've dragged files onto it, you'll find that its central list box contains a list of those files, as shown in Figure 12.3. The list box contains the name and full path of each file that you've dragged onto Recycler.

Figure 12.3

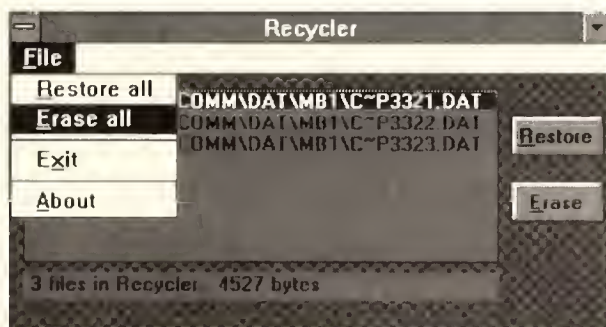
Recycler's list box lists the files it has hidden



The Restore button can be used to unhide whichever file is currently selected in the list box. When you do so, the file disappears from Recycler's list box and reappears in the directory in which it originated. Alternatively, you can use the item labeled "Restore all" on Recycler's File menu (shown in Figure 12.4) to unhide all the files in the bin, or the one labeled "Erase all" to permanently delete all the listed files.

Figure 12.4

Recycler's File menu offers the opportunity to delete or restore all files in the bin

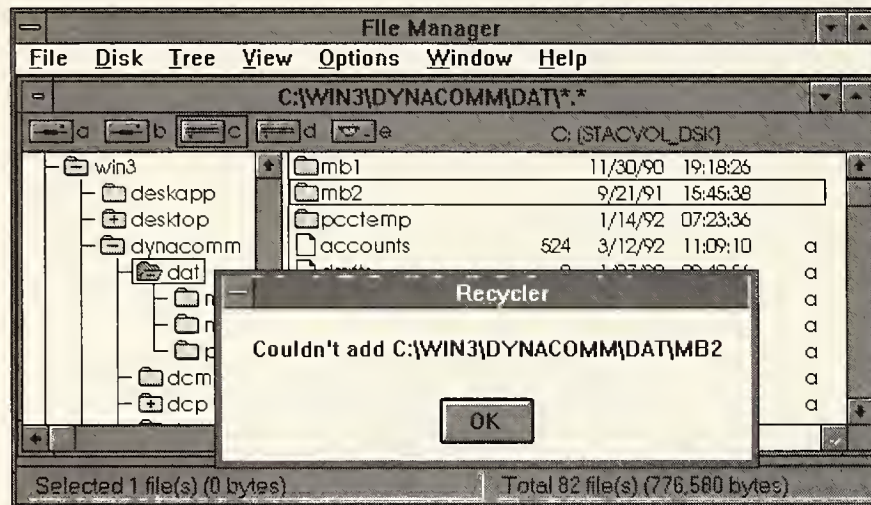


Capabilities and Limitations

That about sums up Recycler's basic features. However, there are also a couple of limitations worth noting. The first is that Recycler doesn't work with subdirectories, because the dynamic link library used by the application doesn't know how to hide directories. So if you drag a subdirectory onto Recycler, it will refuse to accept it, and will post the error message shown in Figure 12.5.

Figure 12.5

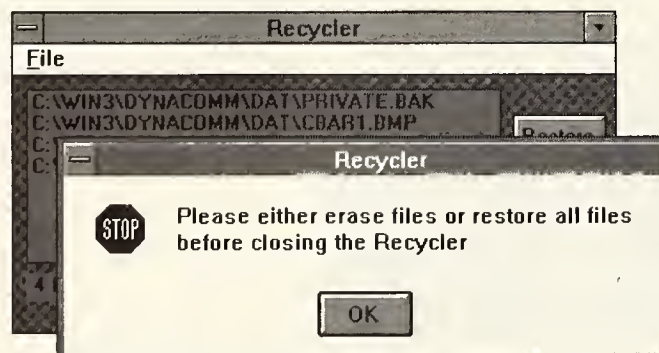
Recycler displays this error message if you drag a subdirectory onto it



The second limitation is that Recycler doesn't keep track of the files in the recycling bin from one session to another. Thus, you must determine whether you want to erase or restore each file in the bin before you shut down the Recycler application. If you try to close Recycler (either by attempting to shut down Windows or by selecting Recycler's File Exit command or the Close command on its System menu) while there are files in the recycling bin, it will post the error message shown in Figure 12.6, and put a halt to the shutdown process.

Figure 12.6

Recycler insists that you either erase or restore the files in the recycling bin before shutting down the application



Despite these limitations Recycler is a useful application. It gives you a chance to think twice about whether you actually want to delete a file, and it also gives you a chance to do a little what-if experimentation with disk space. For instance, if you're trying to clear 10 megabytes of space on your hard drive, you can drag files from various directories into Recycler until it indicates that they total the amount of space you need, and then erase them all

at once. Or you can use File Manager's Search command to find all the BAK or BMP or TXT files on your hard disk, and then drag whichever ones you don't need onto the Recycler—which again allows you to see whether the total disk space they occupy matches the amount you want to free up.

The advantage to using Recycler in this way is that the totals it maintains are cumulative. You can drag files from several different directory windows, and/or the results from several different searches, into Recycler, and it will always indicate the total size of all the selected files. Then it allows you to delete or restore the whole group with a single menu command.

Defining Functional Requirements

Recycler revolves around two key functions: the ability to accept drag-drop messages from File Manager and the ability to hide files.

One of the neatest new features of Windows 3.1 from a programmer's standpoint is that, with very little effort, you can make your application send a message to Windows that says "I accept drag-drop messages." Once this has been done, Windows will alert the application when the user drags files from File Manager onto the application's window and drops them there.

Recycler uses this capability as its only input source for the files it is to hide. You can select any number of files in File Manager, drag them onto Recycler by holding down the left mouse button, and then release the button to add them to the recycling bin. Thus the first major functional requirement for Recycler was that it had to be able to send Windows the "files accepted here" message and react when Windows sends it back an "OK, you've got 'em" message in response to the user dragging files onto the application.

The other major functional requirement for the application was that it had to be able to hide the files that were dropped onto it.

One of the primary purposes of the Recycler application is to make the files dropped on it disappear from file listings and make them invisible to most standard file-manipulation commands. There are several ways to accomplish this. I could, for instance, have had all the files copied to a special Recycler directory and then deleted from their original locations. But there was little additional benefit to be gained from actually moving the files to a new directory, as opposed to merely hiding them. And the additional house-keeping needed to track each file's original and new location seemed like too much work for too small an advantage.

Selecting the Development Tool

The functional requirements for the Recycler project meant that the development tool had to be able to utilize the Windows application programming interface (API) functions that control File Manager's drag and drop features.

and had to offer a way to set DOS file attributes. In addition, the user-interface aspects of the development tool had to be strong enough to let me build an easy-to-use interface for the application.

Eventually I decided to build the project using Visual BASIC, largely because in the midst of experimenting with the new Windows 3.1 drag-drop functions, I had accidentally discovered what appeared to be a simple way to implement support for them in Visual BASIC. I wanted to test that method in a real-world application, especially since I'd had several knowledgeable people express surprise and doubt that it would work: They believed it was impossible to build a generic message loop in Visual BASIC.

Message Loops

When you dig down deep into its substructure, every Windows application revolves around a central message-handling loop that constantly monitors and responds to the messages Windows sends to the application. These messages span an enormous range, everything from "the mouse just moved" to "a button was clicked" to "Windows is shutting down."

Languages such as C and Turbo Pascal for Windows provide the developer with complete control over how an application responds to these messages. But that means the developer also has complete responsibility for ensuring that the messages are handled correctly, since the application won't make any automatic assumptions about how a message should be handled. (The situation isn't as odious as it sounds, however, because an application can call the DefWindowProc function, which provides default processing for any Windows message that the application doesn't otherwise process. Nevertheless, there is a considerable responsibility on the part of the programmer to make sure that all the i's are dotted and t's are crossed in terms of handling messages.)

In Visual BASIC, on the other hand, the message-handling loop is hidden away, beyond the programmer's control. Visual BASIC automatically handles all the message processing for your application, relieving you of the responsibility of doing so, but also constraining your ability to make use of the full range of Windows functions. Visual BASIC provides automatic handling of those messages related to its core commands and functions, and throws away the rest. The programmer cannot add additional functions or hooks to Visual BASIC's central message loop.

Nevertheless, it is possible to sneak a peek at the messages Windows is sending to your application. Visual BASIC has the ability to use functions in external dynamic link libraries, and you can take advantage of that ability by accessing the PeekMessage function in USER.EXE, the dynamic link library that supplies most of the Windows user-interface functions.

The PeekMessage function checks the specified application's queue for a message, and places it in a special message-structure variable that your

application can examine. A Visual BASIC application can call this function, just as it can call nearly any other Windows API function. However, it doesn't do much good to call `PeekMessage` just once, since if there is no message waiting for the application, the function simply returns a value of 0—it doesn't wait for a message to actually arrive. So in order to ensure that `PeekMessage` captures all the messages for your application as they arrive, you have to call it over and over again. In other words, you've got to build a message loop.

Fortunately, Visual BASIC provides a device that allows you to do just that; it's called a `DoEvents` loop. A `DoEvents` loop is a closed loop that your Visual BASIC application performs whenever it is not busy doing something else. In other words, whenever your application isn't busy carrying out some other command, it executes the statements in the `DoEvents` loop, then checks to see if there is anything else it should be doing (such as responding to user input or carrying out a sort function or redrawing its window). If not, it executes the statements in the `DoEvents` loop again.

Visual BASIC `DoEvents` loops are not well suited to act as general-purpose message loops, because Windows messages can pour into an application at overwhelming speed, especially when the user is interacting with that application. And although Visual BASIC is remarkably fast for a pseudo-compiled BASIC, it isn't fast enough to allow a user-coded message loop (as opposed to Visual BASIC's underlying message loop) to keep up with rapid mouse movements or keystrokes. By the time Visual BASIC reacts to one mouse-movement message it may have missed ten more.

Nevertheless, I discovered that despite this inadequacy, Visual BASIC's `DoEvents` loop was more than capable of handling drag-drop messages from File Manager. For one thing, although Windows might be multitasking, Windows users are not. So if the user is dragging files from File Manager onto your Visual BASIC application, he or she isn't also interacting with your application at the same instant. Therefore your application is almost guaranteed to be performing its `DoEvents` loop, rather than responding to user input, at the instant the files are dropped onto it. (This wouldn't necessarily be the case if your application was one that performed extensive background processing, but in an application such as Recycler, which responds exclusively to user input, you can be assured that the `DoEvents` loop will be waiting for the message.)

In addition, the File Manager drag-drop messages come in an easily handled sequential form. First comes the message that one or more files have been dropped onto your application. Then your application asks, "How many files?" and Windows responds with the total number of files that were dropped. Your application can then request the name and path of each file, one at a time, and perform whatever processing is required using that file, before requesting the next name. Since your application controls the dialog with Windows, there is no chance that it will miss one of the file names while it is busy doing something else.

In short, I discovered it was possible to create an apparently foolproof loop for handling drag-drop messages with less than a dozen lines of Visual BASIC code.

DOS Attribute Control

Of course, in addition to obtaining the names of files from the Windows drag-drop messages, Recycler had to be able to do something with those files. Specifically, I wanted to be able to hide the files that were dropped onto the recycling bin.

Alas, Visual BASIC's built-in command set provides very little in the way of file-manipulation functions. It offers ways to create files, delete them, rename them, and obtain their length, but none for setting their attributes. Fortunately, however, there is a growing library of public domain and shareware utilities for extending Visual BASIC. In building Recycler, I was able to make use of a public domain dynamic link library called DISKSTAT.DLL, which was written and posted on CompuServe's MSBASIC forum by Art Krumsee. DISKSTAT.DLL, which is included with the Recycler project code on the disk that accompanies this book, provides functions for obtaining and setting file attributes, as well as several other disk- and file-related functions, all of which can be accessed by Visual BASIC applications.

The Application Framework

The Recycler application was generated from three files: GLOBAL.BAS, FORM1.FRM, and DRAGDROP.BAS.

GLOBAL.BAS is a Visual BASIC global file. It contains the declarations for the global variables used throughout the Recycler application, as well as definitions for the user-defined data types and functions and the external procedures (DLL calls) that the application uses.

FORM1.FRM is a *form*: a special Visual BASIC file type that includes both a window design and program code associated with that window. Often an application will have several forms, each describing a separate window or dialog box. Recycler has only one custom display window, however, since it uses the standard MSGBOX function to create messages to the user, and thus it requires only one Form file.

DRAGDROP.BAS is a module file, meaning that it consists entirely of program code, most of which takes the form of program subroutines or user-defined functions called by other routines. In addition, DRAGDROP.BAS contains a routine called MAIN, which acts as the startup routine for Recycler, meaning that the instructions in that routine are the first to be executed when you launch the Recycler application.

Let's start off our examination of Recycler by exploring the GLOBAL.BAS file.

Exploring GLOBAL.BAS

Recycler uses GLOBAL.BAS to hold definitions of all the function and subroutine calls it makes to the Windows API files and to DISKSTAT.DLL, and to define and initialize a series of global variables.

The GLOBAL.BAS file begins by defining two special variable types—PointAPI and Msg. These are used by some of the API calls that follow, and their definitions must appear in the GLOBAL.BAS listing prior to the functions or subroutines that make use of them.

Type Definitions

The PointAPI type is used by several Windows API functions to designate the horizontal and vertical location of a specified point on the screen. Its definition looks like this:

```
Type PointAPI
X As Integer
Y As Integer
End Type
```

In other words, a variable of type PointAPI will consist of two integers, X and Y. These indicate the horizontal and vertical locations, respectively, of the specified point.

The next definition, that of the variable type Msg, is a bit more complex:

```
Type Msg
hWnd As Integer
Message As Integer
wParam As Integer
lParam As Long
Time As Long
PT As PointAPI
End Type
```

The Msg variable type is used to convey a variety of information about a Windows message. It consists of six components: first the identity of the window that is to receive the message (passed in the form of an integer labeled hWnd); then the message number (in the form of an integer labeled Message); then a pair of components (the integer wParam and the real or long number lParam) whose definition and purpose vary according to the value of Message; then the time at which the message was posted (in the form of a long variable called Time); and finally the position of the screen cursor when the message was posted (in the form of a PointAPI variable called PT).

Visual BASIC can work with user-defined variables like these as a unit, by defining a variable to be of the specified type, as follows:

```
DIM NewStuff AS MSG
```

or it can work with the components that make up the user-defined variable, like this

```
MessNum=NewStuff.Message
```

or

```
MessTime&=NewStuff.Time
```

Function and Subroutine Definitions

Next the GLOBAL.BAS file defines five external functions or subroutines that allow Recycler to make use of the capabilities it needs from external DLLs.

Once you have defined an external function or subroutine in Visual BASIC, your application can use it in exactly the same manner as it uses Visual BASIC's core functions and commands. For instance, the Windows API function `IsWindowVisible` takes a single integer parameter and returns a single integer value, just like the internal Visual BASIC function for obtaining the size of an open disk file, `LOF`. Thus once you have defined the `IsWindowVisible` function, you can use it in your application as easily as you can use `LOF`. The statement

```
WinVis%=IsWindowVisible(WinHandle%)
```

would return an integer indicating whether or not the window designated by `WinHandle%` is visible (after this function, `WinVis%` would be equal to `-1` if `WinHandle%` is visible, or `0` if it is not), just as

```
FileLen%=LOF(FileNum%)
```

would return the length of the file indicated by `FileNum%` to the integer variable `FileLen%`.

Before you can use the Windows API functions, or functions or subroutines from any dynamic link library, you have to define them for Visual BASIC, telling it the parameters the function or subroutine requires and what value, if any, it returns.

The first definition in GLOBAL.BAS is for the `PeekMessage` function, which is used in the `DoEvents` loop that enables Recycler to accept drag-drop messages. The function definition looks like this:

```
Declare Function PeekMessage Lib "USER" (lpMsg As Msg, ByVal hWnd As Integer, ByVal  
wMsgFilterMin As Integer, ByVal wMsgFilterMax As Integer, ByVal wRemoveMsg As  
Integer) As Integer
```

This tells Visual BASIC that the function `PeekMessage` is found in the Windows library `USER.EXE`, that it accepts five parameters, and that it returns an integer value. In order to use `PeekMessage`, you have to pass it the following: a pointer to a `Msg` type variable to use for storing the message; an integer identifying the handle (the integer by which Windows identifies the window) of the window whose messages you want to view; a pair of integers indicating the numbers of the lowest and highest messages that interest you (all Windows messages are numbered—for instance, the `WM_DROPFILES` message, used to indicate that File Manager has dropped files onto the window, is number 563); and an integer indicating whether the message should be removed from the message queue after processing by `PeekMessage`. The integer returned by the function indicates whether or not a message was found. It will be equal to 0 if no message was found, and to -1 if one was located.

Calling the Drag-Drop Functions

The next three definitions in `GLOBAL.BAS` access three routines from the Windows 3.1 library `SHELL.DLL` that control the File Manager drag-drop interface. These are `DragAcceptFiles`, which Recycler sends to indicate that it is willing and able to accept File Manager drag-drop messages; `DragQueryFile`, which Recycler uses to find out what files were dropped onto it after `PeekMessage` returns the `WM_DROPFILES` message; and `DragFinish`, which Recycler uses after obtaining the list of files to tell Windows that it can discard the data structure it had reserved to hold the file names from the `DragQueryFile` operation.

Here's the first of these declarations:

```
Declare Sub DragAcceptFiles Lib "SHELL" (ByVal hWnd As Integer, ByVal Accept As Integer)
```

`DragAcceptFiles` is a simple subroutine call that tells Windows whether the window indicated by the integer variable `hWnd` will or will not accept drag-drop messages, based on the value of the integer variable `Accept`. If `Accept` is -1 (True), then the window will accept drag-drop messages; if it is equal to 0 (False), it will not.

The next definition is

```
Declare Function DragQueryFile Lib "SHELL" (ByVal hDrop As Integer, ByVal IndexFilename As Integer, ByVal lpFileName As String, ByVal Buffsize As Integer) As Integer
```

`DragQueryFile` is a function call that returns an integer whose meaning varies according to the value of the `IndexFilename` parameter used to call the function. If you pass a value of -1 to `DragQueryFile`, it returns the number of files that were dropped onto the window. But if you pass it any value ranging from 0 to the total number of files that were dropped onto the window, it returns the number of bytes in the file name corresponding to that

value. That is, if `IndexFilename=4`, the `DragQueryFile` function will return the length of the name of the fifth file that was dropped on the window.

The other parameters to the `DragQueryFile` function are equally important. The `hDrop` parameter is an integer that identifies the Windows internal data structure used to store the file names from the current drag-drop operation. Windows identifies that structure to your application through the `wParam` parameter of the `WM_DROPFILES` message. The `lpFileName` parameter identifies a null-terminated string (a string ending with ASCII character 0, the null character) in which the application wants Windows to store the name of the dropped file. And the `BuffSize` parameter is an integer telling Windows the length, in bytes, of the null-terminated `lpFileName` string.

When you pass a string from Visual BASIC to a Windows API function that will modify the string, you must first create the string and ensure that it is long enough to hold whatever data Windows might place in it. As you'll see below, Recycler initializes the `lpFileName` string to a length of 129 characters.

The final drag-drop-related API definition is for the `DragFinish` subroutine:

```
Declare Sub DragFinish Lib "SHELL" (ByVal hDrop As Integer)
```

This subroutine call is used to tell Windows that it can discard the data structure it was using to hold file names from the last drag-drop operation. It takes a single parameter, an integer variable that corresponds to the value of `hDrop` obtained from the `wParam` parameter of the `WM_DROPFILES` message.

Calling DISKSTAT.DLL

Because Recycler also makes use of three functions from Art Krumsee's `DISKSTAT.DLL`, it needs to declare them here along with the Windows API functions. The declarations for `SetFileHidden`, `IsFileHidden`, and `FindFile` are as follows:

```
Declare Function SetFileHidden Lib "DISKSTAT.DLL" (ByVal fname As String, ByVal PlusMin As String) As Integer
```

```
Declare Function IsFileHidden Lib "DISKSTAT.DLL" (ByVal fname As String) As Integer
```

```
Declare Function FindFile Lib "DISKSTAT.DLL" (ByVal fName As String, ByVal Searchpath As String, ByVal Fullname As String) As Integer
```

The first declaration tells Visual BASIC that the `SetFileHidden` function can be found in the library called `DISKSTAT.DLL` (which must be on the DOS path for this call to work). The function takes two parameters, `fName` (the name of the file to be hidden or made visible) and `PlusMin` (either a plus sign to indicate that the file is to be hidden, or a minus sign to indicate that it should be visible), and returns an integer value that indicates whether or not the operation was successful. The value returned will be 0 if the operation

was successful, 2 if the file wasn't found, 3 if the path wasn't found, or 5 if DOS refused to provide access to the file.

The `IsFileHidden` function, which is used to determine whether a file is already hidden, takes only a single parameter—the name of the file whose `Hidden` attribute is to be checked—and returns an integer. The returned value will be 0 if the file is not already hidden, or -1 if it is.

The `FindFile` routine, which indicates whether a specified file can be found on a specified path, takes three parameters: the name of the file to find, the path to search, and the name of the string in which `DISK-STAT.DLL` should store the full path of the file if it finds it. The function returns an integer value that will equal 0 if the file was found, or a nonzero number if it was not.

Global Constant and Variable Definitions

The `GLOBAL.BAS` file concludes by defining a series of six global constants and variables. Defining these as global means that they will be available to routines throughout the application. The value of a global constant cannot be changed once it has been established. In contrast, any routine in the application can modify the value of a global variable, after which the new value will be available to every routine in the application.

Recycler's global constants are as follows:

```
Global Const True = -1
Global Const False = 0
```

Declaring the constants `True` and `False` as -1 and 0, respectively, allows you to use them to evaluate the results of functions that return a value of -1 or 0 to indicate success or failure. Thus, you can write:

```
IF IsWindowVisible(MyWindow%) = TRUE ...
```

which improves the readability of your application, compared to

```
IF IsWindowVisible(MyWindow%) = -1 ...
```

Next, come the four global variable definitions:

```
Global Handle As Integer
Global NewMessage As Msg
Global NameOfFile As String * 129
Global TotSize As Long
```

These declarations create the variable `Handle` (which is used to store the number by which `Windows` refers to `Recycler`'s main window) as an integer, `NewMessage` (used to store messages retrieved by `PeekMessage`) as a `Msg` type variable, `NameOfFile` (used to retrieve file names through the

DragQueryFiles function) as a string with a fixed length of 129 characters, and TotSize (which is used to keep track of the total size of all the files in the recycling bin) as a real or long variable.

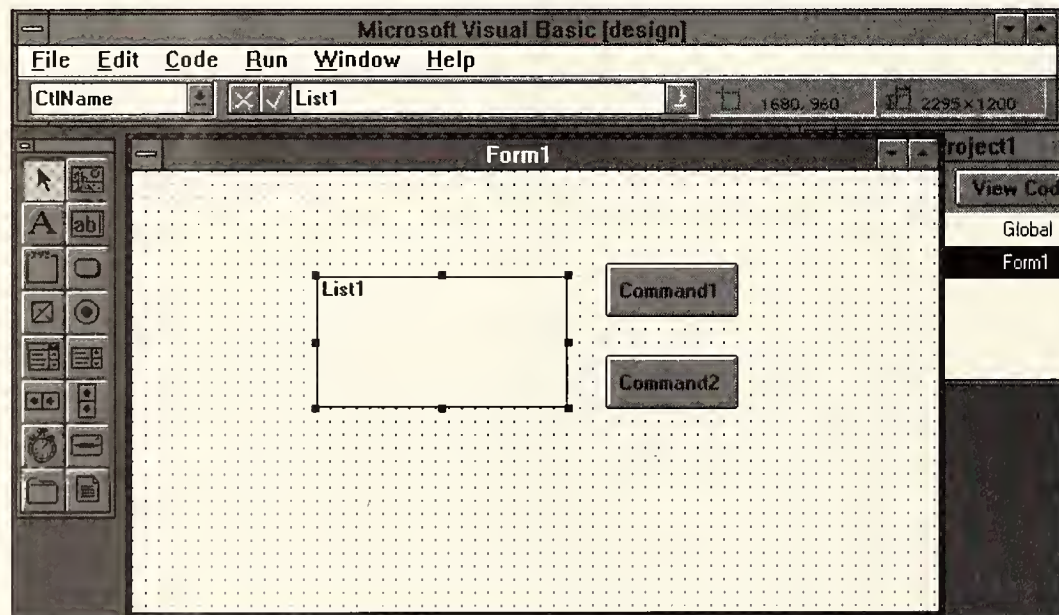
That concludes the tour of the GLOBAL.BAS file. Now let's look at the Form file, which constitutes Recycler's main window.

Exploring FORM1.FRM

Visual BASIC form files consist of a graphic design for a window or dialog box and the code that is attached to the elements of that design. You start off the process of creating a form by selecting the New Form option from Visual BASIC's File menu, and then start laying out user-interface elements on new blank form, by selecting them from Visual BASIC's Toolbox (a palette of available user-interface elements) and dragging them onto the form, as shown in Figure 12.7.

Figure 12.7

A newly created Visual BASIC form with button and list-box controls



Visual BASIC gives you control over a variety of properties for each element of the form design. For instance, you can control the size, location, and background and foreground colors of the form itself, along with its border style, caption, and whether it uses Minimize and Maximize buttons and a Control box. You can have similar control over the appearance and attributes of each user-interface element you create on the form.

In addition, Visual BASIC gives you the ability to write code that tells the application how to respond to a wide range of events for each user-interface

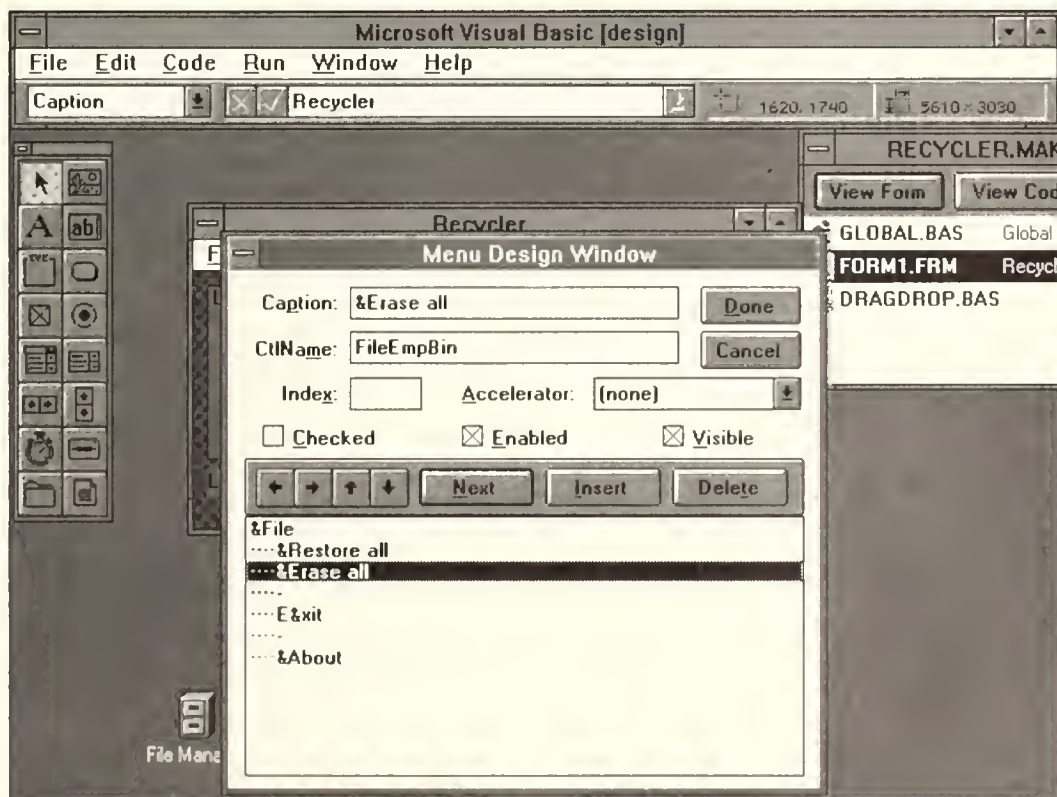
element. For instance, Visual BASIC offers eight unique events for standard command buttons: Click, GotFocus, LostFocus, KeyDown, KeyUp, KeyPress, DragDrop, and DragOver. (Visual BASIC's DragDrop and DragOver event procedures are internal to the application—allowing you to drag items within your application. They don't handle drag-drop messages from File Manager or even from other Visual BASIC applications.)

The form itself can react to events as well—everything from Resize (which occurs when you change the size of the form) to Unload (which occurs when you or the application closes the form).

The Recycler application includes a modest range of user-interface elements: a list box, a pair of command buttons, a label, and a drop-down menu. The list box, label, and buttons were created by selecting them from the Toolbox and drawing them on screen. The menu was created with the Menu Design window shown in Figure 12.8.

Figure 12.8

The Visual BASIC
Menu Design
window



Each menu item is treated as a unique user-interface control. Selecting a menu item triggers its Click procedure. For instance, the Exit item on the File menu is associated with a control called FileExit, so the FileExit_Click procedure is executed when that item is selected.

General Procedures

In addition to code that is directly associated with control events, a form may contain some code items that act as general-purpose subroutines, much like the routines in a code-only module, except that the general routines associated with a form are available for use with that form only.

Recycler's FORM1.FRM has only one general-purpose statement, the variable definition

```
DefInt A-Z
```

This statement tells Visual BASIC that any variable starting with any letter from A to Z should be treated as an integer unless otherwise specified. This eliminates the need to specifically declare variables to be of the type integer, while at the same time ensuring your application will never use a slow, inefficient real number variable unless you specifically tell it to do so.

Now for the event procedures in FORM1.FRM.

Event Procedures

The two buttons on the Recycler form are part of a single control array known as Command1. A *control array* is a group of controls that share the same name, type, and event procedures. Each element of the control array has a unique index number that can be used to identify it, and it may have other unique elements as well, such as its caption or size. Nevertheless, because the elements of a control array share common properties and procedures, they are much more efficient in terms of program size and system-resource use than a unique control for each user-interface element would be.

The Command1_Click Procedure

On Recycler's form, the Restore button has an index number of 0, and the Erase button has an index of 1. The Click event procedure for Command1 evaluates the index number of the item that was clicked, as seen here:

```
Sub Command1_Click (Index As Integer)
  LI = List1.ListIndex
  If LI < 0 Then Exit Sub
  FileToHide$ = List1.List(LI)
  Select Case Index
  Case Is = 0
    Temp = HideFile(FileToHide$, False)
  Case Is = 1
    Warn$ = "Do you really want to delete " + FileToHide$ + "?"
    Temp = MsgBox(Warn$, 20, "Recycler")
  IDNO=7
  If Temp = IDNO Then Exit Sub
```

```
Temp = HideFile(FileToHide$, False)
Kill FileToHide$
End Select
List1.RemoveItem LI
FixTitle
If List1.Listcount = 0 Then Recycler.Icon =
    LoadPicture("RECEMPTY.ICO")
LI = LI - 1
List1.ListIndex = LI
List1.Refresh
End Sub
```

The `Command1_Click` procedure begins by assigning the value of `List1.ListIndex` to the variable `LI`. The `ListIndex` property of a list box identifies which item in the list box is selected. `List1` is the name of the list of files in the recycling bin, so its `List1.ListIndex` property identifies which file is highlighted in `Recycler`'s list box.

Next, the routine evaluates the value of `LI` and, if it is less than 0 (indicating that no item is selected), exits the subroutine, ending processing of the `Click` event.

Otherwise, the routine assigns the value of the `List` property for item `LI` in the list box to the string variable `FileToHide$`. The `List` property identifies the text of a list-box item, so this command assigns the name of the selected file (as it appears in the list box) to `FileToHide$`.

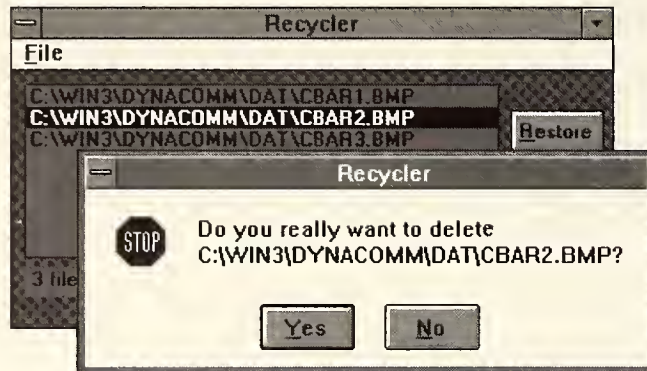
Next, the routine executes a `Select Case` statement that evaluates the value of `Index`, the integer variable in which Visual BASIC has placed the index number of the button the user clicked. If `Index` is equal to 0, then the user pressed the `Restore` button. In that case, the routine calls the `HideFile` subroutine (located in the `DRAGDROP.BAS` module, discussed below), passing it the name of the file to be restored and the value `False`, which tells `HideFile` to turn off the `Hidden` attribute for the file. Then the flow of execution jumps past the end of the `Select Case` routine, to the command `List1.RemoveItem LI`, which is discussed under "Removing the Selected Item," below.

In the meantime, if the variable `Index` was equal to 1, then the `Erase` button was selected. `Recycler` reacts by opening a message box on screen that asks the user to verify that permanent deletion of the selected file is desired, as shown in Figure 12.9.

The `MsgBox` statement takes three parameters: the message to be posted, the style of message box in which to post it, and the title of the message box. The message-box style is the total of the numeric values that Visual BASIC assigns to various message-box elements. In this case, the message box is style 20, since it includes both a stop-sign icon (value 16) and `Yes` and `No` buttons (value 4).

Figure 12.9

Recycler seeks confirmation before deleting any file



If the No button is pressed, Visual BASIC returns a value of 7, and so Recycler exits the Click procedure. Otherwise, it calls the HideFile subroutine to make the selected file visible, and then issues the Visual BASIC Kill command to delete the file (Kill won't work on a hidden file, so the file must be made visible before it can be deleted), thus concluding the Select Case procedure.

Removing the Selected Item Next, no matter which button was pressed, the routine issues the RemoveItem command to remove the selected item from the list box, then calls the FixTitle subroutine (in DRAGDROP.BAS) to update Recycler's file-count data. Then it checks to see if the ListCount property of the list box is equal to 0, in which case it resets the icon used to represent Recycler to that of an empty recycling bin, by loading the file REEMPTY.ICO. (RECFULL.ICO and REEMPTY.ICO are standard ICO-format icon files, which I created using hDC's Icon Designer. All icon files must be located in the same directory as the RECYCLER.EXE application.)

Finally, the routine decreases the value of LI by one, uses the ListIndex command to set the selection bar in the list box to the item that corresponds to LI, issues the Refresh command to refresh the contents of the list box, and then exits, ending processing of the button click.

The FileEmpBin_Click Procedure

FileEmpBin_Click is called when you select the Erase all item from Recycler's File menu. It begins by using a message box to verify that you actually want to erase all the files in the bin.

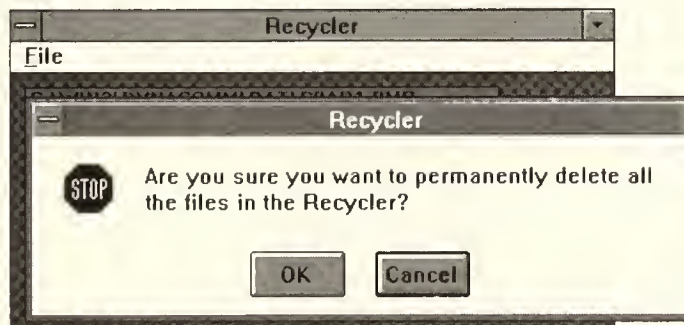
```
Sub FileEmpBin_Click )
    Mess$ = "Are you sure you want to permanently delete all
        the files in the Recycler?"
    Temp = MsgBox($Mess, 273, "Recycler")
```

```
If Temp = 2 Then Exit Sub
LI = List1.ListCount - 1
For X = 0 To LI
FileToKill$ = List1.List(X)
Temp=HideFile(FileToKill$, FALSE)
Kill FileToKill$
Next X
For X = 0 To LI
List1.RemoveItem 0
Next X
Recycler.Icon = LoadPicture("RECEMPTY.ICO")
FixTitle
End Sub
```

The routine uses a message box of type 273, meaning that it has a Stop icon (16), OK and Cancel buttons (1), and that the second button—the Cancel button—is the default (256). The box is shown in Figure 12.10.

Figure 12.10

The Erase all confirmation box



If the Cancel button is selected (Temp=2), the routine exits. Otherwise, it proceeds to erase all the files in the recycling bin. First it uses ListCount to obtain a count of the items in the list box, and then it subtracts 1 from it, because items in the list box are numbered starting with 0, so the first element in the list box would be identified as item 0, and the tenth element in the list box would be identified as item 9. Then, for each element in the list box, it repeats the deletion sequence used by the Delete button, first making the file visible, and then deleting it. Then it removes each item from the list box, changes Recycler's icon to indicate that the bin is empty, calls FixTitle to fix Recycler's window title and file count to show its newly empty state, and then exits.

The FileResAll_Click Procedure

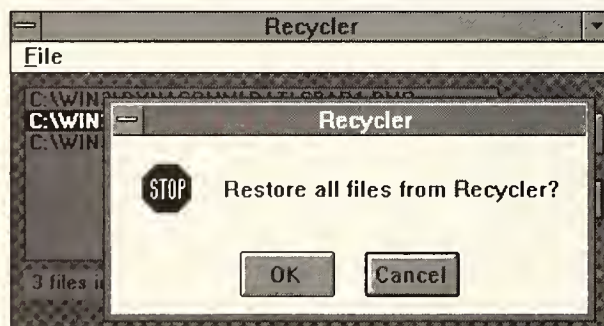
FileResAll_Click is called when you select the Restore all item from Recycler's File menu. It closely resembles the FileEmpBin_Click procedure:

```
Sub FileRecAll_Click ()
X = MsgBox("Restore all files from Recycler?", 273,
  "Recycler")
If X = 2 Then Exit Sub
LI = List1.Listcount
For X = 0 To LI - 1
FileToHide$ = List1.List(0)
N = HideFile(FileToHide$, False)
List1.RemoveItem 0
Next X
FixTitle
Recycler.Icon = LoadPicture("RECYCLE1.ICO")
End Sub
```

Like the Erase all procedure, the Restore all routine starts off by confirming the user's intent, using the message box shown in Figure 12.11.

Figure 12.11

The Restore all confirmation box



It continues by un hiding all the files in the bin and resetting the window's icon and title bar, using a series of steps that are nearly identical to the Erase all procedure. The only difference in the routines is the absence of a Kill FileToHide\$ statement in the Restore all procedure.

The Form_Resize Procedure

Form_Resize is called when the user resizes the form or minimizes it or restores it from a minimized state. I wanted the title of Recycler's window to change depending on whether it was in a minimized or restored condition, so I used the Resize event to call the FixTitle procedure in DRAGDROP.BAS:

```
Sub Form_Resize ()
```



```
FixTitle
End Sub
```

The FileAbout_Click Procedure

FileAbout_Click is called when the user selects the About Recycler item on the File menu:

```
Sub FileAbout_Click ()
x = MsgBox("Recycler by Paul Bonner. Copyright © 1992 by
  Paul Bonner. All Rights Reserved.", 64, "About
  Recycler")
End Sub
```

The message box produced by this procedure is shown in Figure 12.12.

Figure 12.12

The About Recycler message box



The FileExit_Click Procedure

FileExit_Click is called when the user selects the Exit item on the File menu:

```
Sub FileExit_Click()
Unload Recycler
End Sub
```

This procedure simply instructs Visual BASIC to unload the Recycler form (FORM1.FRM). This triggers the Form_Unload procedure.

The Form_Unload Procedure

The Form_Unload procedure performs the steps necessary to shut down the Recycler application:

```
Sub Form_Unload (Cancel As Integer)
If List1.Listcount > 0 Then
MsgBox "Please either erase files or restore all files
  before closing the Recycler", 272, "Recycler"
Cancel = True
```

```
Exit Sub
End If
End
End Sub
```

The procedure begins by examining the value of List1.Listcount. If it is greater than zero, then there are still files in the Recycler, so the procedure posts an error message and halts the shutdown process. Otherwise, it simply issues an End command to halt execution of the program.

That concludes the FORM1.FRM file. Now let's examine the routines in the DRAGDROP.BAS file.

Exploring DRAGDROP.BAS

Like FORM1.FRM, DRAGDROP.BAS starts off by defining all unsigned variables as integers, using the statement

```
DefInt A-Z
```

Beyond that general definition, DRAGDROP.BAS consists of five named subroutines and functions: Main, HideFile, CheckForDir, FixTitle and FixSize.

The Main Subroutine

I used the Set Startup Form item on the Visual BASIC Run menu to designate Main as the startup procedure for the Recycler application, making it the first code executed when the application launches.

Main starts off by instructing Visual BASIC to display the Recycler window. Then it issues the FixTitle command to make sure that the correct title is displayed for the window, and uses the hWnd function to obtain the Windows handle for the Recycler window:

```
Sub Main ()
Recycler.Show
FixTitle
Handle = Recycler.hWnd
FileNum = -1
PM_NOREMOVE = 0
PM_NOYIELD = 2
wRemoveMsg = PM_NOREMOVE Or PM_NOYIELD
DragAcceptFiles Handle, True
Do While DoEvents()
X = PeekMessage(NewMessage, Handle, 563, 563, wRemoveMsg)
If X <> 0 Then
Screen.Mousepointer = 11
```

```

Beep
X = DragQueryFile(NewMessage.wParam, FileNum, NameOfFile,
  128)
For Counter = 0 To X - 1
Y = DragQueryFile(NewMessage.wParam, Counter, NameOfFile,
  128)
If IsFileHidden(NameOfFile) = 0 Then
N = HideFile(NameOfFile, True)
If N = 0 Then
Recycler.List1.AddItem NameOfFile
Else
Fail$ = NameOfFile
MsgBox ("Couldn't add " + Fail$): Fail$ = ""
End If
If Recycler.List1.Listcount = 1 Then Recycler.Icon =
  LoadPicture("RECYCLE2.ICO")
End If
Next counter
FixTitle
DragFinish NewMessage.wParam
Screen.Mousepointer = 0
End If
Loop
End Sub

```

Once those introductory steps are complete, the routine assigns the value of -1 to the variable FileNum, and uses the Or operator to combine two standard parameters for the PeekMessage command: PM_NoRemove (which instructs Windows not to remove the message from the Windows queue) and PM_NoYield (which prevents Visual BASIC from yielding control to another task while the PeekMessage statement is being executed). Then it calls the Windows DragAcceptFiles subroutine, passing it the handle of the Recycler window as the window that will accept drag-drop messages.

Next the Main routine enters the DoEvents loop, which allows it to monitor drag-drop messages. This loop is performed constantly while Recycler is active. It begins by calling PeekMessage to determine whether a message of type 563 (WM_DROPFILES) is waiting for Recycler. If the value returned by PeekMessage does not equal 0, then a WM_DROPFILES message is waiting. Recycler responds to this by turning the cursor into an hourglass to indicate that it is busy (Screen.MousePointer=11), then beeps the PC speaker, and calls the DragQueryFile function.

Since FileNum has been set equal to -1, the DragQueryFile function will return the number of files that were dropped on the window. Then the routine loops through the list of files that were dropped on the window, obtaining

each file's name in turn, and issuing the command to hide it. If the hide operation is successful ($N=0$), the routine adds the name of the file to the list box on the Recycler window.

If, on the other hand, Recycler was unable to hide the file, it informs the user via the message box shown in Figure 12.5. The most common reason for Recycler being unable to follow through is that the item it's attempting to hide isn't a file at all, but rather a subdirectory.

Next, the routine checks the total number of items in List1, and if it is equal to 1, loads the RECFULL.ICO icon to show that Recycler is not empty. Then it loops back to get the next file.

Once all the file names have been retrieved and all the files have been hidden, Recycler calls the FixTitle routine to fix the window caption, sends the DragFinish message to release the Windows data structure reserved for the drag-drop operation, turns the cursor back to its standard shape, and, finally loops back to the beginning of the DoEvents loop and starts over again looking for the WM_DROPFILES message.

Now let's look at the function that actually hides the files dropped onto Recycler.

The HideFile Function

HideFile uses DISKSTAT.DLL's SetFileHidden routine to either hide a file or make a file visible. The function takes two parameters: FileToHide\$ (the name of the file to hide or make visible) and Hiding (an integer variable that indicates whether the file should be hidden or made visible):

```
Function HideFile (FileToHide$, Hiding)
  If Hiding = True Then PlusMin$ = "+": Else PlusMin$ = "-"
  N = CheckForDir(FileToHide$)
  If N <> 0 Then HideFile = 1: Exit Function
  HideFile= SetFileHidden(FileToHide$, PlusMin$)
End Function
```

The HideFile function starts by evaluating the value of the Hiding parameter. If it is True (equal to -1 , per the Constant definition of True in GLOBAL.BAS), then the string variable PlusMin\$ is set to "+". Otherwise, it is set to "-".

Next, the HideFile function calls the CheckForDir function (described next), passing it FileToHide\$ and assigning the result of the function to the integer variable N. CheckForDir ensures that the file HideFile is about to hide is indeed a file, and not a directory. If N contains a nonzero value, then FileToHide\$ is a directory or is otherwise unavailable, and so HideFile assigns the value 1 to its own return value and returns to the procedure that called it. (The Main routine examines the value returned by the HideFile function, and posts the message box shown in Figure 12.5 if the value is not equal to 0.)

Otherwise, HideFile calls the SetFileHidden function, passing it the name of the file to hide and the string variable PlusMin\$, and assigns the results of SetFileHidden to its own result before returning to the routine that called it. Again, the value of HideFile will be 0 if the SetFileHidden function was successful, or another value if it failed.

The CheckForDir Function

CheckForDir is used by the HideFile function to determine whether a file that is about to be hidden or made visible is available for that operation. This is necessary because the DISKSTAT.DLL does not return an error if you try to set the Hidden attribute for a directory or subdirectory, even though doing so has no effect on the directory or subdirectory.

```
Function CheckForDir (FileToHide$)
Searchpath$ = Environ$("PATH")
Result$ = String$(129, 0)
CheckForDir = FindFile(FileToHide$, Searchpath$, Result$)
End Function
```

CheckForDir determines whether the file name being tested points to a file or to a directory by using the FindFile function from DISKSTAT.DLL. If you ask FindFile to find a fully qualified file name for an existing file (such as C:\VB\RECYCLER.EXE), it will always report that it can find it. But if you pass it the full path to a directory entry (such as C:\VB\SAMPLES), it will always report that the file cannot be found. So the FindFile function turns out to be a good way to determine whether a name supplied by a drag-drop function points to a file or to a directory.

The CheckForDir function begins by setting the string SearchPath\$ equal to the DOS path, using Visual BASIC's Environ\$ function to obtain the path. Then it initializes Result\$ as a 129-character null-terminated string, and then calls the FindFile function, passing it the variables FileToHide\$, SearchPath\$, and Result\$. If the file specified by FileToHide\$ is indeed a file, the function will return a value of 0, indicating that the file was found. But if it is a directory, the function will return a nonzero value, indicating that the Recycler shouldn't add FileToName\$ to the recycling bin.

The FixTitle Subroutine

FixTitle is called during Recycler's loading process, and then again whenever the user places files in an empty recycling bin, empties the bin, or minimizes or restores Recycler's window. This routine is responsible for making sure that Recycler's window title and icon caption reflect its current state: the window title should simply read "Recycler" when the window is in a restored state, but the caption should be changed to indicate the number of files and their total size (in kilobytes) when Recycler is minimized. In addition, FixTitle ensures

that the text that appears below the list box in the Recycler window always correctly reflects the number of files in Recycler and their total size in bytes.

```
Sub FixTitle ()
  FixSize
  KB& = TotSize / 1024
  LI = Recycler.List1.Listcount
  Recycler.Label1.Caption = Str$(LI) + " files in Recycler."
    " + Str$(TotSize) + " bytes"
  If LI = 0 Or Recycler.Windowstate <> 1 Then
    Recycler.Caption = "Recycler": Exit Sub
  If LI = 1 Then Recycler.Caption = "Recycler 1-file" +
    Str$(KB&) + " KB": Exit Sub
  Recycler.Caption = "Recycler " + Str$(LI) + "-files" +
    Str$(KB&) + " KB"
End Sub
```

FixTitle starts by calling the FixSize subroutine (detailed below), which sets the global variable TotSize equal to the total size in bytes of all the files in the recycling bin. Then FixTitle divides that total by 1024 to obtain the total size in kilobytes of the files in the bin, and assigns that result to the real number KB&.

FixTitle then uses the ListCount command to count the files in List1 on the Recycler form, and resets the caption of the label on the Recycler form to indicate the number of files and their size in bytes.

Next, FixTitle sets the window title (the caption of the Recycler form), setting it to read "Recycler" if the bin is empty or if the window state of the Recycler window does not equal 1 (the window is not minimized). Alternatively, it will be set to "Recycler 1-file " plus its size in kilobytes, if the minimized Recycler contains a single file; or "Recycler", followed by the total number of files in the bin, followed by "-files " and their total size in kilobytes, if the minimized Recycler contains more than one file.

Once it has set the window title, FixTitle exits, returning control to the routine that called it.

The FixSize Subroutine

FixSize, the last routine in Recycler, simply counts the total size in bytes of all the files in the Recycler bin:

```
Sub FixSize ()
  TotSize = 0
  For X = 0 To Recycler.List1.Listcount - 1
    Temp=FreeFile
    F$=Recycler.List1.List(X)
    If CheckForDir (F$) <> 0 Then Goto Skip
```



```
Open Recycler.List1.List(X) For Input As Temp
TotSize = TotSize + LOF(Temp)
Close Temp
Skip
Next X
End Sub
```

FixSize starts by setting the global variable TotSize equal to 0. Then, for each file in the bin, it obtains a free file handle, checks to make sure the file exists, and if it does, opens the file and uses the Visual BASIC LOF file to obtain its length, adding that value to the current value of TotSize. (It has to open the file because LOF will only work on an open file.) Then it closes the open file and goes on to the next file in the bin. When it has added the size of each file to TotSize, it exits, returning control to the FixTitle routine, which called it.

Wrapping Up Recycler

Recycler wasn't the most challenging project in the world; it's a nice little application, but it doesn't do all that much. Moreover, there are a couple of factors that keep it from being of commercial grade: its inability to work with subdirectories and to keep track of files in the recycling bin from session to session.

Still, those limitations could be easily overcome. You could circumvent the inability of DISKSTAT.DLL to hide subdirectories by having Recycler move files to a designated Recycler subdirectory rather than hiding them, and use an array to keep track of their original location. And you could "train" Recycler to keep track of its contents from session to session by having it save that array to disk every time it shuts down, and reload the array when it starts up.

The point of the Recycler project, however, wasn't to provide the world's best recycling bin. Instead, it was to demonstrate the ease with which you can extend Windows development tools using dynamic link libraries—and the power those libraries can give your applications. Think about it: Visual BASIC doesn't provide built-in support for either of the two crucial functions performed by Recycler—accepting drag-drop messages and hiding files. Yet by making a few external-library declarations, it was a simple matter to build an application in Visual BASIC that revolved around those capabilities. With this kind of extensibility, there are almost no limits to the power of any Windows development tool that can interact with external dynamic link libraries.

Next, in Chapter 13, you'll see another kind of extensibility: that provided by Dynamic Data Exchange.

C H A P T E R

13

Linking Applications through DDE— Windows Broker

*Broker's Origin and
Structure*

*Exploring
BROKER1.XLS*

*Exploring LOTS.XLS
and IBM.XLS*

*Exploring the
BROKER1.XLM File*

*Exploring
BROKER1.DCP*

*Wrapping Up Windows
Broker*

WINDOWS BROKER IS AN EXAMPLE OF HOW WINDOWS'S DYNAMIC Data Exchange (DDE) facility and application macro languages can be combined to form a custom application that joins the capabilities and features of two or more existing applications. Windows Broker links Microsoft's Excel spreadsheet program and FutureSoft Engineering's DynaComm asynchronous communications program to produce an automated stock-trading system that can track the values of the stocks in a portfolio and issue electronic buy and sell orders to an on-line discount brokerage.

Broker's Origin and Structure

One of the more common examples used to illustrate the benefits of Windows DDE is that of automating the downloading of on-line financial data into a spreadsheet-based portfolio-analysis application. That always sounded like a great idea to me, but I'd never actually seen it done. So I decided to test the practicality of the notion once and for all by creating a broker for Windows.

And I decided to push the envelope a little while I was at it. It would have been easy to make Windows Broker simply track changes in the value of a portfolio by downloading stock-price information. But I also wanted it to automate the process of buying and selling stocks through an on-line discount brokerage, so that I could issue a buy or sell order at the push of a button.

To keep the scope of the project manageable, however, I limited it to managing a fictitious portfolio with holdings in only two stocks: IBM and Lotus Development Corporation.

Choosing the Tools

There was no contest in the choice of a spreadsheet in which to build Windows Broker. Most experts agree that Microsoft's Excel is so far superior to the competition in terms of programmability and customizability that I knew from the start it would serve as the core component of the Windows Broker project. (The application was originally built in Excel 3.0, but it can also be used with the new Excel 4.0.)

The choice of a communications tool for use with Windows Broker was a little less obvious, because the application's communications functions could have been handled by any Windows asynchronous communications package with a decent macro language and support for DDE. That includes, among others, Crosstalk Communication's Crosstalk for Windows, Hi-Q's Mission Control, and SynappSys's WinComm. In the end, however, I selected FutureSoft Engineering's DynaComm program for use in this part of the Windows Broker application, mostly because I was already favorably impressed (and familiar) with its macro language from other development projects.

I also had to select a source for electronic stock prices and an on-line brokerage. I found both on CompuServe. Windows Broker uses CompuServe's Quick Quote service to obtain daily stock price data, and Quick & Reilly's Quick*Way on-line brokerage (accessible through CompuServe) to issue buy and sell orders for securities in my fictitious portfolio.

Quick*Way offers a "game account" feature that allows you to set up imaginary portfolios and buy and sell stocks in them, following your gains and losses without actually ever buying a real share of stock. The game account turned out to be awfully useful as I developed Windows Broker, since it allowed me to avoid having to buy and sell real shares of stock just to test the script.

Application Framework

The Windows Broker application consists of five files:

- BROKER1.XLS, an Excel spreadsheet that is used to analyze and present data about the portfolio's holdings
- IBM.XLS and LOTS.XLS, a pair of worksheets used to hold daily stock-price and sales-volume information for IBM and Lotus Development Corporation
- BROKER1.XLM, an Excel macro sheet used to hold the spreadsheet macros that power the Windows Broker application
- BROKER1.DCP, a DynaComm macro file that performs all Windows Broker's on-line activities

Let's look at each of these, starting with the BROKER1.XLS file.

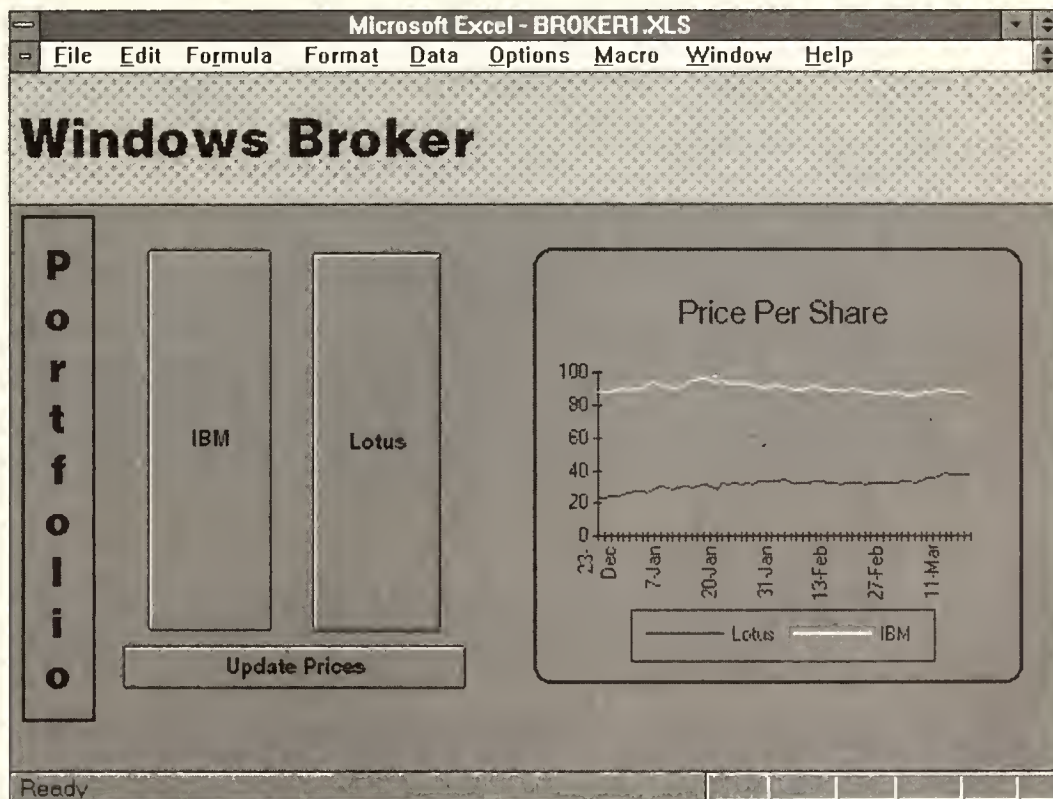
Exploring BROKER1.XLS

BROKER1.XLS is the control center of the Windows Broker application, and the only part of the application with which the user interacts. It keeps track of trades in the portfolio, recording the date of each transaction, the security purchased or sold, the number of shares traded, the price paid or received per share, and the commission paid on the trade.

In addition, BROKER1.XLS contains nine separate screen "pages," self-contained areas of the worksheet, each of which fills the screen with a custom display. These screen pages (discussed below) present information about the portfolio in numeric or graphic form and elicit user input through on-screen buttons and data entry fields. Except for the Opening screen, there are separate versions of each screen for IBM and Lotus stocks.

Opening Screen This screen, shown in Figure 13.1, presents a line chart illustrating the closing-price performance of both securities in the portfolio for the past 60 trading days, plus three buttons: one labeled “Lotus”, which jumps to the Lotus Stock screen; one labeled “IBM”, which jumps to the IBM Stock screen; and one labeled “Update Prices”, which launches Dyna-Comm and instructs it to obtain the latest price and volume information for both securities from CompuServe.

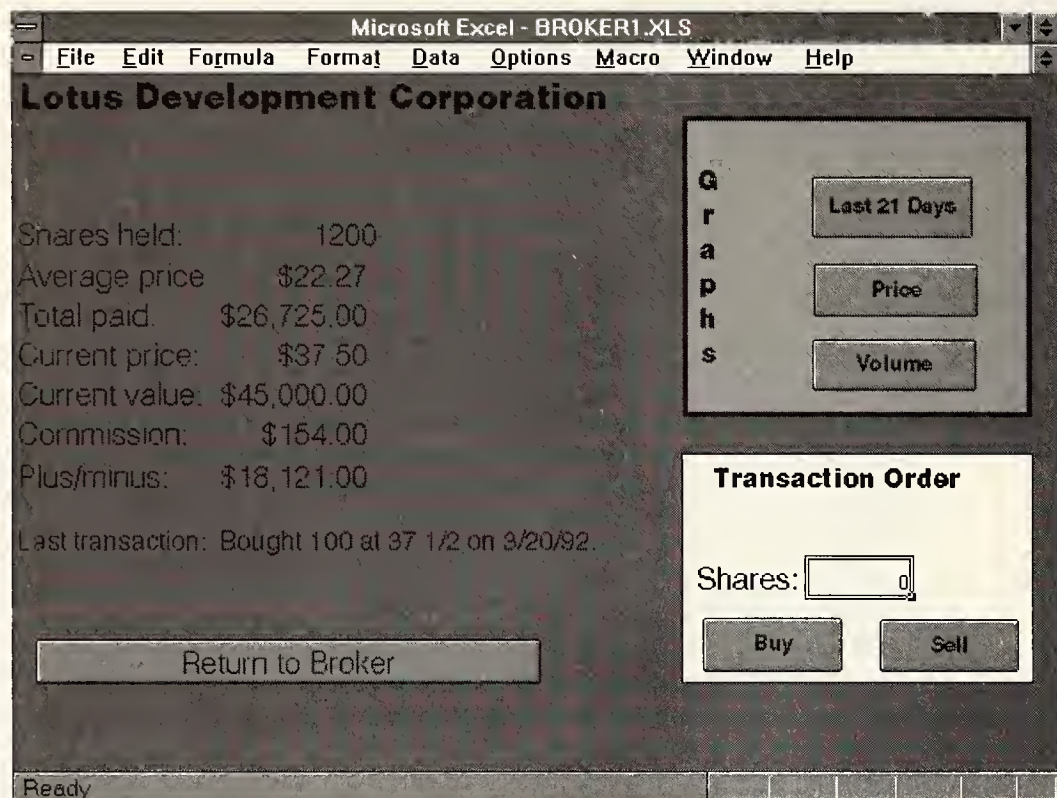
Figure 13.1
Windows Broker's
Opening screen



Stock Screens The Stock screens detail the user's holdings in the current stock, including the total shares held, average price per share, total price paid, current price per share, current value of the portfolio, total commissions paid, and a plus/minus figure that indicates the user's total gain or loss on the security. In addition, each Stock screen presents a summary of the user's last transaction of the stock, buttons for jumping to three separate graphs illustrating various aspects of the stock's performance, and a small Transaction Order form that allows the user to enter a buy or sell order. Figure 13.2 shows the Lotus Stock screen.

Figure 13.2

The Lotus Stock screen



21-Day High-Low-Close Chart These screens (the Lotus version of which is shown in Figure 13.3) illustrate the high, low, and closing prices for the stock over the previous 21 trading days.

60-day Price Chart These screens illustrate the closing price for the stock over the previous 60 trading days, as shown in Figure 13.4.

60-Day Volume Chart These screens, as Figure 13.5 shows, illustrate trading volume for the stock over the previous 60 trading days.

Worksheet Mechanics

The Windows Broker screens take advantage of Excel's strong graphic capabilities and its ability to place buttons and graphs directly on the worksheet. They also utilize its ability to hide many standard spreadsheet features, including the formula bar, Toolbar, scroll bars, and cell gridlines.

On the Opening screen (Figure 13.1), I created the Windows Broker headline and the Portfolio bar using Excel's word processing tool, which allows you to draw a text box directly on the worksheet and to specify its font, fill color, border, and orientation (horizontal or vertical). I then used Excel's button tool to create the Lotus, IBM, and Update Prices buttons and to link them to macros on the BROKER1.XLM macro sheet. Figure 13.6 shows the Excel 4.0 Properties menu used to make this linkage.

Figure 13.3

The Lotus 21-Day High-Low-Close chart

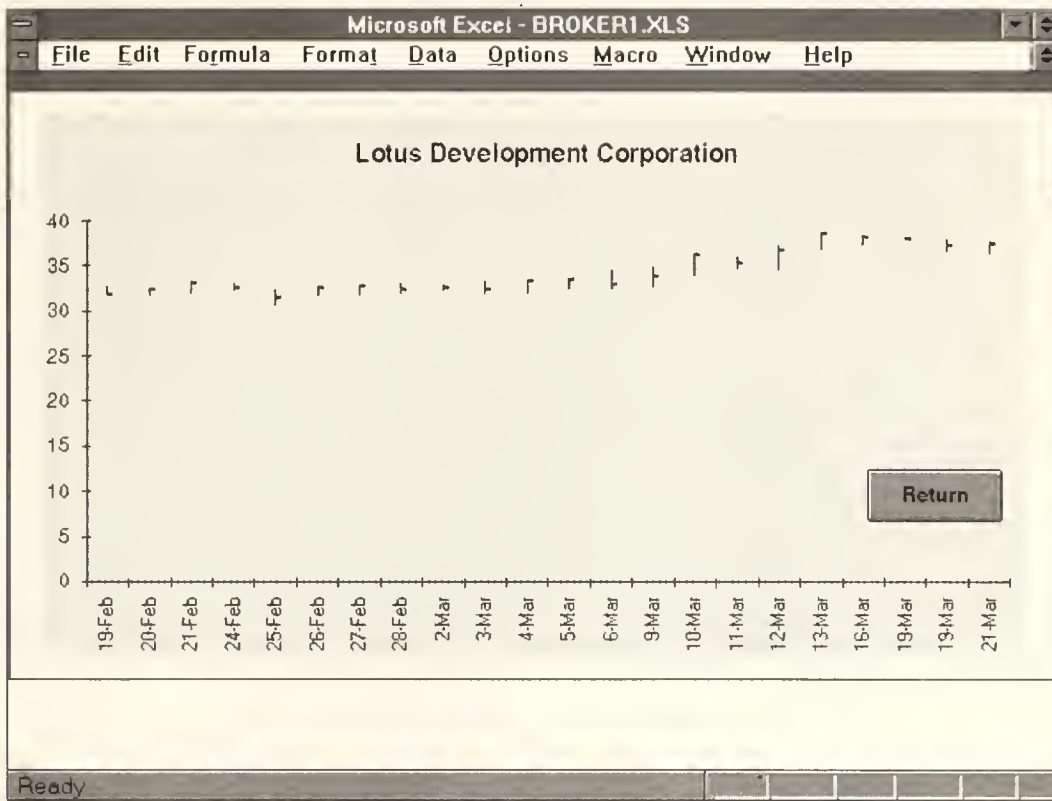


Figure 13.4

The Lotus 60-Day Price chart

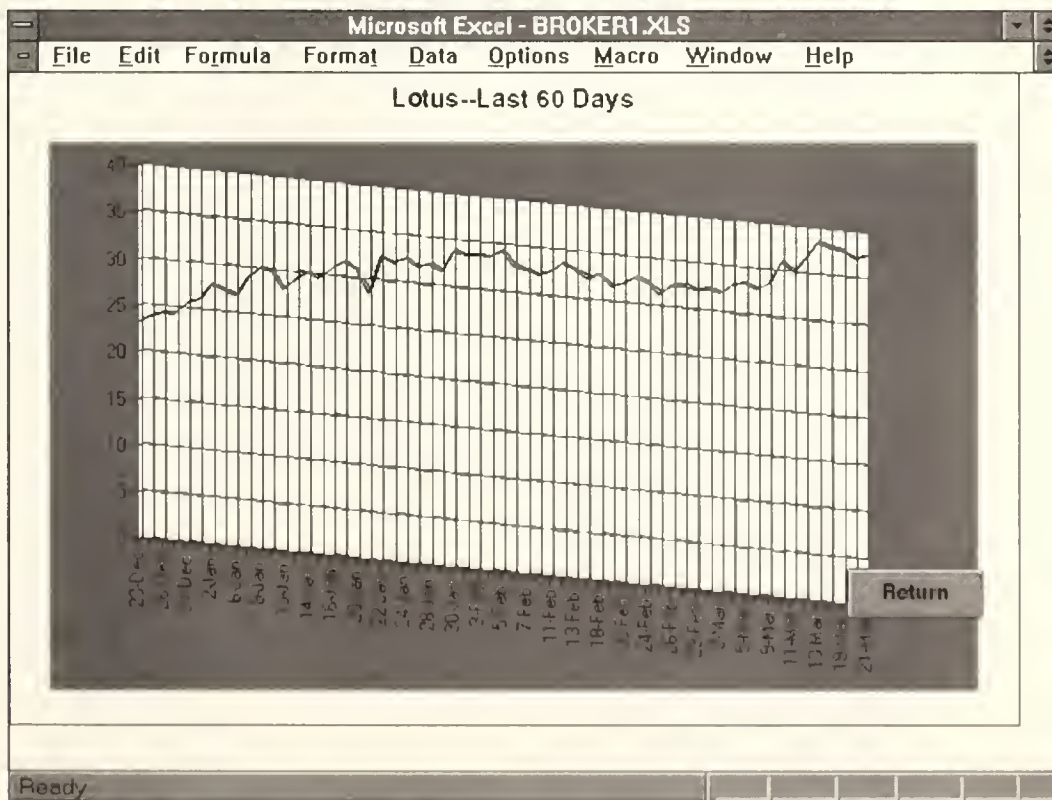


Figure 13.5

The Lotus 60-Day Volume chart

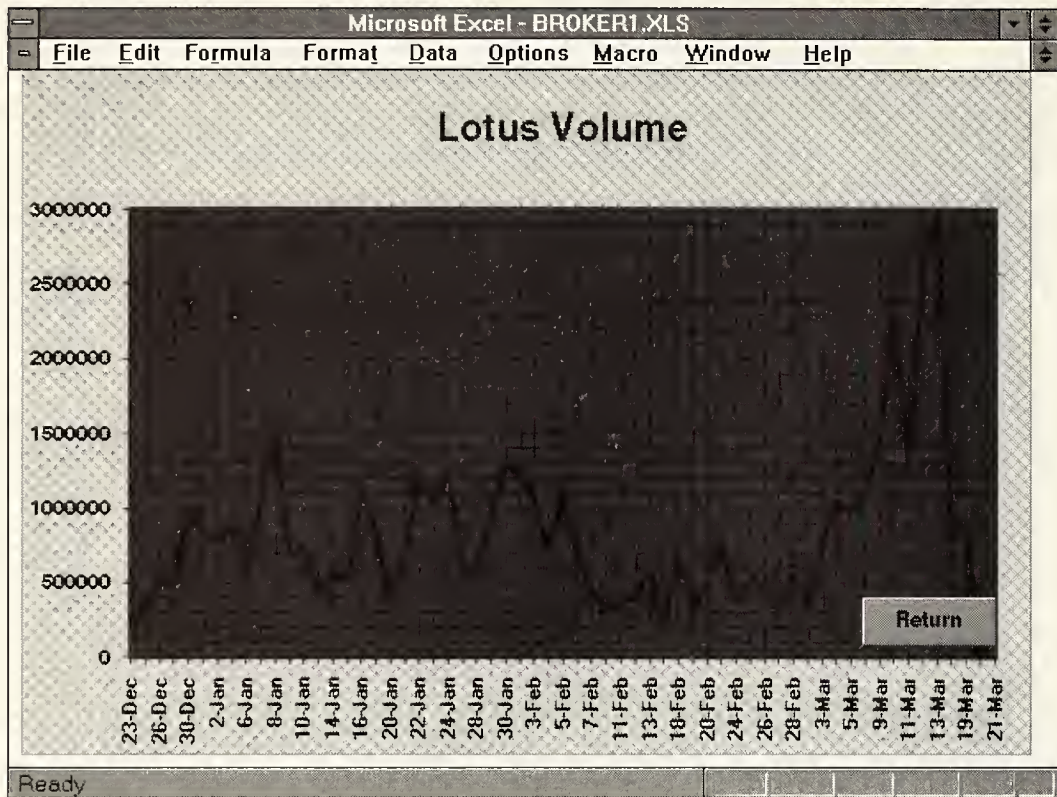
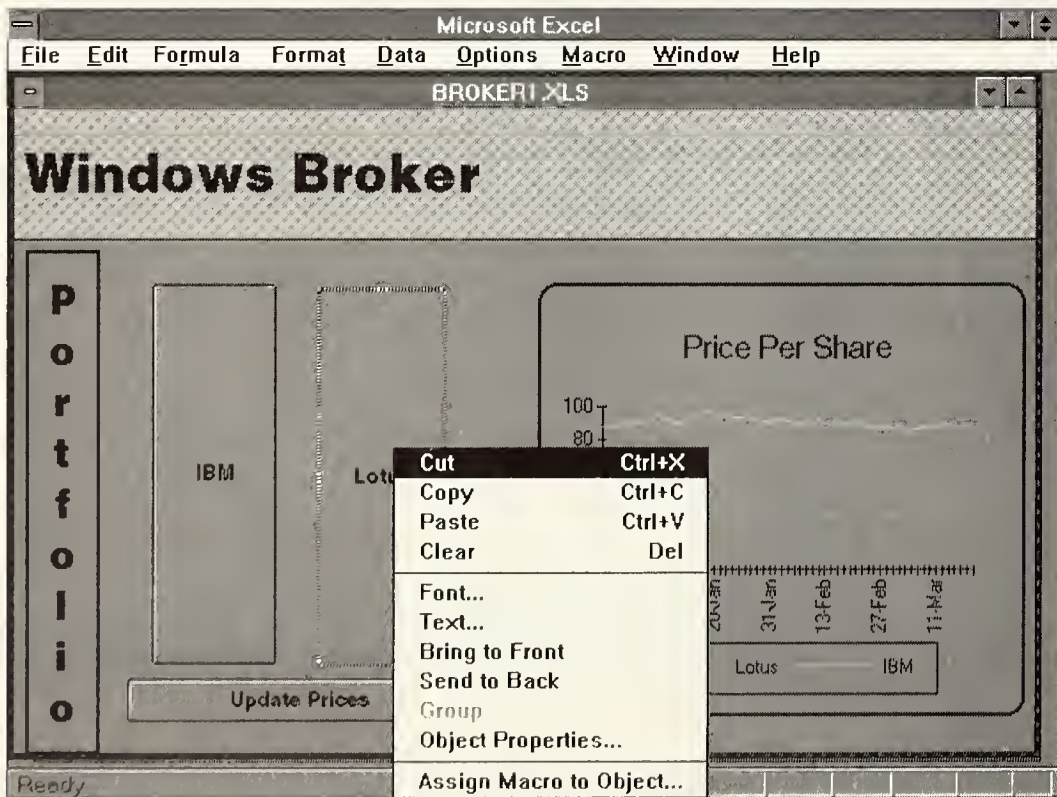


Figure 13.6

The Properties menu for a worksheet button

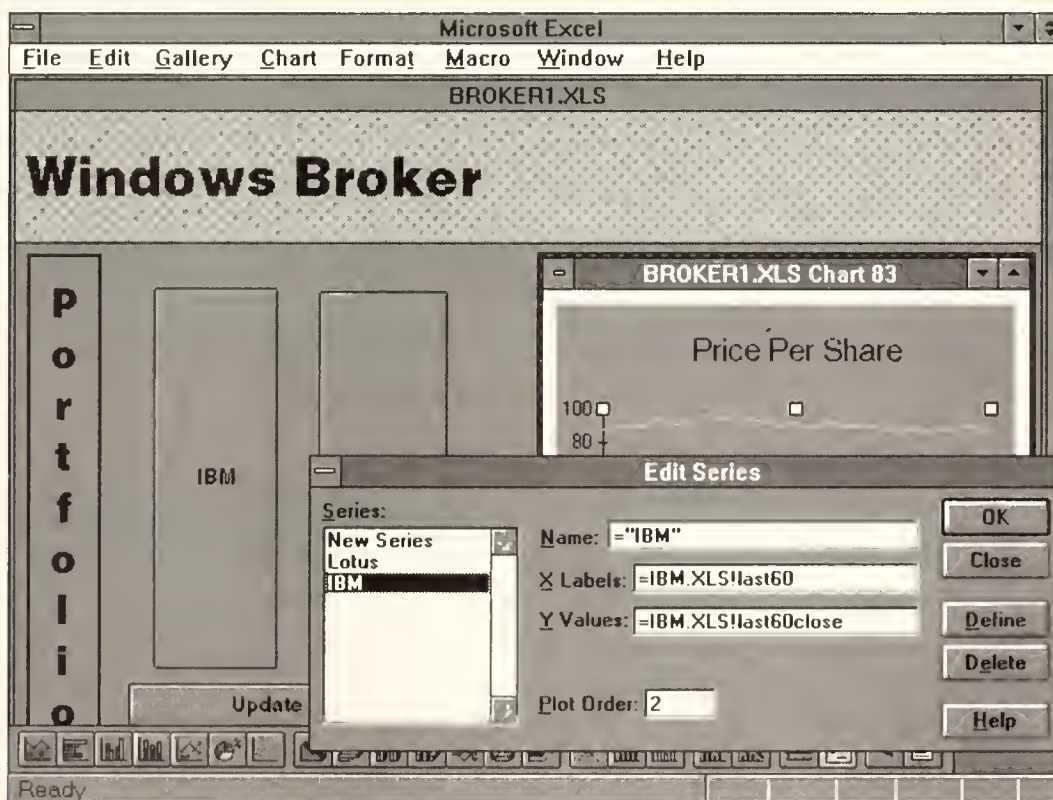


The Opening screen price chart, which graphs changes in the closing prices of both stocks in the portfolio over the past 60 trading days, is linked to the LOTS.XLS and IBM.XLS files. In each of those files, I created a named range called Last60Close, which contains the relevant prices, and one called Last60, which contains the date each price was recorded. The price chart utilizes both of them. It uses the Last60 range to supply the labels for its x-axis, and the Last60Close range to supply its y-axis variables, as shown in the Edit Series dialog box used to define the chart (see Figure 13.7).

The two Stock screens, and their charts, were built using the same methods.

Figure 13.7

The Price chart reflects ranges in the LOTS.XLS and IBM.XLS files



Hidden Data

The BROKER1.XLS file also contains two data ranges that the user won't normally see. The first is hidden behind the Windows Broker headline on the Opening screen. I've used the cells that are obscured by the headline to store a variety of information the DynaComm script needs to access, as detailed in Table 13.1.

The use of each of the items is explained later in this chapter in the section "Exploring BROKER.DCP."

Table 13.1 Data Needed by DynaComm

Cell	Contents
A1	CompuServe telephone number
B1	CompuServe log-on name
C1	CompuServe password
D1	Quick & Reilly account number
E1	Quick & Reilly log-on name
F1	Quick & Reilly password
G1	Quick & Reilly portfolio name
A2	Stock symbol (for transactions)
B2	"B" for Buy or "S" for Sell
C2	Quantity to buy or sell

Transaction Histories

The other hidden data in the worksheet consists of the transaction histories for the two accounts. These are located in the cells immediately below those occupied by the Volume charts, and are named IBMTrans and LOTSTrans, respectively. These ranges—which include the details of each transaction made in the portfolio, and keep track of total shares, profit and loss, and other data—are maintained automatically by the macros that handle stock transactions. Figure 13.8 shows the IBM transaction history.

Exploring LOTS.XLS and IBM.XLS

The LOTS.XLS and IBM.XLS files simply hold the daily price and volume data for Lotus and IBM stock, respectively. Each includes five columns of data, identified by the column headings Date, High, Low, Close, and Volume, as shown in Figure 13.9.

Both worksheets also contain a series of named ranges (listed in Table 13.2) used by the graphs in BROKER1.XLS and the macros in BROKER1.XLM to extract the data they need.

Figure 13.8

The IBMTrans range of BROKER1.XLS

The screenshot shows a Microsoft Excel window titled "Microsoft Excel - BROKER1.XLS". The menu bar includes File, Edit, Formula, Format, Data, Options, Macro, Window, and Help. The main area contains summary statistics for IBM shares:

Total Shares;	450
Average Price	\$90
Total paid	\$40,406
Current Value	\$39,094

Below the summary is a table titled "IBM Transactions" with the following columns: Date, Bought, Sold, Price, Paid, Received, and Co. The data rows are as follows:

Date	Bought	Sold	Price	Paid	Received	Co
3/3680	300	0	90	27000	0	21
3/17/92	100	0	88 1/4	8825	0	\$1
3/17/92	100	0	88 1/4	8825	0	\$2
3/17/92	100	0	88 1/4	8825	0	\$2
3/17/92	200	0	88 1/2	17700	0	\$1
3/17/92	0	250	87 5/8	0	21906	\$1
3/17/92	0	250	87 7/8	0	21969	\$2
3/18/92	150	0	87 3/8	13106	0	\$8

The status bar at the bottom shows "Ready".

Figure 13.9

Price data from the IBM.XLS worksheet

The screenshot shows a Microsoft Excel window titled "Microsoft Excel - IBM.XLS". The menu bar includes File, Edit, Formula, Format, Data, Options, Macro, Window, and Help. The main area displays a table of price data for IBM shares from November to December 1992. The columns are labeled A through J, with data starting from column B. The table has the following structure:

	A	B	C	D	E	F	G	H	I	J
6		Date	High	Low	Close	Vol				
7		27-Nov	114 1/25	112 5/8	113 1/2	15315				
8		28-Nov	113 8/75	112 3/8	112 3/8	13192				
9		29-Nov	112 8/75	111 5/8	112	11957				
10		30-Nov	114 5	111 1/4	113 5/8	18044				
11		3-Dec	114 5	113 1/8	113 3/8	15702				
12		4-Dec	115	113	114 3/4	14577				
13		5-Dec	114 8/75	113 3/4	114 5/8	14845				
14		6-Dec	116	111 1/4	111 1/2	33898				
15		7-Dec	112 8/75	110 3/4	112 1/2	19094				
16		10-Dec	113 7/5	112 1/8	113 3/8	11738				
17		11-Dec	113 1/25	112 3/8	112 7/8	9979				
18		12-Dec	114 5	112 5/8	114 3/8	14719				
19		13-Dec	114 6/25	112 7/8	112 7/8	10931				
20		14-Dec	112 5	111	111 1/4	14352				
21		17-Dec	111 7/5	110 5/8	111 1/2	9549				
22		18-Dec	113 5	111 1/2	113 1/2	16171				
23		19-Dec	114	112 1/8	112 3/4	24639				
24		20-Dec	114 6/25	112 1/4	113 3/4	18960				
25		21-Dec	114 3/75	113 3/8	113 7/8	20846				
26		24-Dec	114 2/5	113 3/4	113 7/8	3413				
27		26-Dec	114 2/5	113 1/2	113 1/2	3968				
28		27-Dec	113 8/75	113	113 5/8	7141				
29		28-Dec	113 8/75	113 1/4	113 3/8	7282				

The status bar at the bottom shows "Ready".

Table 13.2 **Named Ranges in LOTS.XLS and IBM.XLS**

Range	Contents
LAST60	Dates of last 60 prices
LAST60CLOSE	Closing prices for last 60 dates
LAST60VOL	Trading volumes for last 60 dates
LAST21	Dates of last 21 prices
LAST21HIGH	Daily high prices for last 21 dates
LAST21LOW	Daily low prices for last 21 dates
LAST21CLOSE	Closing prices for last 21 dates
LAST21VOL	Trading volume for last 21 dates
LASTPRICE	Most recent closing price

I started off the Windows Broker project by seeding these worksheets with 60 days of historical trades that I'd obtained through a standard interactive session with the CompuServe Quick Quote service. They are now maintained automatically: When you click the Update Prices button, a combination of Excel and DynaComm macros (in BROKER1.XLM and BROKER1.DCP) obtain the latest prices, add them to the bottom of the data tables in the two worksheets, and redefine the ranges to include the latest data.

Exploring the BROKER1.XLM File

An XLM file is referred to in Excel as a *macro sheet*. Generally, an XLM file is used to store macros associated with an XLS (worksheet) file that has the same name as the macro sheet. So BROKER1.XLM stores the macros associated with BROKER1.XLS.

Navigational Macros

The BROKER1.XLM file starts off with a series of scripts that move the active spreadsheet window from one screen page of the application to another in response to the user's commands. The Go_Lots macro leads things off.

The Go_Lots Macro

Go_Lots is executed when the user clicks the Lotus button on Windows Broker's Opening screen, or when the user clicks the button labeled "Return" on any of the three Lotus graph screens.

```
Go_Lots
=HSCROLL(15,TRUE)
=VSCROLL(1,TRUE)
=FORMULA("LOTS",!A2)
=FORMULA("=0",!$S$12)
=DEFINE.NAME("BuyTarget",!$S$12)
=DEFINE.NAME("TransTarget",!$P$11)
=IF(BROKER1.XLS!TransTarget="Unrecorded")
=GOTO(Update)
=END.IF()
=RETURN()
```

Note that in this, and all the Excel macros that follow, command lines are preceded by an equal sign (=), whereas comment lines and macro names appear without any preceding character. You'll also note that all these Excel macros seem to spend most of their time either selecting ranges or changing formulae. This has to do with the peculiar programming model imposed by spreadsheets, in which cells and cell ranges act as the equivalent of variables and arrays in a more standard language. Unfortunately, this can make spreadsheet macro code difficult to follow, since the meaning of =Formula("LOTS",!A2) isn't as apparent as that of a statement such as A\$="Lots". Nevertheless, the two statements are equivalent.

Go_Lots begins by moving the screen window so that the cell at column 15 in row 1 of the worksheet is at the upper-left corner of the screen, thus revealing the Lotus Stock page, which is stored at that location in the BROKER1.XLS worksheet. The presence of the TRUE parameter in the HSCROLL and VSCROLL commands indicates that the number in the previous parameter points to an absolute location, that is, a specific location on the worksheet. Otherwise, if the second parameter was FALSE or was missing, the first parameter would be interpreted as a percentage of the total width or height of the worksheet.

Next, the macro prepares to accept a transaction order involving Lotus stock, by placing the string "LOTS" in cell location A2 (the location DynaComm queries to determine which security to buy or sell) and the number 0 in cell location S12 (the entry blank on the transaction order form). The exclamation point preceding the cell references identifies them as being on the active worksheet (BROKER1.XLS) as opposed to the macro sheet (BROKER1.XLM).

Next, the macro gives location S12 the name BuyTarget and location P11 (the location where Lotus transactions are recorded) the name TransTarget.

These range names are used by the DynaComm script during the stock transaction process.

Finally, the macro checks to see whether the cell TransTarget (P11) contains the string "Unrecorded". If so, it calls the Update macro. Otherwise, it exits.

The Go_IBM macro

The Go_IBM macro is very similar to the Go_LOTS macro:

```
Go_IBM
=HSCROLL(25,TRUE)
=VSCROLL(1,TRUE)
=FORMULA("IBM ",!A2)
=FORMULA("=0",!AC12)
=DEFINE.NAME("BuyTarget",!$AC$12)
=DEFINE.NAME("TransTarget",!$Z$11)
=IF(BROKER1.XLS!TransTarget="Unrecorded")
=GOTO(Update)
=END.IF()
=RETURN()
```

The only differences between Go_IBM and Go_Lots are the location of the Stock screen the macro scrolls to, the value it places in cell A2, and the location to which it assigns the names BuyTarget and TransTarget. The Go_IBM macro places the string "IBM " in A2 and defines the cells AC12 and Z11 as BuyTarget and TransTarget, respectively.

The ChartHLC macro

The macro called ChartHLC is executed when the user presses the button labeled "Last 21 Days" on either Stock screen:

```
ChartHLC
=VSCROLL(23,TRUE)
=RETURN()
```

The graphs that illustrate the high-low-close prices for each stock appear on the screen page below its Stock page, so the ChartHLC macro simply scrolls the screen down until row 23 is at the upper edge of the screen window, in order to expose the high-low-close chart.

The ChartPrice Macro

ChartPrice operates identically to ChartHLC, except that it scrolls the screen down to row 52, to reveal the chart of the stock's closing price over the past 60 trading days:

```
ChartPrice
```

```
=VSCROLL(52,TRUE)
=RETURN()
```

The ChartVol Macro

ChartVol scrolls the screen down to row 82, to reveal the chart of sales volume over the previous 60 trading days for the current stock:

```
ChartVol
=VSCROLL(82,TRUE)
=RETURN()
```

The Home Macro

The last of the navigational macros is Home, which jumps back to Windows Broker's Opening screen:

```
Home
=HSCROLL(1,TRUE)
=VSCROLL(1,TRUE)
=RETURN()
```

Home simply scrolls the screen to the left and then up until the cell at row 1, column 1 is at the upper-left corner of the visible screen window.

Communication Macros

The next several macros interact with DynaComm. They take the data DynaComm obtains, convert it into usable form, and issue buy and sell orders.

The Get_Prices Macro

Get_Prices, the first of the macros that interact with DynaComm, is used to obtain current price data for the two stocks in the portfolio. The macro runs when the user clicks the Update Prices button on the Opening screen.

```
=Get_Prices
=Beep
=FORMULA("&Quotes&",!$D$2)
=INITIATE("DYNACOMM","BROKER1.DCT")
=TERMINATE(A40)
=MESSAGE(TRUE,"Connecting to CompuServe")
=RETURN()
```

Get_Prices begins by beeping the PC's speaker (just for the heck of it), and then places the string "&Quotes&" in cell location D2 of the BROKER1.XLS worksheet. Then it uses the DDE command INITIATE to launch DYNACOMM.EXE, and passes it the name of the compiled

script it should execute: BROKER1.DCT. (DYNACOMM.EXE must be on your DOS path for this command to work.)

At this point, the DynaComm script takes over, examining the value of cell D2 to determine what process it should perform (get quotes, place a transaction order, or record a transaction). It then connects to CompuServe, retrieves the stock prices, and then uses DDE to send the data back to Excel, which stores it on the BROKER1.XLS worksheet in cell locations A26 and A28. Then the DynaComm script instructs Excel to launch the Cleanup macro, which interprets the data retrieved from CompuServe and moves it into the LOTS.XLS and IBM.XLS worksheets. (The discussion of BROKER1.DCP, below, will fully document the workings of the DynaComm script.)

The Cleanup Macro

The stock data that DynaComm receives from CompuServe isn't quite ready for spreadsheet use. Spreadsheets like to keep things orderly—each number and each label stored in a separate cell.

When DynaComm requests the latest price on a stock, however, it receives back a long string of data—including the name of the company (which may be truncated or abbreviated); the day's trading volume for the stock; the stock's high, low, and last prices; the change from the previous day's close; and the date of the last transaction—all of which gets squeezed onto a single line, like this:

```
INTERNATIONAL BUSINESS MA 19230 87.125 85.625 85.875 -1.625 3/19
```

Obviously that string has to be parsed into its constituent values before it is of much use in a spreadsheet, a task that DynaComm's script language undoubtedly could have handled. But rather than trying to figure out exactly how to do the parsing in DynaComm, I elected to simply poke the entire string through DDE into a specific cell location in BROKER1.XLS, and then use Excel's PARSE command, which was made for tasks like this.

This illustrates one of the hidden advantages of building applications that link macro languages from two or more programs—you can let the program that's best suited for a task do the work. The not-so-hidden disadvantage is that you've got to master two or more macro languages before you can know for sure which program is really best suited to the task.

Once it has sent the stock price data through the DDE channel to Excel, DynaComm instructs Excel to launch the Cleanup macro:

```
Cleanup
=MESSAGE(TRUE,"Formatting latest stock data")
=SELECT("R26C1:R28C1")
=PARSE("[[LOTUS DEV CORP COM      ][ 3790][ 22.250][ 21.500][ 21.750][ 0.000]
[ 2/22 ]")
```

Cleanup starts by displaying the string "Formatting latest stock data" on the status bar at the bottom of the Excel screen. Then it selects the cell range C26 to C28 (the range in which the Lotus and IBM stock price data has been stored), and utilizes the Excel PARSE command to break it into seven columns of data—one each for the stock name, the volume, high, low, and close figures, the change in price, and the date. The stock price data supplied by CompuServe always follows the same format, so I was able to use the standard template specified in the parameter to the PARSE command to break it up into individual values. (The square brackets in the template indicate where each data item begins and ends.) I used an actual example of the data for the template. When the macro executes, Excel compares the template to the data it is being asked to parse and carries out the parse operation using the same breaks between data items as in the template.

Next, the macro selects the range in which the PARSE command has placed the high, low, close, price change, and date information for the stocks, and applies Excel's General format to the entire range (meaning that numbers, labels, and dates will be displayed in the worksheet's default format for their type.)

```
=SELECT("R26C3:R28C7")
=FORMAT.NUMBER("General")
=SELECT("R28C3")
=SELECT("R26C6:R28C7")
=CLEAR(3)
=SELECT("R25C2:R28C2")
=CUT()
=SELECT("R25C6")
=PASTE()
=SELECT("R25C3:R28C6")
=CUT()
=SELECT("R25C2")
=PASTE()
=CALCULATION(3)
=SELECT("R26C2:R26C5")
=CUT()
=SELECT("R3C1")
```

It then clears the range that holds the price change and date, since Windows Broker doesn't use either of those items. Then it executes a series of cut-and-paste operations, which first rearranges the data so that it appears in the high, low, close, and volume order used in the LOTS.XLS and IBM.XLS worksheets. And finally it selects all the Lotus data and cuts it to the Clipboard in preparation for the next series of statements.

```
=OPEN("LOTS.XLS")
=MESSAGE(TRUE,"Updating Lotus data")
```

```

=SELECT("R7C3")
=SELECT.END(4)
=SELECT("R[1]C")
=PASTE()
=SELECT("RC[-1]")
=FORMULA("=TODAY()")
=COPY()
=PASTE.SPECIAL(3)
=SetRanges()
=CLOSE(TRUE)

```

In these statements the macro opens the LOTS.XLS worksheet, places the message "Updating Lotus data" on the Excel status line, and then moves the cursor to the bottom of the Lotus stock price data range and pastes the new Lotus data there. It accomplishes this by first selecting the top of the data range (R7C3), then moving to the bottom of the range with the SELECT.END(4) command, then moving one row farther down with the command SELECT("R[1]C"), which tells Excel to select the cell one row down in the same column. Then it issues a PASTE() command to paste the Lotus price and volume data into the worksheet, and then moves one column to the left and enters the formula =TODAY, which always displays the current date.

Next, in order to convert that formula into a static date (one that doesn't change from day to day), the macro copies the cell that holds the =TODAY formula and then issues the PASTE.SPECIAL(3) command, which pastes the value of the formula (the current date) back into the cell.

Then the macro calls the SetRanges subroutine (detailed below), which redefines a series of cell ranges used by BROKER1.XLS to include the new data. Finally, the macro issues the CLOSE(TRUE) command to close the LOTS.XLS file, before returning to the BROKER1.XLS file to work with the IBM price data, using the following statements:

```

=SELECT("R28C2:R28C5")
=CUT()
=SELECT("R3C1")
=OPEN("IBM.XLS")
=MESSAGE(TRUE,"Updating IBM Data")
=SELECT("R8C3:R8C3")
=SELECT.END(4)
=SELECT("R[1]C")
=PASTE()
=SELECT("RC[-1]")
=FORMULA("=TODAY()")
=COPY()
=PASTE.SPECIAL(3)

```



```

=SetRanges()
=CLOSE(TRUE)
=CALCULATION(1)
=SELECT("R1C1")
=MESSAGE(FALSE)
=RETURN()

```

This code is similar to the lines above that enter data into LOTS.XLS. Here the macro selects the IBM price data, cuts it to the Clipboard, and moves the cursor to the top of the BROKER1.XLS file. Then it opens the IBM.XLS file, posts a message that it is updating the IBM data, and follows the same series of steps as does the LOTS.XLS file to post the new data in IBM.XLS. When those steps are complete, it closes IBM.XLS, recalculates the BROKER1.XLS worksheet, moves the cursor to row 1, column 1, turns off the status-line message, and finally issues the RETURN() command, ending the Cleanup macro.

The SetRanges Macro

The SetRanges subroutine macro is used twice by the Cleanup macro to redefine the named ranges in the LOTS.XLS and IBM.XLS files.

```

SetRanges
=SELECT("R7C2")
=SELECT.END(4)
=SELECT("R[-21]C2:RC",)
=DEFINE.NAME("Last21",SELECTION())
=SELECT(OFFSET(SELECTION(),0,1))
=DEFINE.NAME("Last21high",SELECTION())
=SELECT(OFFSET(SELECTION(),0,1))
=DEFINE.NAME("Last21low",SELECTION())
=SELECT(OFFSET(SELECTION(),0,1))
=DEFINE.NAME("Last21close",SELECTION())
=SELECT.END(3)
=SELECT.END(4)
=DEFINE.NAME("Lastprice",SELECTION())
=SELECT.END(1)
=SELECT("R[-60]C2:RC")
=DEFINE.NAME("Last60",SELECTION())
=SELECT(OFFSET(SELECTION(),0,3))
=DEFINE.NAME("Last60close",SELECTION())
=SELECT(OFFSET(SELECTION(),0,1))
=DEFINE.NAME("Last60vol",SELECTION())
=RETURN()

```

Despite how complex its code looks, the operation of SetRanges is simple. It just moves to the last filled cell in a column of data—for instance the closing price column—and then selects that cell and the 20 immediately above it and defines the range as Last21Close (or Last21High, Last21Vol, Last21Low, as the case may be). Then it does the same thing for each of the other names it defines, with the exception that it selects the preceding 59 cells for the 60-day ranges.

That concludes the macros that deal with updating daily stock data. Next come the two macros that are used to actually purchase or sell stock, called Buy and Sell, respectively.

The Buy Macro

Buy is launched when the user presses the button labeled “Buy” on either the IBM or Lotus Stock screen.

```
Buy
=FORMULA("B",!B2)
=FORMULA("=BuyTarget",!C2)
=FORMULA("&BuySell&",!$D$2)
=MESSAGE(TRUE,"Connecting to CompuServe")
=Beep
=FORMULA("&BuySell&",!$D$2)
=INITIATE("DynaComm","BROKER1.DCT")
=TERMINATE(A51)
=RETURN()
```

The macro begins by placing the letter “B” (for Buy) in cell B2 of BROKER1.XLS, the formula =BuyTarget in cell C2, and the text “&Buy-Sell&” in cell D2. BuyTarget will be equal to either LOTS or IBM, depending on which Stock screen was active when the Buy button was pressed. Then the macro uses the Excel status bar to announce that it is connecting to CompuServe, beeps the PC’s speaker, and utilizes the DDE command INITIATE to launch DynaComm and instruct it to execute the compiled script BROKER1.DCT.

The DynaComm script uses the values the Buy macro places in cells B2, C2, and D2 to determine which actions it should perform. Once launched, it carries out the transaction and automatically enters data about it into the Excel worksheet, before ending.

Next the Excel script terminates the DDE connection established by the INITIATE command. Since the Buy macro’s INITIATE command is located in cell A51 of the BROKER1.XLM macro sheet, the command to terminate it is TERMINATE(A51). Then the macro comes to an end with the command RETURN.

The Sell Macro

Sell is nearly identical to the Buy macro, varying only in the value it places in cell B2 and the location used as a parameter in its TERMINATE command:

```
Sell
=FORMULA("S",!B2)
=FORMULA("=buytarget",!C2)
=MESSAGE(TRUE,"Connecting to CompuServe")
=Beep
=FORMULA("&BuySell&",!$D$2)
=INITIATE("DynaComm","BROKER1.DCT")
=TERMINATE(A62)
=RETURN()
```

Transaction-Recording Macros

Once a purchase or sell order has been placed, the DynaComm macro that places it inserts the text string "Unrecorded" into the Last Transaction field on the stock sheet for the stock specified in the order. (This indicates that the worksheet has not yet obtained the data from Quick*Way telling it when the trade actually occurred, the price per share, and the commission paid on the trade.)

From that point on, every time the Go_Lots or Go_IBM macro takes you to that stock sheet, it launches the Update macro to determine whether you wish to obtain the complete data for the last transaction.

The Update Macro

The Update macro starts by creating a message box informing you that the last transaction has not been recorded, and offering you the chance to do so, as shown in Figure 13.10. The code for Update looks like this:

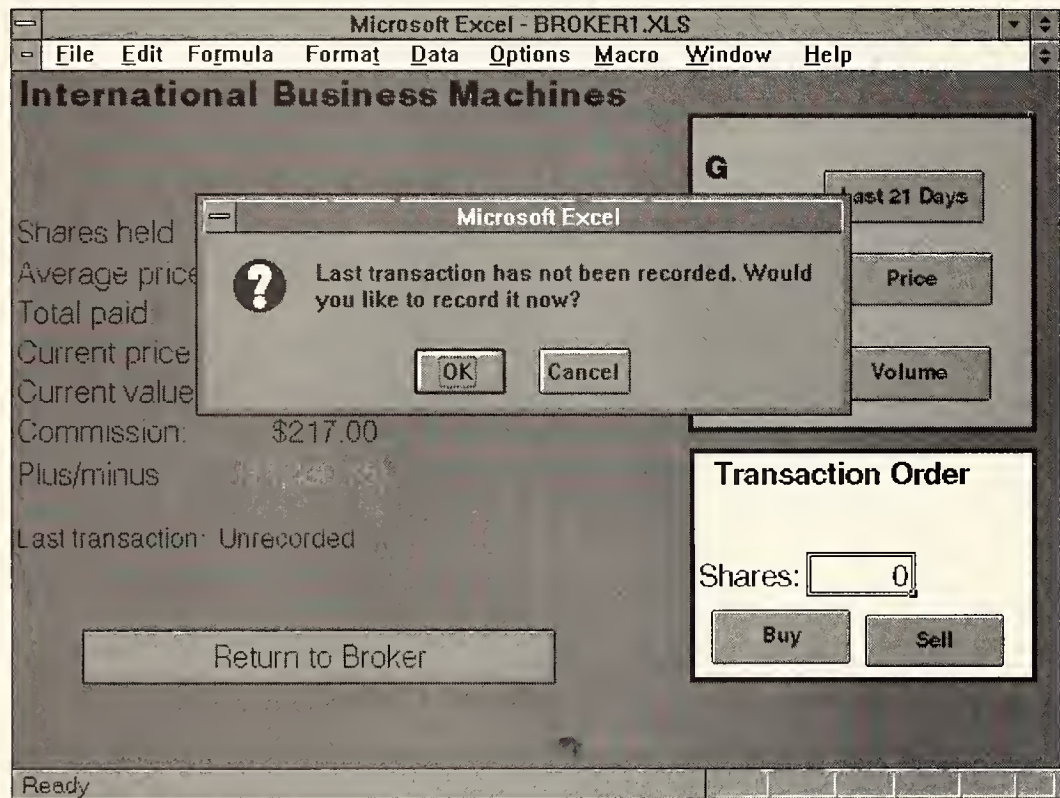
```
Update
=ALERT("Last transaction has not been recorded. Would you like to record it now?".1)
=IF(C2=FALSE)
•RETURN()
=END.IF()
=MESSAGE(TRUE,"Connecting to CompuServe")
=Beep
=FORMULA("&Update&",!$D$2)
=INITIATE("DynaComm","BROKER1.DCT")
=TERMINATE(C9)
=RETURN()
```

If you press the Cancel button on the Alert box, cell C2 on the macro sheet (the location of the =ALERT statement) will be equal to FALSE, so the subroutine will terminate, returning control to the Go_Lots or Go_IBM routine. Otherwise, it places the message "Connecting to CompuServe" on

the Excel status line, beeps the PC's speaker, places the text string "&Update&" in cell D2 of BROKER1.XLS to tell DynaComm what operation it is to perform, and uses the INITIATE command to launch DynaComm and execute BROKER1.DCT. Then it immediately terminates the DDE connection and returns control to the Go_Lots or Go_IBM routine.

Figure 13.10

The Update Transactions reminder box



The DynaComm Update routine checks with Quick*Way to see if the transaction has occurred yet. If so, it obtains the data about the transaction, updates the Last Transaction field, and then pokes the complete transaction data into Excel, starting at cell location A35. Then it instructs Excel to execute the Transaction macro, which moves the data into the area of the BROKER1.XLS worksheet used to store transactions for the current stock. If the transaction has not yet taken place, DynaComm logs off without changing the BROKER1.XLS worksheet. You can enter several transaction orders in a row, and the Update routine will record them as they take place.

The Transaction Macro

By the time DynaComm instructs Excel to launch the Transaction macro, the DynaComm script has already placed all the data it has obtained from

Quick*Way about the transaction into BROKER1.XLS, starting at location A35, as detailed in Table 13.3.

Table 13.3 Transaction Data Storage Locations

Cell	Contents
A35	Transaction date
B35	"Buy" or "Sell"
C35	Quantity of shares transacted
D35	"LOTS" or "IBM"
E35	Price per share
F35	Commission paid

The Transaction macro moves this data into either the LotusTrans or the IBMTrans ranges on BROKER1.XLS, depending on which stock was traded.

```
Transaction
=SELECT("R35C4")
=IF(ACTIVE.CELL()="LOTS",SELECT("LotusTrans"),SELECT("IBMTrans"))
=SELECT.END(4)
=SELECT("R[1]C")
=FORMULA("=R35C1")
=SELECT("RC[1]")
=FORMULA("=R35C2")
=IF(ACTIVE.CELL()="Sell")
=CLEAR(3)
=SELECT("RC[1]")
=FORMULA("=R35C3")
=SELECT("RC[-1]")
=ELSE()
=FORMULA("=R35C3")
=END.IF()
```

The Transaction macro starts by examining the contents of D35 to determine which transaction range should receive the data. If D35 contains "LOTS", it issues the command SELECT ("LotusTrans"). Otherwise, it selects the IBMTrans range. Then it moves to the bottom row of data in the range, then moves down one more row to the first blank row and enters the formula =R35C1, to copy the transaction date into the left-most cell of the new row.

Next the macro has to enter the contents of cell C35—the quantity of shares bought or sold—into either the second or the third cell of the new row. The second column of the transaction range is used to record stock purchases, so it receives the number from C35 if B35 is “Buy”, whereas if B35 is “Sell”, the number from C35 is entered in the third column, which is used to record sales. So the macro starts by moving to the second cell in the new row and entering the formula =R35C2 there, thus copying the value of cell B35 into the cell. Then it evaluates the contents of that cell, and if they are equal to the string “Sell”, clears the cell and moves one column to the right, entering there the contents of C35. Otherwise, if the evaluated cell is equal to “Buy”, it overwrites the =R35C2 formula with =R35C3, entering the number of shares purchased into the second cell of the row.

```
=SELECT("RC[2]")
=FORMULA("=R35C5")
=SELECT("RC[1]")
=FORMULA("=RC[-1]*RC[-3]")
=SELECT("RC[1]")
=FORMULA("=RC[-2]*RC[-3]")
=SELECT("RC[1]")
=FORMULA("=R35C6")
=SELECT("RC:RC[-6]")
=COPY()
=PASTE.SPECIAL(3)
```

Next the macro moves two columns to the right, into the cell used to record the price per share of the stock, and enters the formula =R35C5 to copy the price from cell E35. Then it moves one more column right, into the column that stores the total price paid for purchase transactions, and enters the formula =RC[-1]*RC[-3], setting the value of that cell equal to the number of shares purchased multiplied by the price per share.

After recording the total price paid, the macro moves one column to the right, into the cell that stores the proceeds from sales transactions, and enters the formula =RC[-2]*RC[-3], setting the value of that cell equal to the value of the price per share column multiplied by the value of the quantity sold column. Finally, it once again moves one column to the right and enters the formula =R35C6 to copy the commission paid on the trade into the active cell.

Next, the macro copies the entire new row of data to the Clipboard and then pastes it back into the same row as values rather than formulae. This breaks the link between this record and the values of cells A35 through F35. Without this step, the transaction record would change whenever the contents of those cells changed.

```
=FORMAT.NUMBER("0.00")
```



```

=SELECT("RC")
=FORMAT.NUMBER("dd-mmm-yy")
=SELECT("R[1]C:R[1]C[7]")
=INSERT(2)
=SELECT("R35C1:R35C6")
=EDIT.DELETE(2)
=SELECT("R35C1")
=IF(LEN(ACTIVE.CELL())>0,GOTO(trans),GOTO(Home))
=RETURN

```

Next, the Transaction macro formats the entire new record to display its contents as a number with two-digit accuracy, and then selects the first cell in the record (the date of the transaction) and reformats it to display as a date. Then it moves down one row, highlights the blank cells that will constitute the next record, and issues the INSERT(2) command, which tells Excel to insert new cells in the place of the highlighted ones, thus pushing the highlighted ones down one row. This effectively extends the transaction range and all formulae that refer to it by an additional row, ensuring that the next record that is entered is also included in those formulae.

Finally, the macro jumps back to the cell range in which DynaComm left the transaction data, highlights the range, and deletes it with the command EDIT.DELETE(2), which makes Excel shift the cells below the highlighted range up to take its place. Then the macro checks the contents of cell A35. If A35 is not empty, it means that DynaComm retrieved more than one transaction record during its last update session, so the macro jumps back to the start of the Transaction macro and repeats the entire process for the next record. Otherwise, it returns, its job done.

That concludes the Excel macros for the Windows Broker application.

Exploring BROKER1.DCP

BROKER1.DCP is executed whenever Windows Broker needs to perform an on-line task, no matter whether that task is to retrieve stock-price information, post a buy or sell order, or record transactions.

The BROKER1.DCP script file contains 14 named routines: Intro, GetVars, FixVars, Dial, Select_Task, Messagebar, StripString, Wait_Send, GoQuotes, GoQWK, BuySell, GoUpdate, Update, and Fail. The first five are introductory routines; they are executed in sequence every time the compiled version of the script (BROKER1.DCT) is run. The next three (Messagebar, StripString, and Wait_Send) are general-purpose subroutines that are called repeatedly by one or more of the other routines. The last six routines are task dependent.

The Introductory Routines

Let's begin at the beginning, with the first of the introductory routines, Intro.

The Intro Routine

The Intro routine (identified by the label “*Intro”) consists of just four lines of code:

```
*Intro
LOAD "BROKER1.DCS"
TITLE "Windows Broker Online"
ACCESS "EXCEL" "BROKER1.XLS" %Channel
IF ERROR CANCEL
```

The first statement instructs DynaComm to load the settings file called BROKER1.DCS, which contains the communications settings to be used for communicating with CompuServe. Next, the routine changes the title of the DynaComm window to “Windows Broker Online”, and then attempts to open a DDE channel to the BROKER1.XLS worksheet. If the access attempt is successful, the channel number will automatically be assigned to the integer variable %Channel. Otherwise, the IF ERROR routine issues the CANCEL command, ending script execution.

The GetVars Routine

Once a DDE channel has been opened to the BROKER1.XLS worksheet, the GetVars routine uses it to obtain the data the script needs to carry out its assigned task:

```
*GetVars
REQUEST $Task FROM %Channel "R2C4"
REQUEST $Phone FROM %Channel "R1C1"
REQUEST $Name FROM %Channel "R1C2"
REQUEST $Pass FROM %Channel "R1C3"
REQUEST $Account FROM %Channel "R1C4"
REQUEST $Buyname FROM %Channel "R1C5"
REQUEST $Buypass FROM %Channel "R1C6"
REQUEST $Portfolio FROM %Channel "R1C7"
REQUEST $Buy FROM %Channel "R2C1"
REQUEST $Buyse11 FROM %Channel "R2C2"
REQUEST $Buynum FROM %Channel "R2C3"
ACCESS CANCEL %Channel
```

GetVars starts off by using the DynaComm DDE command REQUEST to obtain the contents of cell location D2 (R2C4) and assign them to the variable \$Task. (Following this command, \$Task will be equal to either &Buy-Sell&, &Update&, or &Quotes&.)

Next, the routine issues ten more requests to obtain the CompuServe telephone number; the user's CompuServe identification and password; his or her Quick*Way account number, log-on name, and password; the name of the portfolio in which any trades are to take place; the symbol of the stock to buy or sell (if any); whether to buy or sell; and the number of shares to buy or sell. All these values are stored under the Windows Broker banner on the Opening screen of the application. Finally, the routine uses the ACCESS CANCEL routine to close the DDE channel.

The FixVars Routine

Once the variables have been obtained from Excel, DynaComm executes the FixVars routine. FixVars resolves a messy anomaly involving DDE communications between Excel and DynaComm. The data DynaComm obtains from Excel tends to arrive with several carriage-return/line-feed combinations of unknown origin tacked onto to it. FixVars removes those extra characters.

```
*FixVars
PERFORM StripString($Task)
PERFORM StripString($Phone)
PERFORM StripString($Name)
PERFORM StripString($Pass)
PERFORM StripString($Buynum)
PERFORM StripString($Buypass)
PERFORM StripString($Account)
PERFORM StripString($Portfolio)
$Buy=SUBSTR($Buy,1,4)
$BuySell=SUBSTR($BuySell,1,1)
%Buynum=NUM($Buynum)
$Buynum=STR(%Buynum)
```

Most of the strings that arrive from Excel are enclosed in ampersands, so the FixVars routine calls the StripString subroutine to cut the strings down until they consist of only that data previously separated by ampersands, thus eliminating any extraneous characters tacked onto the end of the string during the DDE communication.

In the case of the \$Buy, \$BuySell, and \$BuyNum variables, however, there is no need to enclose them in bracketed ampersands or to use StripString. Since the lengths of \$Buy (four characters—either "LOTS" or "IBM ") and \$BuySell (one character—either "B" or "S") are known, the FixVars routine simply uses the DynaComm SUBSTR (substring) function to change them to the correct size. Meanwhile, since \$BuyNum contains a number, the extraneous carriage-return/line-feed characters don't affect the string's numeric value. So the routine simply uses the DynaComm NUM function to convert \$BuyNum to an integer, which it assigns to the integer variable %BuyNum and then converts back to a string, thereby removing the extraneous characters.

The Dial Routine

Once all the variables have been cleaned up, DynaComm executes the Dial routine (and its LoginLoop subroutine) to connect to CompuServe.

```
*Dial
DIAL $Phone
WAIT DELAY "1"
*LoginLoop
SEND NOCR "^C"
WHEN STRING 1 "ID" RESUME
WHEN TIMER "2" GOTO LoginLoop
WAIT RESUME
WHEN CANCEL
SEND $Name
PERFORM Wait_Send ("Password: ", $Pass)
WAIT STRING "Last"
WAIT QUIET "1"
```

The Dial routine starts off by issuing the DynaComm DIAL command, giving it the telephone number stored in the string variable \$Phone as a parameter. At this point DynaComm automatically pauses script execution until a connection is made.

Once the telephone has been answered, the script waits one more second, and then executes a loop called LoginLoop, in which it sends the Ctrl-C command to get CompuServe's attention and establishes a pair of WHEN conditions. The first tells DynaComm that when it receives the string "ID" from CompuServe, it should continue execution at the statement after the WAIT RESUME command. The second WHEN condition tells DynaComm that it should jump back to the beginning of LoginLoop every time its internal timer reaches two seconds. The following statement, WAIT RESUME, pauses script execution until the RESUME command is issued.

What all that means is that DynaComm sends a Ctrl-C to CompuServe every two seconds until it receives the CompuServe log-in prompt "Please enter user ID:". Once that string is received, the script cancels the WHEN conditions, sends the user's CompuServe log-on name, and then calls the Wait_Send subroutine, passing it parameters that tell it to wait for the prompt "Password: " and then send the contents of the variable \$Pass back to CompuServe.

Once the log-on procedure is complete, the script waits for the string "Last", which appears in the standard CompuServe welcome message. (The message begins, "Last access: " followed by the time and date the user last accessed CompuServe.) It then waits for the line to go quiet for one second, which indicates that the welcome message is done, before proceeding to the Select_Task routine.

The Select_Task Routine

Once the script has successfully logged onto CompuServe (which the “Last” prompt is proof of), it performs the Select_Task routine to determine what its next step should be:

```
*Select_Task
SWITCH $Task
CASE "Update"
PERFORM GoQWK
PERFORM GoUpdate
LEAVE
CASE "BuySell"
PERFORM GoQWK
PERFORM BuySell
LEAVE
CASE "Quotes"
PERFORM GoQuotes
SWITCH END
QUIT
```

Select_Task uses DynaComm’s SWITCH statement, which is the equivalent of the SELECT CASE statement described in Chapter 2, to evaluate the contents of the variable \$Task. If \$Task contains the string “Update”, then the script’s job is to update the portfolio-transaction records. It does so by performing the GoQWK routine, followed by the GoUpdate routine. Once the GoUpdate routine is complete, the script leaves the SWITCH routine and executes the first command past the SWITCH END statement, QUIT, which stops the script and unloads DynaComm.

Meanwhile, if \$Task is equal to “BuySell”, the script’s job is to either purchase or sell stock, so it executes first the GoQWK routine, followed by the BuySell routine. If \$Task is equal to “Quotes”, the script’s job is to get current stock prices, so it performs the GoQuotes routine.

General-Purpose Subroutines

The following three routines—Messagebar, StripString, and Wait_Send—are repeatedly called as subroutines by other routines in BROKER1.DCT.

The Messagebar Subroutine

Messagebar is called whenever the script wants to send a command to Excel through the DDE channel:

```
*Messagebar ($Message)
ACCESS "EXCEL" "SYSTEM" %Channel
INSTRUCT %Channel $Message
```

```
ACCESS CANCEL %Channel
RETURN
```

This routine takes its name from the fact that most of the calls to it throughout `BROKER1.DCP` are intended to update the Excel status-line message bar. For instance, the commands

```
$Message='[MESSAGE(TRUE,"Hanging Up")]'  
PERFORM Messagebar($Message)
```

would place the text “Hanging Up” on the Excel message bar. But the `INSTRUCT` statement can actually be used to send any command to Excel. Thus the script also uses the `Messagebar` routine to command Excel to launch macros, such as the `Cleanup` macro Excel uses to incorporate new stock-price data into the worksheet. The commands to launch the `Cleanup` macro would look like this:

```
$Message='[RUN("BROKER1.XLM!Cleanup")][BEEP()]'  
PERFORM Messagebar($Message)
```

The `Messagebar` routine operates by first establishing a DDE channel to Excel, then instructing it perform the command specified in the `$Message` variable, and finally canceling the DDE channel and returning to the calling routine.

The StripString Subroutine

`StripString` is called repeatedly by the `FixVars` routine to clean up data items obtained via DDE from Excel:

```
*StripString($Stripstring)  
%P1=POS($Stripstring,"&")  
%P2=POS($Stripstring,"&",%P1+1)  
$Stripstring=SUBSTR($Stripstring,%P1+1,%P2-2)  
RETURN
```

`StripString` uses the `DynaComm` `POS` function to obtain the location of the first ampersand in the string `FixVars` passes to it, and assigns that location to the integer variable `%P1`. Then it uses `POS` again to find the next ampersand’s location, which it assigns to `%P2`. Finally, it sets the new value of `$Stripstring` equal to the part of the original `$Stringstring` that lies between `%P1` and `%P2`, using `DynaComm`’s `SUBSTR` function.

Thus if the text of `$Stripstring`, as passed by `FixVars`, is “&MyAccountName&JFJGLD”, `%P1` will be equal to 1, `%P2` will be equal to 15, and the new `$Stripstring` will be “MyAccountName”.

The Wait_Send Subroutine

Wait_Send is used whenever the script has to wait for a known prompt and respond with a known string:

```
*Wait_Send ($Prompt,$Response)
WAIT STRING $Prompt
WAIT QUIET "1"
SEND $Response
RETURN
```

The Wait_Send routine pauses script execution until it receives the text designated by \$Prompt from CompuServe, then waits for one second of silence on the communications line (which gives DynaComm a chance to finish painting the screen once CompuServe stops sending data), then sends the text in response to CompuServe before returning to the calling routine. The text it waits for doesn't have to be on a line by itself; in fact, in most cases the script waits for just a word or two out of a long line of text. All that matters is that the text is unique enough to be a reliable indicator that CompuServe has arrived at the specific prompt the script is waiting for.

The WAIT QUIET statement here and in other routines really isn't necessary, since most of the time the BROKER1.DCT script will operate in the background, hidden behind Excel, where it doesn't really matter how messy the screen looks. But since some users might like to activate DynaComm with the Alt-Tab switch in order to watch the on-line session take place, these brief pauses are a nice device for keeping things tidy on screen without seriously affecting the script's performance time.

Task-Specific Routines

Execution of the remaining routines in BROKER1.DCP is based on the task the script has been launched to perform. The first of these, GoQuotes, is used to update stock prices.

The GoQuotes Routine

GoQuotes is used when the BROKER1.DCT script has been executed in response to the user clicking on the Update Prices button. The routine takes over following a successful log-in by the Dial routine with the main CompuServe or ZiffNet prompt (since some CompuServe members may connect initially with ZiffNet, home of *PC/Computing's* PCContakt service, *PC Magazine's* Magnet, and other Ziff-Davis information services, rather than with the main CompuServe area):

```
*GoQuotes
SEND "GO CIS:QUOTES"
$Message='[MESSAGE(TRUE,"Connecting To Quick Quote")]'
```

PERFORM Messagebar(\$Message)

```

PERFORM Wait_Send ("choice", "1")
PERFORM Wait_Send ("Issue:", "LOTS, IBM")
$Message='[MESSAGE(TRUE,"Collecting Quotes")]'
```

PERFORM Messagebar(\$Message)

```

COLLECT $Junk
COLLECT $Junk
COLLECT $Junk
COLLECT $Junk
COLLECT $LOTS
COLLECT $IBM
PERFORM Wait_Send ("Issue:", "^M")
PERFORM Wait_Send ("!", "/Off")
$Message='[MESSAGE(TRUE,"Hanging Up")]'
```

PERFORM Messagebar(\$Message)

```

HANGUP
CLEAR
WAIT DELAY "2"
ACCESS "EXCEL" "BROKER1.XLS" %Channel
POKE $LOTS TO %Channel "R26C1"
POKE $IBM TO %Channel "R28C1"
ACCESS CANCEL %Channel
$Message='[MESSAGE(FALSE)]'
```

PERFORM Messagebar(\$Message)

```

$Message='[RUN("BROKER1.XLM!Cleanup")][BEEP()]'
```

PERFORM Messagebar(\$Message)

```

RETURN
```

The routines start by issuing the command `GO CIS:QUOTES`, which links DynaComm to CompuServe's Quick Quote service no matter whether you've dialed into CompuServe or ZiffNet. Then the `Wait_Send` subroutine is called, with instructions to wait for the string "choice" (which appears in the CompuServe Quick Quote main menu prompt "Enter choice:") and to respond by sending a 1 to access the Current Quotes menu item.

Next, the routine waits for the "Issue:" prompt, to which it responds with the symbols of the stocks for which it wants current prices. Then it updates Excel's status bar to read "Collecting Quotes", and issues a series of `COLLECT` commands, each of which captures a complete line of text from CompuServe.

The first four `COLLECT`s receive things that we're not interested in—an empty line, a column headings line, and so on—so the script just assigns them, in sequence, to the variable `$Junk`. Then comes the data we're after, two lines that look like this:

LOTUS DEV CORP COM	5947	37.750	36.750	36.875	-0.625	4:00
INTERNATIONAL BUSINESS MA	19260	86.625	85.750	86.125	0.250	4:36

The script assigns the first of these lines to the variable \$LOTS and the second to \$IBM. Then it waits for the “Issue:” prompt to reappear, sends a Ctrl-M to leave the Quick Quote service, and then waits for the standard CompuServe “!” prompt. When that prompt appears, the script sends the command /OFF (signaling to CompuServe that the user wishes to disconnect) and updates the Excel status-line message area to read “Hanging Up”. Then the script issues the DynaComm HANGUP command to disconnect the modem, and the CLEAR command to clear the terminal screen.

That concludes the script’s on-line work. Now all it has to do is get the stock price data into the Excel worksheet. It begins this part of its job by establishing a new DDE link to Excel, and then using the DynaComm DDE command POKE (which sends a data item to the DDE host and waits for an acknowledgment) to copy the contents of the variable \$LOTS into cell location A26 of the BROKER1.XLS file, and the contents of \$IBM into cell location A28. With that job done, it terminates the DDE connection with the command ACCESS CANCEL %Channel, and calls the Messagebar subroutine with the parameter [MESSAGE(FALSE)], to turn off the Excel status-bar message line. Finally, the routine uses Messagebar again to instruct Excel to run the Cleanup macro on BROKER1.XLM (described earlier), and returns to the Select_Task routine, which called it.

The GoQWK Routine

The Select_Task routine calls GoQWK to establish a link to the Quick*Way brokerage service whenever the user wishes to place an order or record a transaction:

```
*GoQWK
$Message='[MESSAGE(TRUE,"Connecting To Quick*Way")]'
```

```
PERFORM Messagebar($Message)
SEND "GO CIS:QWK"
WHEN STRING 1 "Access" RESUME
WAIT RESUME
WAIT DELAY "1"
SEND "8"
PERFORM Wait_Send ("name", $Account)
PERFORM Wait_Send ("word", $Buyname)
RETURN
```

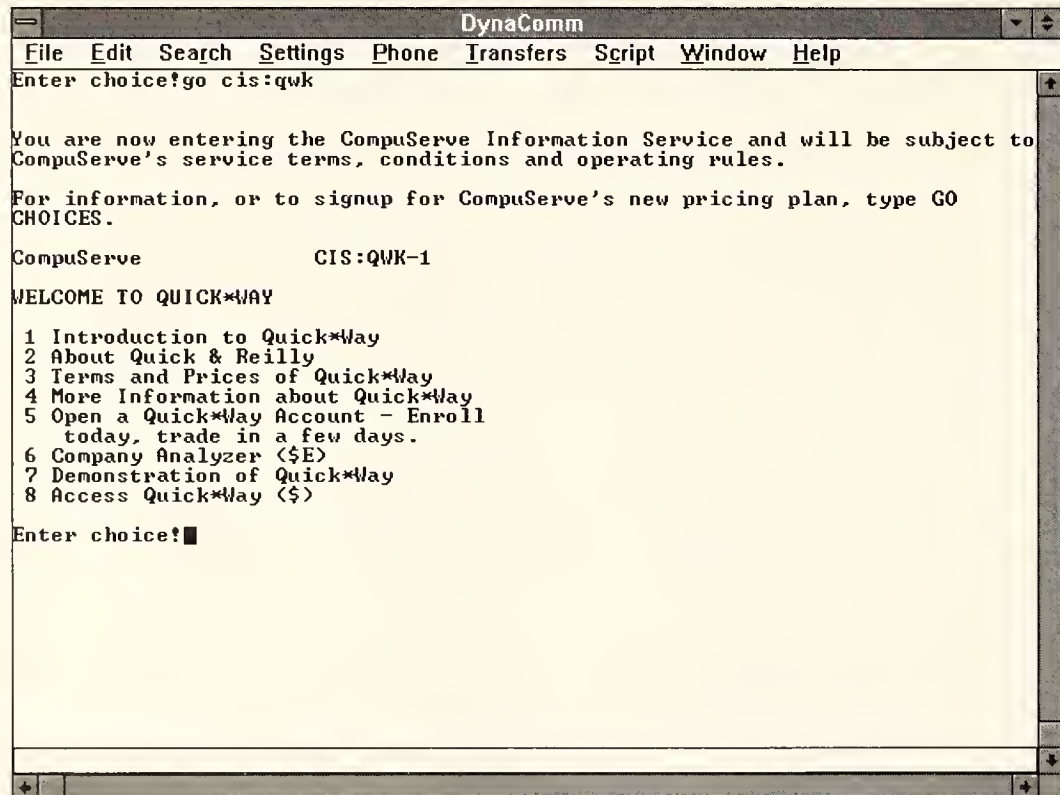
The routine springs into action at the main CompuServe prompt, following a successful log-in. It starts by updating the Excel status-line message bar to read “Connecting to Quick*Way”, and then issues the command GO CIS:QWK to link to the Quick*Way brokerage.

Quick*Way’s opening screen, shown in Figure 13.11, is a menu offering a variety of information about the Quick*Way service, as well as the opportunity to open an account. But the script is interested in only one item, number

8, which reads "Access Quick*Way", so it waits for that text to arrive over the communications line. When it does, the script pauses for a second, and then responds by selecting menu item 8.

Figure 13.11

The main
Quick*Way menu



The script then issues two `Wait_Send` commands in a row, waiting first for Quick*Way's "Username" prompt (to which it replies by sending the contents of the string variable `$Account`) and then for the prompt "Password:" (to which it replies by sending the contents of the variable `$Buyname`).

That concludes the process of linking to the Quick*Way service, so the script returns to the `Select_Task` routine to see what it is to do next.

The BuySell Routine

`Select_Task` calls the `BuySell` routine to enter stock purchase or sale orders:

```

*BuySell
PERFORM Wait_Send ("NUMBER", "211")
$message='[MESSAGE(TRUE,"Sending transaction order")]
PERFORM Messagebar($message)
PERFORM Wait_Send ("Short Name", $Portfolio)
PERFORM Wait_Send ("Symbol", $Buy)
PERFORM Wait_Send ("(S)", $Buysell)
  
```

```
PERFORM Wait_Send ("of", $Buynum)
PERFORM Wait_Send ("word", $Buypass)
```

Once it has logged into Quick*Way, the script receives the prompt “TYPE NUMBER TO SELECT”, to which it replies by sending “211”, the address of Quick*Way’s BUY OR SELL STOCKS page. This page consists of a series of prompts about the transaction. So, after updating the Excel status-line message bar, the script issues a series of Wait_Send commands that respond to each of the prompts that appear: Quick*Way sends “Short Name” (asking for the short name of the portfolio), and the script responds with \$Portfolio. Quick*Way prompts “Symbol” (asking for the symbol of the stock to buy or sell) and the script responds with \$Buy. Quick*Way prompts “(S)” (as in (B)uy or (S)ell) and the script responds with \$Buysell. Quick*Way prompts “of” (from “Number of Shares”) and the script responds with \$Buynum. Finally, Quick*Way prompts “word” (asking for the account’s special transaction password) and the script responds with \$Buypass.

```
PERFORM Wait_Send ("Correct", "Y")
IF $Buysell="S" WHEN STRING "specific" SEND "N"
PERFORM Wait_Send ("Again","N")
WHEN CANCEL
PERFORM Wait_Send ("Other","N")
PERFORM Wait_Send ("NUMBER", "0")
WAIT STRING "!"
$Message='[MESSAGE(TRUE,"Hanging up")]'
PERFORM Messagebar($Message)
SEND "/OFF"
HANGUP
```

Once the order has been entered, Quick*Way displays a summary of it and asks “Is this ORDER Correct”, to which the script responds “Y”. Next, in the case of a sell order in which the user owns more shares of the security than he or she is selling, Quick*Way will ask if the seller wishes to sell specific shares. The script anticipates this eventuality with a WHEN STRING condition that sends the response “N” if it receives the text “specific”.

Meanwhile, the script calls Wait_Send to handle the other prompt that might appear at this point, “Order again in this Portfolio”, to which it also responds “N”. Once that prompt has been received, the possibility of receiving the “specific” prompt has passed, so the script cancels the WHEN STRING condition. Finally, it responds to the Quick*Way “Order in another Portfolio” prompt by sending “N”, concluding the Quick*Way transaction routine.

Once the transaction is complete, Quick*Way repeats its “TYPE NUMBER TO SELECT” prompt. This time the script responds by sending a 0, the command to disconnect from the Quick*Way service. It then waits for a

“!” prompt from CompuServe, to which it responds by sending the “/off” command. And then it updates the Excel status-bar message area to read “Hanging up” before issuing the DynaComm HANGUP command.

```
ACCESS "EXCEL" "BROKER1.XLS" %Channel
INSTRUCT %Channel '[BEEP()]'
INSTRUCT %Channel '[FORMULA("Unrecorded","Transtarget")]'
INSTRUCT %Channel '[MESSAGE(FALSE)]'
ACCESS CANCEL %Channel
RETURN
```

The final steps in the BuySell routine begin with the script opening a DDE channel to BROKER1.XLS. It uses the channel first to instruct Excel to beep the PC's speaker (signaling the user that the transaction has been entered), and then to tell Excel to enter the text “Unrecorded” in the named range Transtarget (which points to the Last Transaction: area for the stock just purchased or sold). Finally, it instructs Excel to clear its status-bar message area, closes the channel, and returns to the Switch_Task routine, which called it.

The GoUpdate Routine

GoUpdate is called following GoQwk when the script has been instructed to record the last purchase or sale transaction. Its first job is to determine if the data the script seeks about the transaction is available yet.

```
*GoUpdate
WHEN STRING 1 "513" WHEN CANCEL STRING, GOTO Update
WAIT STRING "SELECT"
WHEN CANCEL STRING
GOTO Fail
```

Quick*Way relays information about completed transactions by means of messages. If it has information for you about one or more recent transactions, it sends the line “- You have new Messages (Menu # 513) -” before its initial “TYPE NUMBER TO SELECT” prompt. So the GoUpdate routine simply establishes a WHEN STRING condition that looks for the text string “513”. If it receives that text, it cancels the WHEN STRING condition, and jumps to the Update routine (described below). Meanwhile, if it receives the text “SELECT” without having received “513”, then no transaction messages are available, and so it jumps to the Fail routine (described below).

The Update Routine

Update is executed when the script receives the message “- You have new Messages (Menu # 513) -” during an on-line session launched for the purpose of updating transactions.

```
*Update
$Message='[MESSAGE(TRUE,"Retrieving Transaction Data")]'
```

```
PERFORM Messagebar($Message)
SEND "513"
TABLE DEFINE 0 FIELDS CHAR 8 CHAR 4 CHAR 12 CHAR 4 CHAR 12 CHAR 12
WAIT STRING "Message"
```

The routine begins by posting the message “Retrieving Transaction Data” on the Excel status bar. Then it sends the text “513” to Quick*Way, instructing it to go to the messages page. Next, the routine defines a Dyna-Comm structured table to hold the message data. Each record in the table consists of six fields. The first field has room for 8 characters of data, the second 4, the third 12, the fourth 4, and the fifth and sixth 12 each. Then it waits for the string “Message”, after which it enters a subroutine called Loop.

Transaction messages from Quick*Way follow a standard form. A typical message is shown in Figure 13.12.

Figure 13.12

A typical
Quick*Way
message

```
Message Received 03:50 PM ON 3/20/92
The Message You May Wish to Delete is:
```

```
PBONNER Acct. no.: Q5555-5555 Your
Sell order for 100 LOTS at Mkt was
executed at 36 3/4. The commission
was $29.00.
```

The Loop subroutine is designed to interpret these messages:

```
*Loop
COLLECT $Date
WAIT STRING "Acct"
COLLECT $Junk
COLLECT $Order
COLLECT $Price
COLLECT $Commis
PERFORM Wait_Send ("Delete", "Y")
```

Loop starts by using a COLLECT command to assign the line that starts “Message Received” to the variable \$Date, and then waits for the text string

“Acct”. It assigns the line on which “Acct” appears (which has no useful data) to the variable \$Junk, and then follows that by assigning the contents of the next three lines to the variables \$Order, \$Price, and \$Commis.

So if the Loop subroutine received the message listed above, \$Date would contain the text “Received 03:50 PM ON 3/20/92” (“Message” would be omitted because the COLLECT command wasn’t issued until after it had been received). \$Order would contain “Sell order for 100 LOTS at Mkt was”. \$Price would contain “executed at 36 3/4. The commission”. And \$Commis would contain “was \$29.00.”

Next, the script waits for the prompt “Delete this Message: (Y,N)” to which it responds “Y”. Then it sets about turning the text in \$Date, \$Order, \$Price, and \$Commis into useful data:

```
SET %Temp LENGTH($Date)
SET @R0.1 SUBSTR($Date,%Temp-8,8)
SET %Temp LENGTH($Order)
SET @R0.2 SUBSTR($Order,1,4)
SET @R0.3 SUBSTR($Order,15,%Temp-29)
SET @R0.4 SUBSTR($Order,%Temp-14,4)
SET @R0.4 FILTER(@R0.4," ","")
SET %Temp LENGTH($Price)
SET $Price SUBSTR($Price,13,%Temp-28)
PARSE $Price $Price "." $Junk
SET @R0.5 $Price
SET %Temp LENGTH($Commis)
IF SUBSTR($Commis,1,3)="was" $Commis=SUBSTR($Commis,4,%Temp-4)
SET %Temp LENGTH($Commis)
SET @R0.6 SUBSTR($Commis,2,%Temp-2)
IF @R0.2="Sell" $TRANS="Sold ", ELSE $TRANS="Bought "
$Quan=TRIM(@R0.3)
$Price=TRIM(@R0.5)
$Date=TRIM(@R0.1)
$Trans=$Trans | $Quan | " at " | $Price | " on " | $Date | "."
```

The first step it performs here is to extract the date the transaction was recorded from the string variable \$Date and place it in the first field of the current DynaComm table record. This is easy because we know that the date will always occupy the last eight character positions of that string, so the script uses the LENGTH function to determine the length of \$Date, and then sets the first field in the current record equal to SUBSTR(\$Date,%Temp-8,8), the part of \$Date that starts eight characters from the end of the string and extends for eight characters.

Next, the routine has to determine whether the transaction was a purchase or a sale, what stock was purchased or sold, and how many shares were involved—all of which it can extract from \$Order. Since the first four characters of \$Order will always be either “Buy” or “Sell”, it is easy to determine the nature of the transaction using SUBSTR, and to assign that to the second field in the current table record.

Determining the stock symbol and the quantity bought or sold is a bit trickier, since locations of that data in the string \$Order vary according to the length of the string (which itself varies according to the nature of the transaction, the stock symbol, and the number of shares bought or sold—all of which can be of variable length). Still, there are certain things we know about those locations. By studying several such records, I determined that the quantity of shares involved will always start at position 15 or 16, and there will always be at least 29 characters in the string other than those that indicate the quantity. So the routine extracts characters from the string starting at position 15 and extending for a number of characters determined by subtracting 29 from the length of the string (thus, if the string is 33 characters long, it extracts characters 15 through 18) and assigns them to the third field of the current record.

I also knew that the stock symbol would always start 13 or 14 characters from the end of the string (depending upon whether it is a 3-character symbol (IBM) or a 4-character one (LOTS)), so the routine extracts 4 characters starting 14 characters from the end of the string, and assigns them to the fourth field in the current table record. Then it uses the FILTER command to strip any extraneous spaces from the beginning and/or the end of that record, so that, for instance, either “ IBM” or “IBM ” would become “IBM”.

Next the routine has to extract the price paid or received per share from the string in \$Price. This is relatively easy because the price will always start at position 13 of the string and there are always 28 characters other than the price in the string. So the routine simply extracts the substring that starts at position 13 and extends a number of characters, which it determines by subtracting 28 from the total length of the string, and assigns the result to \$Price. Then, just in case the period that follows the price is part of the extracted substring, it uses the PARSE command to divide \$Price into two substrings—the first consisting of the characters to the left of the period, the second of the characters to the right—and redefines \$Price yet again to consist of the former. Then it assigns the value of \$Price to the fifth field in the current table record.

Finally, the routine must extract the commission paid on the transaction from the \$Commis string. Depending on whether the price per share was an even dollar amount (32) or a fractional amount (32 1/4), the word “was” might appear at the end of the \$Price line or at the beginning of the \$Commis line (as in the message in Figure 13.12). So the routine checks to see whether “was” appears at the beginning of \$Commis, and if it does, strips it off using SUBSTR. Then, no matter what the original \$Commis string contained, it uses SUBSTR to strip off the dollar sign and period from the string, and assigns the resulting number to the sixth field of the current table record.

Next, the routine builds the string it will place in the Last Transaction: field for the stock involved in this transaction. If the second field of the current table record contains "Sell", it starts the string with "Sold ". Otherwise, the routine starts the string with "Bought ". Next it assigns the values of other fields to \$Quan (the number of shares bought or sold), \$Price (the price per share), and \$Date (the transaction date), using the TRIM function to remove any leading or trailing spaces from all three strings. Then it assembles the final string by using the DynaComm string concatenation command (indicated by the vertical bar). Thus after this process \$Trans would read "Sold 100 at 36 3/4 on 3/20/92."

```
RECORD WRITE 0
ACCESS "EXCEL" "BROKER1.XLS" %Channel
IF @R0.4="LOTS" POKE $Trans TO %Channel "LOTSTarget"
IF SUBSTR(@R0.4,1,3) ="IBM" POKE $Trans TO %Channel "IBMTarget"
ACCESS CANCEL %Channel
$Trans=""
SEND ""
WHEN STRING "received" GOTO LOOP
WAIT STRING "you have"
WHEN CANCEL STRING
```

Next the routine saves the current table record (which DynaComm maintains in a temporary file), and opens a DDE channel to Excel. Then, depending on the contents of the fourth field of the record, it instructs Excel to copy the string \$Trans into one of two named ranges: LOTSTarget or IBMTarget, which correspond to the Last Transaction: fields on the two Stock screens. Then it cancels the DDE session, clears the string \$Trans, and sends a carriage return to tell Quick*Way to proceed. It then monitors the text it receives from Quick*Way, looking for the string "received", which would indicate that another transaction message is waiting. If it gets that string, it jumps back to the beginning of the Loop routine and processes the new message. Otherwise, if it receives the message "You have no more Messages", it cancels the string condition.

```
SEND ""
WAIT QUIET "1"
SEND ""
WAIT QUIET "1"
SEND "0"
WAIT QUIET "1"
SEND "0"
$MESSAGE='[MESSAGE(TRUE,"Transaction(s) recorded. Hanging up.")]'
PERFORM Messagebar($Message)
SEND "/off"
WAIT QUIET "1"
HANGUP
ACCESS "EXCEL" "BROKER1.XLS" %Channel
INSTRUC %Channel '[BEEP()]'
```

```

INSTRUCT %Channel '[MESSAGE(FALSE)]'
TABLE SAVE 0 TO CLIPBOARD AS SYLK
INSTRUCT %Channel '[SELECT("R35C1")]'
INSTRUCT %Channel '[PASTE()]'
INSTRUCT %Channel '[RUN("BROKER1.XLM!Trans")][BEEP()]'
ACCESS CANCEL %Channel
RETURN

```

If no more messages are waiting, the routine backs out of the Quick*Way service by sending a pair of returns, followed by a 0. Then it updates the Excel status bar, sends the /off command to CompuServe, and hangs up.

Now all that remains is to copy the contents of the data table into Excel. So the routine opens a DDE channel to BROKER1.XLS and instructs Excel first to beep the PC's speaker and then to clear its status line. Next the routine saves the data table to the Windows Clipboard in SYLK format (SYLK is a data exchange format used by Excel), then instructs Excel to select cell A35 and paste the contents of the Clipboard there. Finally, the routine concludes by instructing Excel to run the macro called Trans on BROKER1.XLM, then terminate the DDE connection and return to the Select_Task routine, which called it.

The Fail Routine

Fail is performed only if no messages were waiting for the user on Quick*Way when the script was instructed to update transaction records:

```

*Fail
$Message='[BEEP()]'
PERFORM Messagebar($Message)
$Message='[MESSAGE(TRUE,"Transaction not yet recorded. Hanging up.")]'
PERFORM Messagebar($Message)
SEND "0"
WAIT QUIET "1"
SEND "/off"
HANGUP
$Message='[BEEP()]'
PERFORM Messagebar($Message)
$Message='[MESSAGE(FALSE)]'
PERFORM Messagebar($Message)
RETURN

```

Fail starts off by instructing Excel to beep the PC's speaker, and then to post the message "Transaction not yet recorded. Hanging up." on its status line. Next the routine exits from Quick*Way and disconnects from CompuServe. Then it instructs Excel to beep the PC's speaker again and clear its message bar. Finally, it exits, returning to the Switch_Task routine.

Wrapping Up Windows Broker

Windows Broker is more like the beginning of a full-fledged stock-trading system than a finished application. It does its job well, but what it does is fairly limited. Certainly a full-scale system would have to be able to work with more than two securities, and should provide easy ways to add additional securities to the portfolio, view a complete transaction history, and perform additional analytic functions.

Fortunately, all those features and more are possible, thanks to the tremendous flexibility afforded the developer by the combined power of the Excel and DynaComm languages. By combining tools in this manner, you can use them to create applications that exceed in range and scope anything that would be possible using a single product's macro language. And although this does entail the complication of having to learn more than one language, it still saves you tremendous amounts of coding time and effort compared to developing a similar application from scratch using a general-purpose program-development tool.

C H A P T E R

14

Enhancing Applications— DocMan

Functional Requirements

A Tour of DocMan

*Exploring
DCGLOBAL.BAS*

Exploring FORM1.FRM

*Exploring
DOCMAN2.FRM*

*Exploring
ACTIONS.FRM*

*Exploring
FINDDL.G.FRM*

*Exploring
GLOBCODE.BAS*

Inside Ami Pro

Wrapping Up DocMan

DOCMAN IS A GLIMPSE INTO A BETTER WORLD—ONE IN WHICH YOUR interaction with your PC will not be ruled by DOS's ridiculous eight-letter file names and mind-numbing directory structures—or even by applications. Instead, in this better world, documents will rule. In other words, the computer will conform to your needs. You'll name and access documents according to what makes sense to you, and let the computer take care of its own housekeeping.

DocMan also serves as an example of how to build a custom application that interacts with and modifies the behavior of an existing commercial application. In this case, DocMan (written in Visual BASIC) serves as a front end for Lotus's Ami Professional word processor. DocMan provides a way for you to select Ami Pro documents to open, print, or delete on the basis of 120-character titles, 300-character descriptions, and as many as four indexed keywords, rather than by their 8-character file name.

Opening Moves

For the most part, Windows does a good job of shielding the user from the complexities and limitations of DOS. However, except for the long labels one can apply to Program Manager icons, it does nothing to address the problem of DOS's eight-letter limit for file names.

For instance, say you're creating a report detailing your third-quarter sales analysis for 1992, and you get the crazy notion you'd like to call the file "1992 Third Quarter Sales Analysis Report." DOS won't let you do anything that sensible. Instead, you're stuck with something like Q392SLAN.DOC—which flows off the tongue somewhat less trippingly, and may be considerably more difficult to decipher six months down the road.

Windows itself doesn't do anything to change that. Some Windows applications, including Word for Windows and Ami Pro, do attach document-description or summary sheets to files, but they don't go far enough. They still force you to rely on DOS's eight-letter file names as your primary interface for opening, saving, and finding documents.

This reliance on DOS file names, and the insistence by even the most advanced Windows applications that the user navigate through labyrinthine directory structures to locate files, is an unacceptable anachronism. In a day and age in which applications provide single-button click access to amazingly complex procedures, it is hard to believe—and even harder to accept—that the simple act of opening and closing documents hasn't progressed since the earliest days of personal computing.

In fact, I found it so hard to accept that I decided to do something about it, and thus was born DocMan. Eventually I expect to see the kind of functionality DocMan provides extended to every application. But for now, I have at least have one application (Ami Pro) trained to deal with documents in a more modern and civilized manner.

Functional Requirements

DocMan's functional requirements were very simple. I wanted it to provide a way for users of Windows applications to locate and manipulate documents using long file names, elaborate descriptions, or keywords, rather than standard file names and directory paths.

If you want to provide long file names to an application operating under DOS, there are two ways to go about it: You can build the facility into your application, as WordPerfect did; or you can build a TSR application that tries to intercept the File Open and Save commands of selected applications and impose its own in their place, as World Software did with Extend*A*Name. Both models have their limitations. The former limits the use of long file names to a single application, whereas the latter is subject to the memory constraints and occasional reliability problems of memory-resident software.

The Windows environment provides a third option: the opportunity to create a stand-alone program that acts as a document control center, keeping track of long file names, descriptions, and keywords for your documents, and interacting with other Windows applications via DDE or other mechanisms. By using the document control center to select, open, print, and create documents, the user can take advantage of the extended descriptive elements that it provides.

Many Windows applications provide the user with the ability to customize the application's menus, adding options or modifying the basic function of commands such as File Open, Print, or Save. I took advantage of this capability by designing DocMan to receive messages containing descriptive information about a file at the time the application that created the file saves it to disk. Although the present version of DocMan works only with Ami Pro, I designed it to be extensible to work with nearly any other Windows application.

Selecting Development Tools

If my intent with this project had been to build an application that would only be used with documents created by Ami Pro, I could have built DocMan entirely in Ami Pro's powerful macro language, and saved myself considerable trouble figuring out and coding the interface between Ami Pro and the DocMan application. However, since I wanted to retain the option of extending DocMan to work with other applications, I elected to build it using a more general-purpose application-development tool.

I chose to use Visual BASIC as that tool for several reasons: the ease with which it allows the developer to build an attractive and responsive user interface, the relative ease of writing BASIC code, and Visual BASIC's ability to create a relatively small and speedy stand-alone EXE file. (The final DocMan EXE file is less than 45k in size. Of course, the 265k Visual BASIC runtime engine, VBRUN100.DLL, must also be present on the user's disk.)

In addition, I used the Ami Pro macro language to create a series of four macros that are used to ensure every document the user creates in Ami Pro is registered in DocMan.

A Tour of DocMan

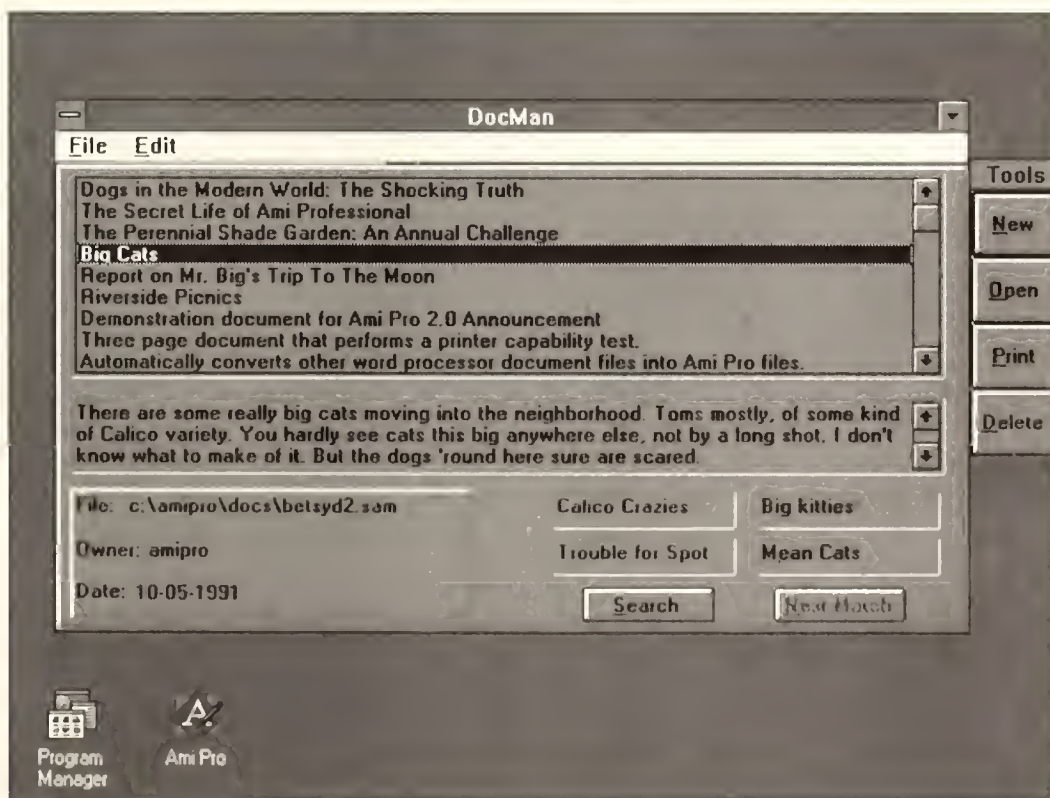
DocMan consists of two primary user-interface screens, plus an opening banner and an About box.

The OpenDM Screen

The first of these two screens features a form named OpenDM and its companion, a floating palette labeled “Tools”, both shown in Figure 14.1. The OpenDM form features a large list box that shows the documents that have been registered in DocMan by their titles, which can each be up to 120 characters long. Immediately beneath the list box is a large editable text field, which is used to present the description (up to 300 characters long) of whichever document is selected in the list box. Beneath that box appear three non-editable text fields, listing the DOS file name of the document, the application that created it, and the date it was created; and four editable text fields presenting the four keywords the user has assigned to the document.

Figure 14.1

The OpenDM form and its companion tools palette



The OpenDM form also includes two command buttons. The first, labeled “Search”, is used to open the other main user-interface element in DocMan, the Find Document dialog box (the FindDlg form), from which the user can initiate a keyword search of DocMan’s document listings. The second command button, labeled “Next Match”, is initially grayed, but will be enabled whenever a successful search has been carried out. It acts as a shortcut that allows the user to find the next document that matches the same criteria as the last successful search.

The tool’s palette, which resides in a separate form file named Actions, initially appears along the right-hand side of the OpenDM form, although the user can reposition it at will. It consists of four buttons: New, Open, Print, and Delete.

You can create a new Ami Pro document at any time by clicking on the New button. When you want to open, print, or delete an existing document, you can do so by either clicking on the corresponding button, or by placing the cursor over a document title in OpenDM’s list box, and then pressing the right mouse button and holding it down while you drag to the desired command button. When you begin to drag the mouse, the mouse pointer changes to the Ami Pro icon. Then when you release the right mouse button, the command button under the mouse pointer will carry out its action (open, print, or delete) on the file you’ve dragged onto it.

In addition, the OpenDM screen has two pull-down menus (shown below). The File menu has New, Open, Print, and Delete options (which can be used in lieu of the Actions form buttons), as well as an Exit option that stops the DocMan application and an About option that opens the About DocMan box. The Edit menu has Copy, Cut, Paste, and Delete options, which allow you to carry out these common editing actions in OpenDM’s editable text fields.

File	Edit
New	Ctrl+N
Open	Ctrl+O
Print	Ctrl+P
Delete	Ctrl+D
Exit	
About	

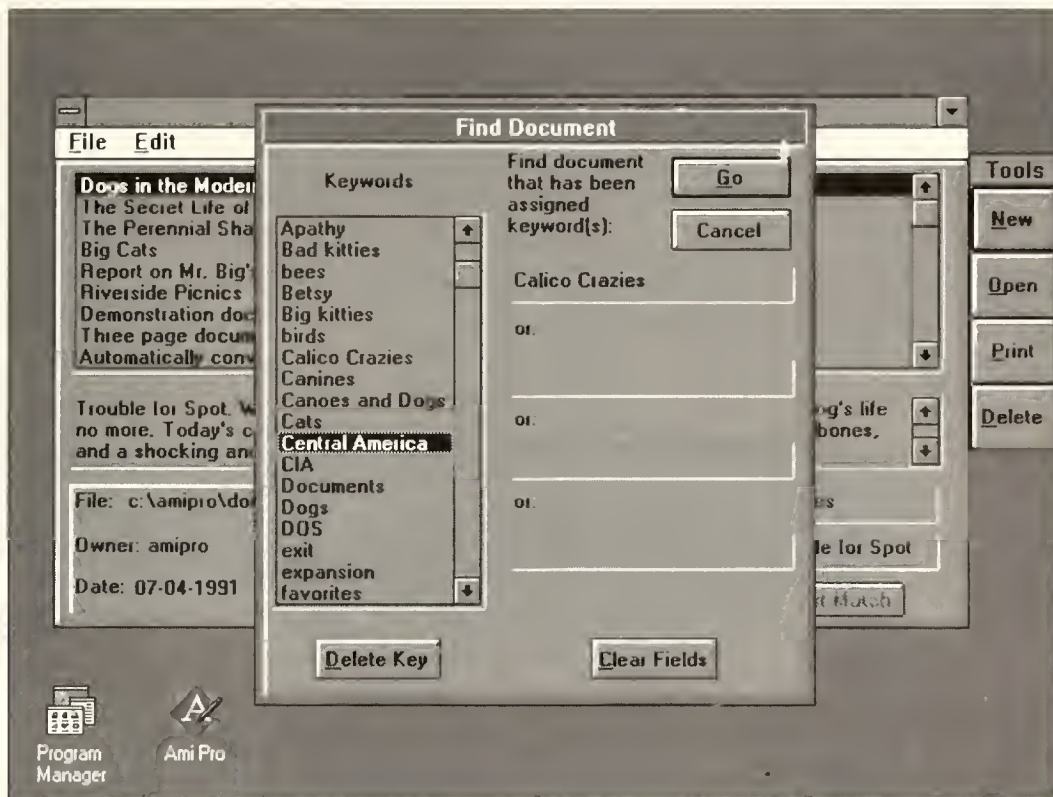
Edit	
Copy	Shift+Ins
Cut	Shift+Del
Paste	Ctrl+Ins
Delete	Del

The FindDlg Screen

The FindDlg form, shown in Figure 14.2, appears when you click the Search button on OpenDM or press Alt-S. It lets you initiate a keyword search of all the documents that have been registered with DocMan. (Actually, the term *keyword* is a bit of a misnomer, since a keyword can be up to 31 characters long and may consist of several words.)

Figure 14.2

DocMan's Find Document dialog box



The list box on the left-hand side of the Find Document dialog box lists all the keywords for every document registered with DocMan. You can select any one of these keywords for use in the search either by double-clicking on it (in which case it will appear in the first empty keyword field on the right side of the dialog box) or by dragging it onto a keyword field while holding down the right mouse button. When you drag a keyword, the mouse-pointer turns to a key shape, and stays that way until you release the right mouse button. The keyword you drag replaces any text currently in the keyword field over which you release it.

In addition (if you're sure of your spelling), you can type a keyword directly into any of the keyword fields.

The keyword searches carried out by DocMan are *OR-based*, meaning that once you've entered the keywords you want to search for and have clicked the Go button, DocMan will locate the first document in its list that has been assigned any of the keywords you have selected. If the document it finds is not the one you wanted, you can click the Next Match button on OpenDM to search for the next matching document, or the Search button to return to the Find Document dialog box and change the search criteria.

FindDlg contains three other buttons besides Go: Cancel, which returns to the OpenDM screen without carrying out the search; Delete Key, which

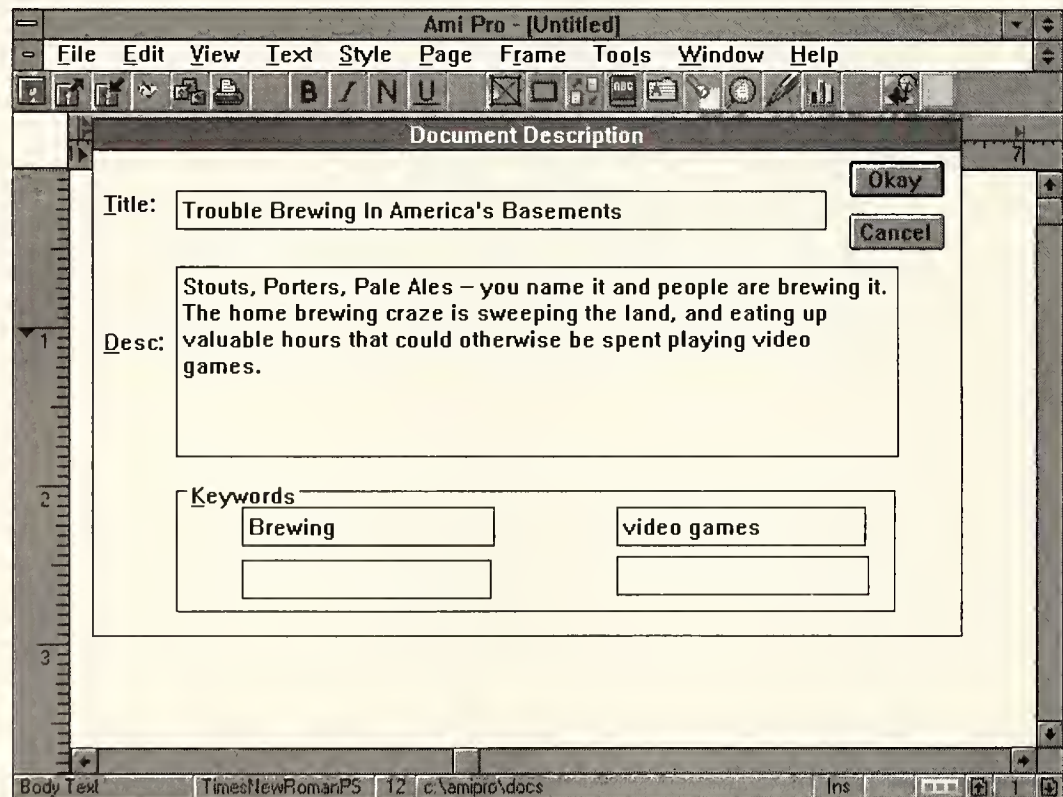
removes the currently selected keyword from the keywords list; and Clear Fields, which clears all four keyword fields on the FindDlg screen.

The Ami Pro Document Description Dialog Box

In addition to the screens that are part of DocMan itself, the application includes one custom screen in Ami Pro: a dialog box containing a form you fill out every time you create a new document in Ami Pro. The form, which was built using the dialog box editor in the Ami Pro Macro Developer's Kit, is shown in Figure 14.3.

Figure 14.3

The Document Description dialog box



The Document Description form contains six user-entry fields: one for the title of the document (up to 120 characters), one for the description (up to 300 characters), and four for the document's keywords. This information, along with the document's DOS file name and path, is all sent to DocMan when you save the file for the first time (thus preventing entries in DocMan for files you elect to abandon without saving), and again whenever you use Ami Pro's Save As option to save the file under a different name.

DocMan's Skeleton

Although DocMan is a considerably more complex application than Recycler (the project described in Chapter 12) it includes the same elements as that earlier Visual BASIC project: a global module, in which the application declares its global variables and defines its external library calls; a form (or in DocMan's case, several forms), consisting of a screen design and the code linked to it; and a code-only module, which contains the code for all the application's general purpose subroutines. These modules are examined in the sections that follow. Also essential to DocMan are four macros written in Ami Pro. These are discussed at the end of the chapter.

Exploring DCGLOBAL.BAS

DocMan's global module is called DCGLOBAL.BAS, and contains the declarations for a series of constants, variables, and external library functions that are used throughout DocMan.

Constant Declarations

DCGLOBAL.BAS begins—like every good code module in Visual BASIC—by declaring that all variables in the file are integers unless stated otherwise. It then sets out to declare a series of constants for use throughout DocMan:

```
DefInt A-Z
Global Const MB_OK = &H0
Global Const MB_OKCANCEL = 1
Global Const MB_ABORTRETRYIGNORE = 2
Global Const MB_ICONEXCLAMATION = 48
Global Const IDOK = 1
Global Const IDCANCEL = 2
Global Const IDABORT = 3
Global Const IDRETRY = 4
Global Const IDIGNORE = 5
Global Const TRUE = 1
Global Const FALSE = 0
Global Const SHIFT_MASK = 1
Global Const CTRL_MASK = 2
Global Const ALT_MASK = 4
```

The first four Const declarations identify four different message box-style parameters used in various combinations throughout the application. The MB_OK style, for instance, is used to indicate that a message box should include only an OK button, and the MB_ABORTRETRYIGNORE style indicates that the message box should have three buttons: Abort, Retry,

and Ignore. MB_ICONEXCLAMATION indicates that the message box should include the standard exclamation mark icon in addition to whatever button(s) it contains.

The next five Const definitions are used to evaluate the result of a message box, which button the user pressed. If the integer value returned by the message box is equal to IDOK (1), then the user selected the OK button, and if it is equal to IDRETRY (4), the user selected the Retry button.

The constants TRUE and FALSE are used throughout the application to evaluate the results of functions that return a value of either -1 (TRUE) or 0 (FALSE).

Finally, the SHIFT_MASK, ALT_MASK, and CTRL_MASK constants are used to evaluate the results of the SHIFT argument in aKeyUp message to determine which (if any) of the three corresponding keys were pressed at the time the KeyUp message was triggered. (Despite its name, SHIFT indicates the status of the Ctrl and Alt keys in addition to that of the Shift key.)

External-Function Declarations

Next, DCGLOBAL.BAS declares a series of external functions:

```
Declare Function FindWindow Lib "USER" (ByVal CName As Any, ByVal Caption As Any) As Integer
Declare Function IsIconic Lib "USER" (ByVal hWnd As Any) As Integer
Declare Function PostMessage Lib "USER" (ByVal hWnd As Integer, ByVal wParam As Integer, ByVal lParam As Integer, ByVal wParam As Integer, lParam As Any) As Integer
Declare Function GetClassWord Lib "USER" (ByVal hWnd As Integer, ByVal nIndex As Integer) As Integer
Declare Function GetModuleFilename Lib "KERNEL" (ByVal hModule As Integer, ByVal lpFilename As String, ByVal nSize As Integer) As Integer
Declare Function SetActiveWindow Lib "USER" (ByVal hWnd As Integer) As Integer
```

The FindWindow function in the Windows USER.EXE library is used to obtain the handle of an open window. The function can search for the window using either its caption or its class name (each Windows application has a unique class name) or both.

The IsIconic function indicates whether or not the window specified in its hWnd parameter is minimized.

The PostMessage function places the Windows message specified in its wParam parameter in the application message queue for the window specified in its hWnd parameter. The wParam and lParam parameters are used to specify any parameters required by the wParam message.

The GetClassWord function can be used to retrieve a variety of information about the window specified in its hWnd parameter. DocMan uses it to obtain its own module handle (that is, the module handle of the OpenDM window), which it then passes to the next function declared here, GetModuleFilename, to obtain its own fully qualified file name (that is, the full path of the DOCMAN.EXE file). From that it can extract the path in which its

DOCMAN.EXE file is stored, which it needs in order to know where to look for its data files.

Finally, the SetActiveWindow function is used to activate the window designated by its hWnd parameter. DocMan uses it to make Ami Pro the active application if it is running in the background when you instruct DocMan to open or print a file or create a new file.

Next, DCGLOBAL.BAS declares two constants that are used in conjunction with the PostMessage function described above:

```
Global Const WM_SYSCOMMAND = &H112
Global Const SC_RESTORE = &HF120
```

WM_SYSCOMMAND and SC_RESTORE are used in conjunction to indicate that the application specified by the hWnd parameter of the PostMessage function should restore its window to its normal size. WM_SYSCOMMAND is used as the wParam parameter to PostMessage, whereas SC_RESTORE is used as the lParam parameter.

Data-Storage Declarations

Next, DCGLOBAL.BAS contains a series of declarations that DocMan uses to create and manage its data files. First it declares the global constant RANDOMFILE, which is used to indicate that a file is to be opened in random-access mode:

```
Global Const RANDOMFILE=4
```

Then it defines a custom data type called DocRec that will be used to store data about each document registered in DocMan:

```
Type DocRec
RecordNum As Long
Title As String * 120
Description As String * 300
File As String * 60
Owner As String * 8
Date As String * 10
Key1 As String * 31
Key2 As String * 31
Key3 As String * 31
Key4 As String * 31
End Type
```

This definition states that each record of type DocRec will start with its record number, followed by a 120-character title, a 300-character description, a 60-character file name and path, an 8-character field called Owner (which

will be used to identify the application that created the document), a 10-character date, and four 31-character keyword fields.

Next, DCGLOBAL.BAS creates a variable called RecordVar using the newly defined record type. This variable will be used to hold records as they are loaded into memory or saved to disk.

```
Global RecordVar As DocRec
```

Variable Declarations

Finally, DCGLOBAL.BAS concludes by declaring a series of variables that will be used in various routines throughout DocMan:

```
Global EditFile As String
Global OpenFileNum As Integer
Global CleanUpFileNum As Integer
Global FileNum As Integer
Global RecordNumber As Long
Global LastRecord As Long
Global NewRecordFlag As Integer
Global WhichRecChanged As Integer
Global Keyword As String
Global FindNext As Integer
Global ExePath As String
```

The purpose of these variables will be explained in the descriptions of the program code in which they are used.

Exploring FORM1.FRM

Because DocMan goes through a number of elaborate steps as it loads the OpenDM form into memory, the process of doing so can take several seconds. Rather than leaving the user facing a blank screen throughout that time, I created a small form called FORM1.FRM, which is displayed during the program-loading process and then hidden as soon as OpenDM is completely loaded into memory. FORM1.FRM simply displays a brief message about the application, as shown in Figure 14.4.

FORM1.FRM contains only two blocks of code: Form_Load and Form_Paint. Since FORM1.FRM is designated as the startup form for DocMan, it is the first part of the application that loads into memory, and its Form_Load procedure is the first to be executed:

```
Sub Form_Load ()
CenterForm Form1
End Sub
```


Figure 14.4
DocMan's opening
banner



Form_Load simply calls the CenterForm routine in the GLOBCODE.BAS file, which centers the form on the screen.

Once the Form_Load procedure has been executed, the Form_Paint routine is performed:

```
Sub Form_Paint ()  
Label1.Refresh  
Label2.Refresh  
Load OpenDM  
OpenDM.Show  
End Sub
```

The Label1.Refresh and Label2.Refresh commands ensure that the text on the form is completely painted before the next statement, Load OpenDM, is executed. Without them, the application would charge ahead and load OpenDM before the opening banner was completely drawn, leaving the user looking at an empty square rather than the pithy text that FORM1.FRM is supposed to display.

The Load OpenDM command loads the OpenDM form into memory, and executes its Form_Load procedure. Once that process is complete, the next command, OpenDM.Show, displays the OpenDM form to the user.

Exploring DOCMAN2.FRM

The OpenDM form is the mainstay of the DocMan application. Through it the user can open, print, delete, and create files; initiate searches; and access every other aspect of the application. OpenDM's screen and program code are stored in a file called DOCMAN2.FRM

General and Loading Routines

In addition to event-related subroutines such as Form_Load or Command1_Click, every Visual BASIC form file has a general-routines area that can be used to declare variables or external functions for use with the current form only, or for use with subroutines called only by other routines in the current form.

OpenDM's general-declarations statement contains only this one line:

```
DefInt A-Z
```

You may be wondering why I repeated this declaration here, even though it already appears in the DCGLOBAL.BAS file. Unlike every other declaration in an application's global module, DefInt is always applied locally. In other words, the DefInt A-Z in DCGLOBAL.BAS applies only to variables defined in that file. If you want DefInt A-Z to take effect throughout your application, you must declare it in every form and code module in your application. (FORM1.FRM had no DefInt only because its procedures use no variables of any type.)

The Form_Load Procedure

OpenDM's Form_Load procedure is executed as OpenDM is loaded into memory.

```
Sub Form_Load ()
CenterForm OpenDM
GetPath
Load FindDlg
CenterForm FindDlg
Load Actions
Load AboutDLG
Actions.Left = OpenDM.Left + OpenDM.Width + 50
Actions.Top = OpenDM.Top + 100
Actions.Show 0
```

It starts by calling CenterForm in GLOBCODE.BAS to center OpenDM on screen. Next it calls the GetPath routine (also in GLOBCODE.BAS), which determines the directory path DocMan is stored in. Then it loads the Find Document dialog box into memory, and calls CenterForm again to make

sure the dialog box will be centered on screen when it is displayed. Next, the routine loads the AboutDLG and Actions forms into memory, positions the Actions form at the right edge of the OpenDM window, and issues the Actions.Show command to display it.

Next, OpenDM reads the DOCMAN.DAT file into memory. DOCMAN.DAT contains the title, description, and keyword data for every document that has been registered with DocMan.

```
RecordLen = Len(Recordvar)
FileNum = Fileopener(ExePath + "DOCMAN.DAT", RANDOMFILE,
    RecordLen)
Lastrecord = LOF(FileNum) \ Len(Recordvar)
For I = 1 To Lastrecord
    Get FileNum, I, Recordvar
    Titles.AddItem RTrim$(Recordvar.Title)
Next I
```

The data-access procedure starts by determining the length of the global variable Recordvar (which was earlier defined as being of the custom record type DocRec). Then it calls the Fileopener routine in GLOBCODE.BAS, passing it the path and name of the file to open, the mode of access it should use, and the length of each data record.

Next, the routine determines the number of records in the DOCMAN.DAT file by dividing the total length of the file by the length of RecordVar (which is the length of a single record). Then it loops through the file, reading each record and adding the document-title portion of the record to the Titles list box on OpenDM using Visual BASIC's AddItem command, before reading the next record.

When all the records have been read, OpenDM's Form_Load routine concludes, as follows:

```
WhichRecChanged=-1
If Lastrecord > 0 Then Titles.Listindex = 0
ReadSelectedRecord
Command2.Enabled = False
Form1.Hide
End Sub
```

First it sets the WhichRecChanged variable (which, as its name suggests, points to a changed record) to -1, indicating that the current record has not changed, and then, as long as at least one record has been read, sets the ListIndex property of the Titles list box to 0. The Listindex property determines which item in the list box is highlighted. Listindex numbering starts from zero, so this command simply selects the first item in the list, as long as the list isn't empty. The routine has to set WhichRecChanged to -1

before setting the Listindex property, because setting Listindex automatically triggers the Click routine for the list box, which saves the current record back to disk if WhichRecChanged is greater than -1.

Next, the routine calls the ReadSelectedRecord subroutine in GLOB-CODE.BAS to load the description, keywords, and other data for the current record into the proper edit boxes on the form. Then it disables the Next Match button (Command2) and hides the opening banner.

The Form_Paint Procedure

Form_Paint is automatically executed once the Form_Load routine has been completed.

```
Sub Form_Paint ()
AutoRedraw = True
Call Frame(OpenDM, 30, 30, OpenDM.Height - 670, OpenDM.Width - 100, 2)
Call Frame(OpenDM, Titles.Left - 30, Titles.Top - 30, Titles.Height + 60,
Titles.Width + 60, 1)
Call Frame(OpenDM, Description.Left - 30, Description.Top - 30, Description.Height
+ 60, Description.Width + 60, 1)
Call Frame(OpenDM, Label1(4).Left - 30, Label1(4).Top - 30, (Text1(2).Top -
Text1(0).Top) + Text1(2).Height + 60, Text1(0).Left + Text1(0).Width + 60, 2)
For X = 4 To 7
Call Frame(OpenDM, Text1(X).Left - 15, Text1(X).Top - 15, Text1(X).Height + 30,
Text1(X).Width + 30, 1)
Next X
End Sub
```

Form_Paint begins by setting the AutoRedraw flag for OpenDM to True. Doing so instructs Visual BASIC to maintain a bitmapped image of the form in memory and to henceforth use that image to restore DocMan's screen display, rather than reexecuting the commands that follow. This eliminates the need to redraw the OpenDM screen when DocMan is restored from an iconized state or brought to the top after having been obscured by another window, thus speeding DocMan's display.

The remaining commands in the Form_Paint routine call the Frame routine in GLOB-CODE.BAS and instruct it to draw the lines that give the OpenDM form and its controls a three-dimensional appearance. The parameters to the Frame routine tell it the name of the form on which the frame should be drawn, the x-y coordinate of the top-left corner of the frame, the frame's height and width, and finally the frame's style (either a 1 or a 2, depending on whether a thin concave frame or a thick convex frame is desired).

User-Action Routines

Once the Form_Paint procedure has finished, the process of loading DocMan is complete, and all remaining routines occur only in response to user actions.

The Titles_GotFocus Procedure

Titles_GotFocus is called whenever the input focus is given to the Titles list box. This occurs when the user clicks on the list box or tabs to it.

```
Sub Titles_GotFocus ()
  MenuCut.Enabled = False
  MenuCopy.Enabled = False
  MenuPaste.Enabled = False
  MenuDelete.Enabled = False
End Sub
```

The routine disables the Cut, Copy, Paste, and Delete options on DocMan's edit menu, because they cannot be used to edit or otherwise change the contents of the Titles list box. The menu items appear grayed out when disabled, as shown here:

Edit	
<u>C</u> opy	Shift+Ins
C <u>u</u> t	Shift+Del
<u>P</u> aste	Ctrl+Ins
<u>D</u> elete	Del

The Titles_Click Procedure

Titles_Click is called in response to the user selecting a new record by clicking on the Titles list box or by moving the list box's selection bar using the cursor keys.

```
Sub Titles_Click ()
  If WhichRecChanged > -1 Then
    WriteChangedRecord
  End If
  ReadSelectedRecord
End Sub
```

Titles_Click starts by checking whether the current record has been changed (in which case WhichRecChanged would be greater than -1) and, if so, calls the WriteChangedRecord routine in GLOBCODE.BAS to save the new contents of the record to disk. Then it calls the ReadSelectedRecord routine to read the description, keywords, file name, and other data for the newly selected record.

The Titles_LostFocus Procedure

Titles_Lostfocus is called when the input focus moves to another control. This occurs in response to the user clicking on or tabbing to the other control.

```

Sub Titles_LostFocus ()
MenuCut.Enabled = TRUE
MenuCopy.Enabled = TRUE
MenuPaste.Enabled = TRUE
MenuDelete.Enabled = TRUE
End Sub

```

The `Titles_LostFocus` routine simply reenables the menu items the `GotFocus` routine disabled.

The Titles_MouseMove Procedure

`Titles_MouseMove` is called whenever the user moves the mouse over the `Titles` list box.

```

Sub Titles_MouseMove (Button As Integer, Shift As Integer, X As Single, Y As Single)
If Button <> 2 Then Exit Sub
Titles_Click
Getfile
If Editfile = "" Then Exit Sub
Titles.Drag
End Sub

```

The routine begins by checking the value of the `Button` argument to determine if the right-hand mouse button is down (`Button = 2`). If not, the routine ends. (Using only the right mouse button for drag operations allows the user to use the left mouse button for the standard operations of selecting list-box items and scrolling the list box.)

If the right-hand mouse button is down, the routine calls the `Titles_Click` routine to save the current record, if necessary, and then calls the `GetFile` routine in `GLOBCODE.BAS` to obtain the title of the document under the mouse pointer. If no document is highlighted, the routine ends. Otherwise it calls the `Titles.Drag` method, which automatically tells Visual BASIC to change the mouse pointer to the specified drag icon for the list box (the Ami Pro icon) and to allow the user to drag the icon across the screen. This icon was specified during the design process, by setting the `DragIcon` property for the listbox, as shown in Figure 14.5.

The Command1_Click Procedure

`Command1_Click` is called when the user clicks on the button labeled “Search”.

```

Sub Command1_Click ()
If WhichRecChanged > -1 Then WriteChangedRecord
If FindDlg.Text1(0).Text + FindDlg.Text1(1).Text + FindDlg.Text1(2).Text +
FindDlg.Text1(3).Text = "" Then
OpenDM.Titles.listindex = 0

```



```

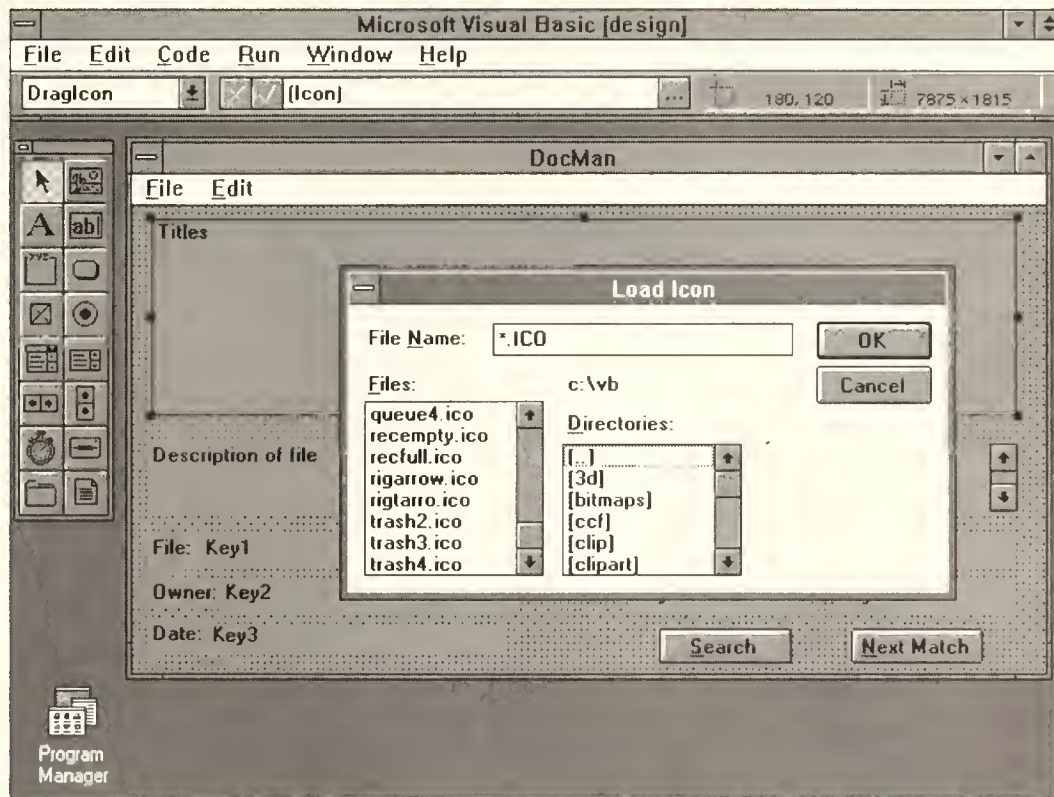
Else FindNext = 1
End If
FindDlg.Show 1
End Sub

```

Command1_Click first checks the status of WhichRecChanged and, if it doesn't equal -1, calls the WriteChangedRecord routine in GLOBCODE.BAS.

Figure 14.5

Setting the drag icon for the Titles list box



Next, it combines the contents of the four keyword fields on FindDlg and checks to see whether together they equal "", which would indicate that all four are empty. If so, it moves the selection bar to the top of the Titles list box, so that the search will start with the first record and proceed from there. Otherwise, it sets the flag FindNext equal to 1, in order to instruct the search routine to begin searching at the record following the current record. Then it issues the FindDlg.Show 1 command, instructing Visual BASIC to display the Find Documents dialog box. The parameter 1 specifies that FindDlg should be modal, meaning that no input will be allowed in any of DocMan's other forms until the user has closed the Find Document dialog box. If the parameter had been 0 or missing, the form would be modeless, like the Actions form.

The Command2_Click Procedure

Command2_Click is called in response to the user clicking on the Next Match button.

```
Sub Command2_Click ()
  If WhichRecChanged > -1 Then WriteChangedRecord
  FindNext = 1
  FindRecord
End Sub
```

FindNext checks the status of WhichRecChanged, and calls WriteChangedRecord if necessary. Then it sets FindNext equal to 1 and calls the FindRecord routine in GLOBCODE.BAS, which searches for the next document matching the specification of the previous search.

The Description_Change Procedure

Description_Change is called whenever the user changes the contents of the description field for the current record.

```
Sub Description_Change ()
  WhichRecChanged = Titles.ListIndex
  TestLength OpenDM.Description, 300
End Sub
```

The routine begins by setting the value of WhichRecChanged to the ListIndex property of the Titles list box (thus identifying the record that needs to be saved when the user selects another record), and then calls the TestLength routine in GLOBCODE.BAS, passing it the full name of the Description field (OpenDM.Description) and the maximum length allowed for its contents. TestLength ensures that the user doesn't enter more data than is allowed in the field.

The Text1_Change Procedure

Text1_Change is called whenever the user changes the contents of any of the four keyword fields—Text1(4) through Text1(7).

```
Sub Text1_Change (Index As Integer)
  WhichRecChanged = Titles.listindex
  TestLength OpenDM.Text1(Index), 31
End Sub
```

The routine sets WhichRecChanged equal to Titles.ListIndex, and then calls TestLength to ensure that the user hasn't entered more than 31 characters in the field.

The Text1_GotFocus Procedure

Text1_GotFocus is called whenever any of the seven fields that share the name Text1 receive the input focus.

```
Sub Text1_GotFocus (Index As Integer)
  If Index < 4 Then
    Beep
    Text1(4).SetFocus
  Else
    Text1(Index).SelStart = 0
    Text1(Index).SelLength = 31
  End If
End Sub
```

The first three Text1 fields are Text1(0), Text1(1), and Text1(2), which correspond, respectively, to the File, Owner, and Date fields to the left of the keyword fields, all of which are read-only—that is, user input to these fields is not allowed. (Once upon a time there was a Text1(3) too, which held other data, but it was eliminated early in the development process.)

The Text1_GotFocus routine begins by evaluating the value of Index and, if it is less than 4 (indicating that the user tried to set the focus to one of the three read-only fields), it beeps the PC's speaker, moves the input focus to Text1(4), and exits. Otherwise, it highlights all the text in the selected field, as shown in Figure 14.6.

The Form_LinkExecute Procedure

Form_LinkExecute is the most complex procedure in the DOCMAN2.FRM listing. It is called in response to the OpenDM form's receiving a DDEExecute message from Ami Pro.

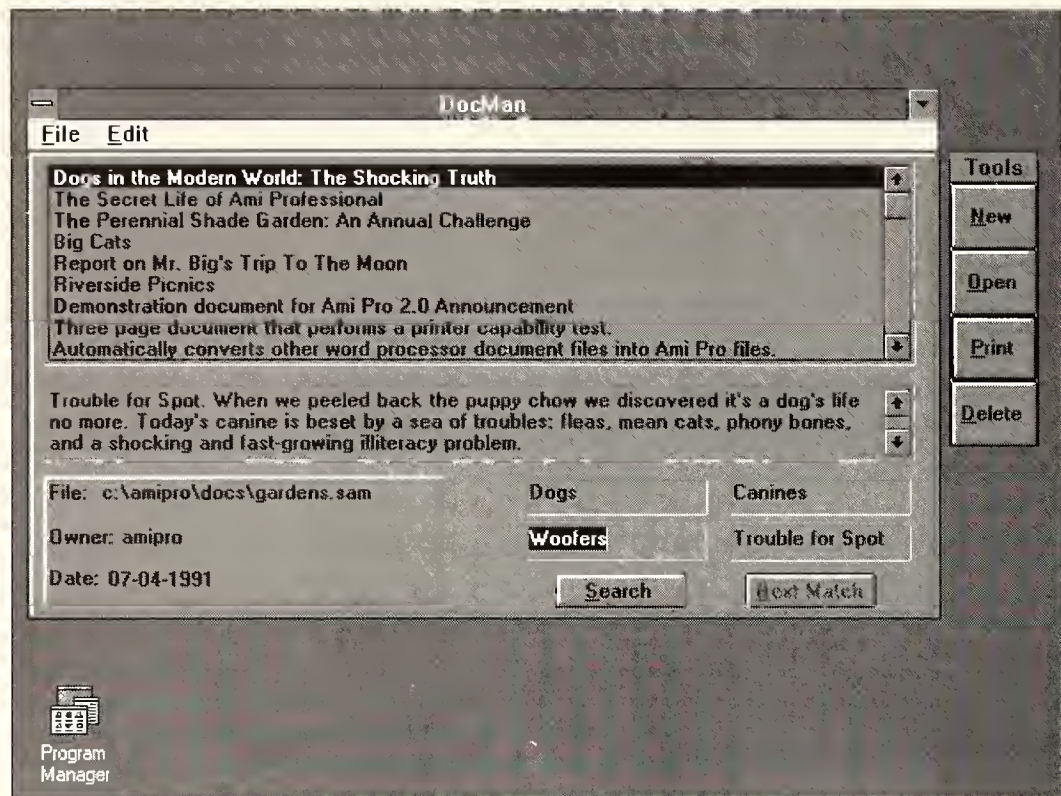
The Ami Pro macro SaveWithNewName (described near the end of this chapter) is called when the user saves a new file for the first time or saves an existing file under a new name. The macro establishes a DDE link to the OpenDM form, and sends a series of DDEExecute command lines to DocMan.

The first of these command lines starts with the text "titl=", followed by the document's title. This is followed by "appl=", along with the name of the application that created the document (which will always be "amipro" in this version of DocMan). Then follows the document's DOS file name, 300-character description, and four keywords. Ami Pro then sends one final command string, "show", which instructs DocMan to process the data it has just received.

DocMan receives each command string in turn as the CmdStr argument to the LinkExecute message. So the Form_LinkExecute routine evaluates the contents of each CmdStr as it arrives, using a long Select Case statement.

Figure 14.6

The text in each keyword field is highlighted as the field receives the input focus



```

Sub Form_LinkExecute (Cmdstr As String, Cancel As Integer)
  Select Case Left$(Cmdstr, 4)
  Case "titl"
    NewRecordFlag = False
    If WhichRecChanged > -1 Then WriteChangedRecord
    NewRecordFlag = TRUE
    Titles.AddItem Mid$(cmdstr, 6, 120)
    Lastrecord = Lastrecord + 1
    Recordvar.Recordnum = Lastrecord + 1
    OpenDM.Titles.Listindex = Lastrecord - 1
    Recordvar.Title = Mid$(Cmdstr, 6, 120)
  
```

The “titl” string is the first command string sent by Ami Pro, so DocMan knows when it receives a command starting with those four characters that the commands that follow are for a new record. The routine begins by setting the variable NewRecordFlag to False, and then calls WriteChangedRecord, if necessary, before setting NewRecordFlag to True.

WriteChangedRecord exits immediately if NewRecordFlag is True, so the routine has to set it to False before calling the routine in order to force WriteChangedRecord to save the current record.

Next, the routine extracts 120 characters of text from the command string, starting at the sixth character (the character immediately to the right of the = sign), and calls the Visual BASIC AddItem command, which adds the string as a new item at the bottom of the Titles list box.

Then it increments the record count and the value of RecordNum (which indicates the current record), moves the selection bar in the Titles list box to the new item, and sets the Title field of Recordvar to the new document title.

At this point, Recordvar (the variable of type DocRec that holds the contents of the current record) contains only the title of the new record. The remaining data in Recordvar is left over from the record that was previously current. So the routine has to obtain the remainder of the data about the new record from Ami Pro before the record can be saved to disk.

```

Case "appl"
Text1(1).Text = Mid$(Cmdstr, 6, 8)
Recordvar.Owner = OpenDM.Text1(1).Text
Case "desc"
Description.Text = Mid$(Cmdstr, 6, 300)
Recordvar.Description = OpenDM.Description.Text
Case "key1"
Text1(4).Text = Mid$(Cmdstr, 6, 31)
Recordvar.Key1 = OpenDM.Text1(4).Text
Case "key2"
Text1(5).Text = Mid$(Cmdstr, 6, 31)
Recordvar.Key2 = OpenDM.Text1(5).Text
Case "key3"
Text1(6).Text = Mid$(Cmdstr, 6, 31)
Recordvar.Key3 = OpenDM.Text1(6).Text
Case "key4"
Text1(7).Text = Mid$(Cmdstr, 6, 31)
Recordvar.Key4 = OpenDM.Text1(7).Text
Case "file"
Text1(0).Text = Mid$(Cmdstr, 6, 99)
Text1(2).Text = Date$
Recordvar.File = OpenDM.Text1(0).Text
Recordvar.Date = OpenDM.Text1(2).Text

```

The procedures for handling the command strings that start with “appl”, “desc”, “key1”, “key2”, “key3”, and “key4” are all fairly straightforward. DocMan uses the Visual BASIC Mid\$ routine to extract the desired text from the command string, add it to the correct edit field on OpenDM for data of that type, and then add it to the appropriate field in Recordvar.

The file= routine is only slightly more complex. First the document’s DOS file name is placed in Text1(0), then the current date (obtained via the

Visual BASIC Date\$ function) is placed in Text1(2). Then the File and Date fields of Recordvar receive the same data, the last pieces necessary to constitute the new record.

```

Case "show"
  If NewRecordFlag = TRUE Then
    Put FileNum, lastrecord, recordvar
    Addkeys
  End If
  OpenDM.Windowstate = 0
  Actions.Windowstate = 0
  Actions.Show
  OpenDM.Show
  AppActivate "DocMan"
  If Newrecordflag = True Then Titles.Listindex = Lastrecord
    - 1
  NewRecordFlag = False
End Select
End Sub

```

When the “show” command string is received, DocMan checks the value of NewRecordFlag and then uses the Visual BASIC Put command to write the new record to disk. (It checks the value of NewRecordFlag because Ami Pro also uses the “show” command string to make DocMan activate itself, not only when a new record is being created. On those occasions I didn’t want the routine to write a record to disk.)

Next, the routine calls the AddKeys function in GLOBCODE.BAS to add the keywords from the new record to the keyword list on the Find Document dialog box. Then it sets the Windowstate property of both the Actions form and OpenDM to 0 (normal), thus restoring them in case they were minimized when OpenDM received the DDE message from Ami Pro.

It then issues the commands Actions.Show and OpenDM.Show, bringing both windows to the top (moving them in front of Ami Pro, which was most likely obscuring them at the time the DDE message came in), and uses the AppActivate command to make DocMan the active application.

Then, if the NewRecordFlag variable is equal to True, it sets the Listindex property of the Titles list box to LastRecord – 1 (placing the selection bar on the new record, since Listindex is numbered from zero and LastRecord counts from one). It then concludes by setting NewRecordFlag to False, ends the Select Case routine, and exits the subroutine.

Thus, at the conclusion of this routine, both DocMan’s main window and its tools palette will be visible on screen, with Ami Pro in the background, and the selection bar in the Titles list box will be on the newly created record.

The Form_Unload Procedure

Form_Unload is called when the OpenDM form receives an Unload message, signaling that the application is to close down.

```
Sub Form_Unload (Cancel As Integer)
Unload FindDlg
Unload Actions
Unload AboutDLG
Unload Form1
End Sub
```

The Form_Unload routine instructs Visual BASIC to unload the four other forms used by the application (FindDlg, Actions, AboutDLG, and Form1) before shutting down. You must explicitly unload every form you load in Visual BASIC; otherwise, they remain in memory even after your application has shut down, consuming RAM and system resources.

The Form_Resize Procedure

Form_Resize is called when the size of the OpenDM form changes. Since OpenDM has a fixed border (rather than being sizeable), this will only occur when the application is minimized or restored.

```
Sub Form_Resize ()
Select Case OpenDM.Windowstate
Case 0
Actions.Show
OpenDM.SetFocus
Case 1
Actions.Hide
End Select
End Sub
```

This routine's only job is to make sure that the Actions tools palette is hidden when you minimize OpenDM, and is visible when you restore it. (DocMan hides the Actions form rather than minimizing it because minimizing it would add a second icon to the bottom of the screen. Hiding it allows you to control both forms from a single icon.)

The routine evaluates the value of the OpenDM.Windowstate property. If it is equal to 0, then the user has just restored the OpenDM window, so the routine makes Actions visible and then sets the input focus to OpenDM. On the other hand, if the Windowstate property is 1, then the user has just minimized OpenDM, so the routine hides the Actions form.

Menu Item Routines

The remaining routines in DOCMAN2.BAS are called in response to the user's menu selections. Most consist of single-word calls to routines in GLOBCODE.BAS.

The ExitMenuItem_Click Procedure

ExitMenuItem_Click is called when the user selects the Exit item on DocMan's File menu.

```
Sub ExitMenuItem_Click ()
ExitDocMan
End Sub
```

The routine reacts to the user's menu selection by calling the ExitDocMan routine in GLOBCODE.BAS.

The AboutMenuItem_Click Procedure

AboutMenuItem_Click is called when the user selects the About item on the File menu.

```
Sub AboutMenuItem_Click ()
AboutDLG.Show 1
End Sub
```

The routine instructs DocMan to display the AboutDLG dialog box in modal form.

The NewMenuItem_Click Procedure

NewMenuItem_Click is called when the user selects the New item on the File menu. It simply calls the NewFile routine in GLOBCODE.BAS.

```
Sub NewMenuItem_Click ()
NewFile
End Sub
```

The PrintMenuItem_Click Procedure

PrintMenuItem_Click calls the PrintFile routine in GLOBCODE.BAS when the user selects the Print item from DocMan's File menu.

```
Sub PrintMenuItem_Click ()
PrintFile
End Sub
```

The DeleteMenuItem_Click Procedure

DeleteMenuItem_Click is called when the user selects Delete from the DocMan File menu.

```
Sub DeleteMenuItem_Click ()
DeleteFile
End Sub
```

DeleteMenuItem_Click calls the DeleteFile routine in GLOBCODE.BAS.

The OpenMenuItem_Click Procedure

OpenMenuItem_Click calls the OpenDoc routine in GLOBCODE.BAS when the user selects the Open item on the File menu.

```
Sub OpenMenuItem_Click ()
Opendoc
End Sub
```

The MenuCopy_Click Procedure

MenuCopy_Click is called when the Copy item on DocMan's Edit menu is selected.

```
Sub MenuCopy_Click ()
SendKeys "^{INSERT}"
End Sub
```

All Visual BASIC edit controls react in the standard manner to the Windows shortcut keys for Copy, Cut, Paste and Delete, so the Copy command simply sends the standard Ctrl-Ins keystroke to the edit control that has the input focus, instructing it to copy any text that the user has selected to the Windows Clipboard.

The MenuCut_Click Procedure

MenuCut_Click, which is called when the user selects the Cut item from the Edit menu, sends the standard Cut (Shift-Del) shortcut to the current edit control.

```
Sub MenuCut_Click ()
SendKeys "+{DEL}"
End Sub
```

The MenuDelete_Click Procedure

MenuDelete_Click is called when the user selects the Delete item on DocMan's Edit menu. It responds by sending the Del keypress, causing the current edit control to delete any text the user has selected.

```
Sub MenuDelete_Click ()
SendKeys "{Delete}"
End Sub
```


The MenuPaste_Click Procedure

MenuPaste_Click, the final routine in DOCMAN2.FRM, is called when the Paste command is selected from DocMan's Edit menu. It sends the Shift-Ins shortcut command, for Paste, to the current edit control, instructing it to paste the contents of the Clipboard at the current text-insertion point.

```
Sub MenuPaste_Click ()
SendKeys "+{INSERT}"
End Sub
```

Exploring ACTIONS.FRM

The Tools palette called Actions is designed to provide the user with simple single-click or drag-and-drop access to the same functions provided by the File menu on the OpenDM form.

ACTIONS.FRM consists of a DefInt A-Z statement and four subroutines: Form_Load, Command1_Click, Command1_DragDrop, and Command1_KeyUp.

The Form_Load Procedure

Form_Load is performed as ACTIONS.FRM is loaded into memory, before it is actually displayed on screen.

```
Sub Form_Load ()
Actions.Width = 825
End Sub
```

The Actions Form_Load routine simply sets the width property to 825 twips (the default unit of measurement in Visual BASIC). *Twips* are logical units that vary according to the resolution of your display. Visual BASIC won't allow you to create a form narrower than about 930 units at design time (the width required to display the Control box and minimize and maximize buttons—which appear on all forms at design time even when you have set the form's property options not to display any of them at runtime, as I did with ACTIONS.FRM).

However, since I wanted the form to be just large enough to hold its buttons, I had two options: I could make the buttons wide enough to fill the narrowest form that Visual BASIC would let me create at design time, or I could adjust the width of the form at runtime to fit the size of the buttons I had created. Making the buttons larger would just have been a waste of screen space, so I chose to adjust the size of the form as it loads into memory.

Button Routines

The four buttons on the Actions form are part of a control array called `Command1`. This means that they share the same event-code routines. Each button has a unique index number (ranging from 0 to 3), which is passed to a shared event routine at the time the routine is called. So, for instance, if you click the New button (index 0), the `Command1_Click` procedure receives 0 as its index parameter.

The `Command1_Click` Procedure

`Command1_Click` is performed when the user clicks any of the four buttons on the form.

```
Sub Command1_Click (Index As Integer)
  Select Case Index
    Case 0
      NewFile
    Case 1
      OpenDoc
    Case 2
      PrintFile
    Case 3
      DeleteFile
  End Select
End Sub
```

`Command1_Click` uses a `Select Case` statement to evaluate the value of the `Index` argument and call the appropriate routine in `GLOBCODE.BAS` (the `NewFile` routine if you clicked the New button, the `OpenDoc` routine if you clicked the Open button, and so on).

The `Command1_DragDrop` Procedure

`Command1_DragDrop` is called when you drop a file dragged from the Titles list box onto any of the four buttons on the Actions form. When you drag a document onto the Open button and drop it, it is clear that you want to open the document, just as if you had clicked the Open button after having selected the document in the Titles list box. Similarly, the meaning of dropping a document onto the Print and Delete buttons is also pretty obvious. But dropping a document onto the New button doesn't make any sense, so the routine screens out `DragDrop` calls with index number 0.

```
Sub Command1_DragDrop (Index As Integer, Source As Control, X As Single, Y As Single)
  If Index > 0 Then Command1_Click (Index)
End Sub
```

If the `Index` argument received by the routine is greater than 0, the routine simply calls the `Command1_Click` routine, passing along the same index

number, and allowing it to determine which routine in GLOBCODE.BAS should be called to process your action. Meanwhile, it simply ignores documents that you drop onto the New button (index 0), since it assumes that you have done so in error.

The Command1_KeyUp Procedure

As I started to work on DocMan, I discovered that the problem of using a separate form such as ACTIONS.FRM as a tools palette is that when the palette has the input focus, the shortcut keystrokes for accessing buttons and menus on the main form don't get processed. So I added the following routine to ACTIONS.FRM, designing it to send every keystroke typed while ACTIONS.FRM has the input focus to the OpenDM form. Doing this in the KeyUp routine for the form's buttons, rather than in their KeyDown or Key-Press routines, gives Actions the opportunity to process the shortcut keys for the ACTIONS.FRM buttons.

```
Sub Command1_KeyUp (Index As Integer, KeyCode As Integer, Shift As Integer)
OpenDM.SetFocus
SendString$ = ""
If Shift And SHIFT_MASK Then SendString$ = "+" + SendString$
If Shift And ALT_MASK Then SendString$ = "%" + SendString$
If Shift And CTRL_MASK Then SendString$ = "^" + SendString$
SendString$ = SendString$ + Chr$(KeyCode)
SendKeys SendString$
End Sub
```

The KeyUp routine is called with three arguments: the index of the command button that has the input focus, the keycode of the key that was pressed, and an integer variable containing a bit field that reflects the state of the Shift, Ctrl, and Alt keys. In order to discover which, if any, of those keys were pressed, the routine must perform a series of three logical AND operations using the SHIFT_MASK, ALT_MASK, and CTRL_MASK constants that were defined in DCGLOBAL.BAS.

The routine begins by giving OpenDM the input focus. Then it assembles a string called SendString\$ by adding the character +, %, or ^ to the empty string if the results of the AND operations indicate that the Shift, Alt, or Ctrl key was pressed. Then it adds a one-character string consisting of CHR\$(KeyCode) to the end of SendString\$. (The CHR\$(KeyCode) function returns character number KeyCode in the ANSI character set.) Finally, it uses the Visual BASIC SendKeys function to send SendString\$ to OpenDM.

That concludes discussion of the routines in ACTIONS.FRM.

Exploring FINDDL.G.FRM

FindDlg, the Find Document dialog box, is loaded into memory as OpenDM loads and is then made visible whenever you press the Search button on the OpenDM form.

General and Loading Routines

FindDlg's general procedures are limited to a simple DefInt A-Z statement. The first procedure executed as it loads is Form_Load.

The Form_Load Procedure

The FindDlg Form_Load Procedure is responsible for loading the DMKEYS.DAT file, which contains the keywords for all the documents that have been registered with DocMan.

```
Sub Form_Load ()
    Keysfilenum = FreeFile
    On Error GoTo Loaderror
    KeysFile$ = ExePath + "DMKEYS.DAT"
    Open KeysFile$ For Input As Keysfilenum
```

The routine begins by using the Visual BASIC FreeFile function to obtain the first available file handle. Then it sets up an On Error routine that will jump to the Loaderror label if an error occurs during the operations that follow. Next, it creates the variable KeyFile\$ by concatenating the contents of ExePath and the string "DMKEYS.DAT", and then instructs Visual BASIC to open KeyFile\$ for read operations and to assign the handle Keysfilenum to it.

```
Do Until EOF(Keysfilenum)
    Line Input #Keysfilenum, Item$
    Item$ = RTrim$(Item$)
    FindDlg.List1.AddItem Item$
Loop
Close Keysfilenum
FindNext = 0
Exit Sub
```

Next, the routine enters a Do loop, in which it reads one line at a time from the newly opened file until it encounters the end-of-file marker. It assigns the contents of each line to the string variable Item\$ as it is read, then trims any excess spaces from the end of Item\$ before adding it to the Keywords list box (FindDlg.List1). Then it loops back and reads the next line. When the end-of-file marker is found, the routine exits the Do loop and closes the file. It then sets the global variable FindNext equal to 0 and exits.

In the event an error occurs during the opening or reading of DMKEYS.DAT, the On Error condition in the FindDlg Form_Load routine jumps to the label “Loaderror:”, which follows the lines listed above.

```
Loaderror:
Action = FileErrors(Err, KeyFile$)
Select Case Action
Case 0
Resume
Case Else
Exit Sub
End Select
End Sub
```

The Loaderror subroutine calls the FileErrors function in the GLOB-CODE.BAS file, passing it the values of Err (the Visual BASIC runtime error code) and KeysFile\$, and assigning the integer value returned by the function to the variable Action. If Action is equal to 0, then the user selected the Ignore or Resume button from a message box created by the FileErrors function, and thus the routine attempts to resume the file-access process. Otherwise, it exits the File_Load subroutine.

The Form_Paint Procedure

Form_Paint is called the first time the FindDlg dialog box is displayed on screen.

```
Sub Form_Paint ()
AutoRedraw = True
Call Frame(FindDlg, 30, 30, FindDlg.Height - 500, FindDlg.Width - 230, 2)
Call Frame(FindDlg, List1.Left - 30, List1.Top - 30, List1.Height + 60, List1.Width + 60, 1)
For X = 0 To 3
Call Frame(FindDlg, Text1(X).Left - 15, Text1(X).Top - 15, Text1(X).Height + 30, Text1(X).Width + 30, 1)
Next X
End Sub
```

The routine begins by setting the form’s AutoRedraw property to True, precluding the need to redraw the form the next time it is displayed. Then it makes a series of six calls to the Frame procedure in GLOB-CODE.BAS to create the appearance of three-dimensional borders around the edges of the dialog box and its controls.

The Form_Unload Procedure

Form_Unload is called when DocMan attempts to unload FindDlg from memory. It reverses the actions of the Form_Load routine by saving the current keyword list to disk.

```
Sub Form_Unload (Cancel As Integer)
Keysfilenum = Freefile
KeysFile$ = ExePath + "DMKEYS.DAT"
On Error Resume Next
Kill KeysFile$
```

The routine begins by obtaining a free file handle and establishing the **On Error Resume Next** condition, which tells DocMan to ignore any disk errors that occur. Then it uses the Visual BASIC **Kill** command to delete the current version of **KeysFile\$**. (The **On Error Resume Next** condition keeps DocMan from balking if **KeysFile\$** doesn't exist when this command is executed.)

```
On Error GoTo Unloaderror
Open KeysFile$ For Output As Keysfilenum
Items = FindDlg.List1.Listcount
Check = 0
Do While Check < (Items)
Out$ = FindDlg.List1.List(Check)
Print # Keysfilenum, Out$
Check = Check + 1
Loop
Close Keysfilenum
Exit Sub
```

Once it has deleted the old copy of **KeysFile\$**, the routine establishes a new **On Error** condition that will jump to the **Unloaderror** label (below) in the event of a file system error. Next it opens **KeysFile\$** for output, and obtains a count of the number of items in the **Keywords** list. It then loops through the list, starting with item 0 and continuing until it reaches the end, assigning each item in turn to the variable **Out\$** and using the Visual BASIC **Print #** command to save the contents of **Out\$** to **KeysFile\$** before looping back and obtaining the next item in the list box.

When it has gone through the entire list, the routine closes the file and exits.

The code following the "**Unloaderror:**" label is executed only if an error occurred while saving the file. Like the error routine in the **Form_Load** procedure, it calls the **FileErrors** function in **GLOBCODE.BAS** and either attempts to resume or exits depending on the value it returns.

```
Unloaderror:
Action = Fileerrors(Err, Keysfilename$)
Select Case Action
Case 0
Resume
```



```
Case Else
Exit Sub
End Select
End Sub
```

Event Procedures

The remaining procedures in FINDDL.G.FRM are called in response to user actions.

The Command1_Click Procedure

The Go and Cancel buttons on the Find Document dialog box form a single control array called Command1, so when you click either button the Command1_Click procedure evaluates its Index argument to determine what action it should take.

```
Sub Command1_Click (Index As Integer)
Select Case Index
Case 0
FindRecord
Case 1
Hide
End Select
End Sub
```

If you click the Go button (index 0), the routine calls the FindRecord subroutine in GLOBCODE.BAS, whereas if you click the Cancel button (index 1), the routine issues the Hide command to hide the FindDlg form, shifting the input focus back to OpenDM.

The List1_Click Procedure

List1 is the Keywords list box. The List1_Click procedure is performed each time you click once on an item in the list box, or move the list box's selection bar to a new item using the cursor keys.

```
Sub List1_Click ()
Word = List1.Listindex
Keyword = List1.List(Word)
End Sub
```

The routine assigns the value of the List1.Listindex property to the integer variable Word (List1.Listindex will be equal to 0 if the first item is selected, or to 10 if the 11th item in the list is selected), and then evaluates the List property for that item. The List property returns the text of a list-box item, so the statement Keyword = List1.List(Word) assigns the text of item Word of list box List1 to the string variable Keyword, which was

declared as a global variable in DCGLOBAL.BAS. The List1_Dblclick, Text1_DragDrop, and Command3_DragDrop functions described below all make use of the Keyword variable this routine establishes.

The List1_Dblclick Procedure

List1_Dblclick is called when you click twice on an item in the Keywords list box. It is designed to copy the selected keyword to the first empty keyword search field on the right side of the form.

```
Sub List1_Dblclick ()
  For X = 0 To 3
    If Text1(X).Text = "" Then Text1(X).Text = Keyword: Exit For
  Next X
End Sub
```

Since every Dblclick event is preceded by a Click event, the value of Keyword has already been set by the List1_Click routine. So the Dblclick routine simply examines the contents of the four keyword search fields (a control array called Text1 with four elements numbered 0 through 3) and places the contents of Keyword into the first field it finds whose Text property is empty. Once it has placed Keyword, it exits the For loop used to examine the fields. If none of the fields are empty, the routine simply exits without making any changes in the contents of the fields.

The List1_MouseMove Procedure

List1_MouseMove is used to initiate drag-drop operations.

```
Sub List1_MouseMove (Button As Integer, Shift As Integer, X As Single, Y As Single)
  If Button <> 2 Then Exit Sub
  List1_Click
  List1.Drag
End Sub
```

Because I only wanted drag operations to take place if the *right* mouse button was depressed, the routine begins by examining the value of its Button argument. If it does not equal 2 (the right button), the routine exits. Otherwise, it calls List1_Click to get the current value of Keyword, and then List1.Drag to initiate the drag procedure.

The Text1_DragDrop Procedure

Text1_DragDrop is called when the user drags a keyword onto any of the four keyword search fields on the right side of the form.

```
Sub Text1_Dragdrop (Index As Integer, Source As Control, X As Single, Y As Single)
  Text1(Index).Text = KeyWord
  Keyword = ""
End Sub
```

The `Text1_DragDrop` routine uses the value of its `Index` argument to place the contents of `Keyword` into the `Text` property of the field the keyword was dropped on.

The `Command2_Click` Procedure

`Command2_Click` is called when the user clicks the `Clear Fields` button.

```
Sub Command2_Click ()
  For X=0 to 3
    Text1(X).Text = ""
  Next X
  OpenDM.Titles.Listindex = 0
  FindNext = 0
  OpenDM.Command2.Enabled = FALSE
End Sub
```

`Command2_Click` begins by clearing the contents of the four keyword fields (by setting the `Text` property of each field to ""). Then it moves the selection bar in `OpenDM`'s `Titles` list box to the first item in the list (so that the next search will start at the beginning of the list), sets `FindNext` to 0 (since you can't find the next occurrence of the search criteria if the search criteria are blank), and disables the `Next Match` button on `OpenDM` (for the same reason).

The `Command3_Click` Procedure

`Command3_Click` is called when the user clicks on the `Delete Key` button on the `FindDlg` form, signaling that the key currently selected in `List1` should be removed from the `Keywords` list.

```
Sub Command3_Click ()
  Word = List1.Listindex
  If Word < 0 Then Exit Sub
  FindDlg.List1.RemoveItem Word
  WriteKeyFields
End Sub
```

The routine begins by obtaining the current `Listindex` number for `List1` (the number of the selected keyword) and, if it is less than 0 (indicating that none of the keywords is selected) it exits immediately. Otherwise, it removes the selected item from the list and then calls the `WriteKeyFields` routine in the `GLOBCODE.BAS` file, which writes the changed keyword list to disk.

The `Command3_DragDrop` Procedure

`Command3_DragDrop` is called when the user drags a keyword onto the `Delete Key` button. It responds by triggering the `Command3_Click` event.


```

Sub Command3_DragDrop (Source As Control, X As Single, Y
    As Single)
Command3_Click
End Sub

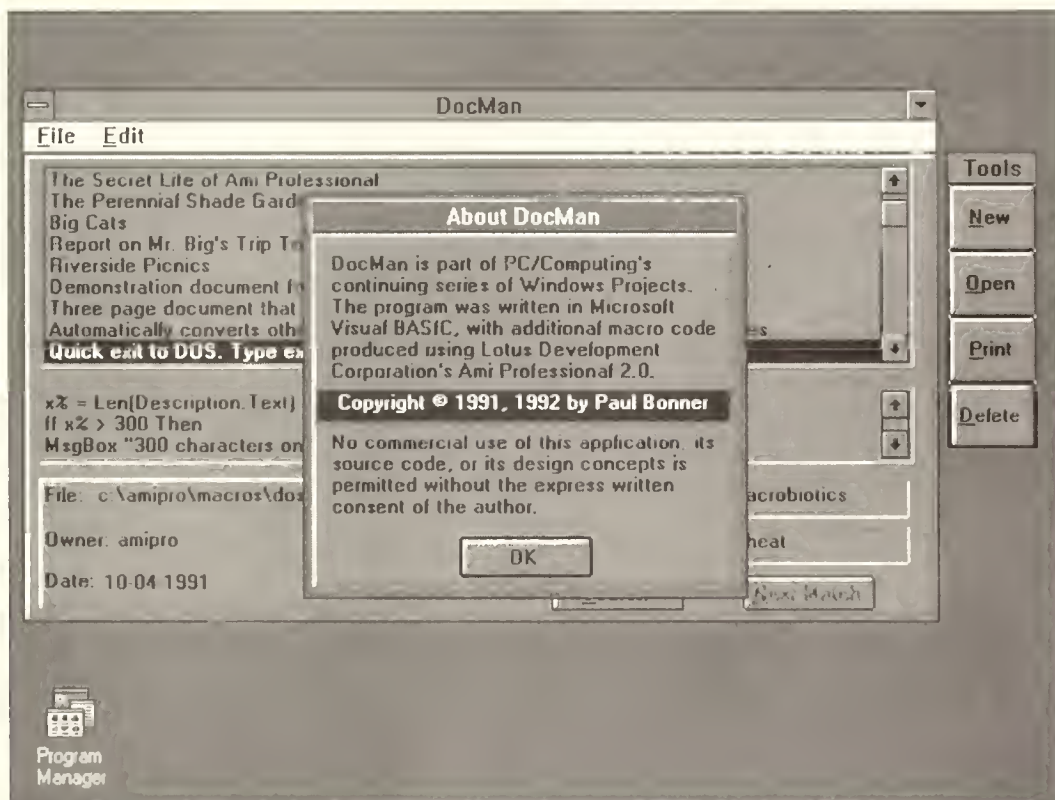
```

That concludes the discussion of the routines in FINDDL.G.FRM.

Exploring ABOUTDLG.FRM

AboutDLG, the last form in the DocMan application, constitutes the About DocMan dialog box. Shown in Figure 14.7, the About DocMan dialog box appears when you select the About item on OpenDM's File menu.

Figure 14.7
The About DocMan
dialog box



The code for AboutDLG is limited to a DefInt A-Z statement and two routines: Form_Paint and Command1_Click.

The Form_Paint Procedure

AboutDLG's Form_Paint procedure simply issues one call to the Frame routine in GLOBCODE.BAS to create a three-dimensional effect around the borders of the dialog box.

```
Sub Form_Paint ()
Call Frame(AboutDLG, 30, 30, AboutDLG.Height - 520, AboutDLG.Width - 220, 2)
End Sub
```

The Command1_Click Procedure

Command1_Click issues the command to unload AboutDLG when the user clicks the OK button.

```
Sub Command1_Click ()
Unload AboutDLG
End Sub
```

Once the About DocMan dialog box has been unloaded, the input focus returns to the OpenDM form.

Exploring GLOBCODE.BAS

GLOBCODE.BAS is a massive code-only module containing no less than 26 subroutines and functions. In addition, its general declarations section is a bit more complex than those of DocMan's forms, since it contains a series of variable declarations, in addition to the ubiquitous DefInt A-Z statement:

```
DefInt A-Z
Dim TestKey(3) As String
Const RunProg$ = "amipro.exe "
Const ClassName$ = "AmiProWndA"
Const HiColor = &HFFFFFF
Const LoColor = &H808080
```

The four-element string array TestKey is used by the FindRecord and TestField routines, which carry out the search requests entered through FindDlg. The constants RunProg\$ and ClassName\$ are used to launch Ami Pro or to detect an existing instance of Ami Pro in memory. Finally, the constants HiColor and LoColor are used to hold the hexadecimal RGB numbers that represent the two colors used by the Frame routine, light gray (&H808080) and white (&HFFFFFF).

In the actual GLOBCODE.BAS file, the remaining, named routines appear in alphabetical order. However, since discussing them in that order would only make the already sketchy flow of an event-driven program even more difficult to follow, related routines have been placed "next door" to one another in the following discussion. (Refer to the index for page numbers of specific routines.)

The CenterForm Procedure

CenterForm is called by the OpenDM and FindDlg forms during their Form_Load routines to center the form on screen.

```
Sub CenterForm (F As Form)
    F.Left = (Screen.Width - F.Width) / 2
    F.Top = (Screen.Height - F.Height) / 2
End Sub
```

The routine uses a variation of the technique that typists used to center text on a page before the invention of word processors. It subtracts the width of the form from the width of the screen, divides the result by two, and then uses that number as the left edge of the form. Then it subtracts the height of the form from the screen height, divides that by two, and uses the resulting number as the top edge of the form.

The Frame Procedure

Frame is called by all but one of DocMan's forms as part of the Form_Paint procedure. Each time it is called it draws four lines that together create the impression of a three-dimensional frame or box.

This version of Frame is one of many Visual BASIC variations on a common routine for creating the 3-D chiseled-steel effect popular among Windows developers. It accepts six parameters from the calling routine: F, the name of the form on which it is to draw lines; L, the left edge of the area to be framed; T, the top edge of the area to be framed; H, the height of the area to be framed; W, the width of the area to be framed; and Style, the style of the frame it will draw.

If the Style is set to 1, the routine will draw a thin frame with white lines along the left and bottom edges of the frame and dark gray lines along the top and right edges, creating a concave effect (that is, making the control around which the frame has been drawn appear to be set below the form's surface). If, on the other hand, Style is set to 2, the routine draws a thick frame that reverses the color order of the thin frame, creating a convex effect. DocMan uses Style 2 only to draw the borders around the edges of a form. Controls are framed using Style 1.

```
Sub Frame (F As Form, L, T, H, W, Style)
    F.DrawWidth = Style
    Select Case Style
    Case 1
        F.ForeColor = HiColor
        SwapColor& = LoColor
    Case 2
        F.ForeColor = LoColor
        SwapColor& = HiColor
    End Select
End Sub
```



```

End Select
'draw bottom, then right
F.Line (L + Style, T + H)-(L + W - Style, T + H)
F.Line (L + W, T + Style)-(L + W, T + H - Style)
'reverse colors
F.ForeColor = SwapColor&
'draw top, then left
F.Line (L - Style, T)-(L + W - Style, T)
F.Line (L, T + Style)-(L, T + H - Style)
End Sub

```

The routine starts by setting the form's DrawWidth property equal to the value of Style, so that the frame will be 1 pixel wide if Style is equal to 1, or 2 pixels wide if Style is equal to 2. Then it assigns values to both the form ForeColor property (the color in which lines will be drawn) and a variable called SwapColor&, using a Select Case routine to make the assignments based on the value of Style. Then it issues a pair of F.Line commands to draw the bottom and right edges of the frame.

The line for the bottom edge starts at a point whose x coordinate is equal to the value of L plus the value of Style (that is, the intended left edge of the frame plus the width of the line), and whose y coordinate is equal to the value of T + H (the top of the frame plus the height of the frame). The bottom line then extends to a point whose x coordinate is equal to the value of L plus the value of W (the width of the frame) minus the value of Style, and whose y coordinate is equal to the value of T + H.

The line along the right edge of the frame follows a similar pattern.

Next, the routine sets the form's ForeColor property equal to SwapColor&, and then draws the lines for the left and top edges of the frame, before returning to the calling routine.

The OpenDoc Procedure

OpenDoc is called when the user selects the Open item from OpenDM's File menu or the Open button from the tools palette.

```

Sub Opendoc ()
Screen.Mousepointer = 11
Getfile
If Editfile = "" Then Exit Sub
AppLoaded = Loaded(Classname$)
If AppLoaded = 0 Then
LaunchApp RunProg$, Editfile, 1
Else
T = SetActiveWindow(AppLoaded)
RestoreApp AppLoaded
SendKeys "%F0" + Editfile + "~", TRUE

```

```
End If
Screen.Mousepointer = 1
End Sub
```

OpenDoc begins by setting the cursor to an hourglass shape (Screen.Mousepointer = 11) to indicate that DocMan is busy. Then it calls the GetFile routine (discussed below) to obtain the name of the file whose title is currently highlighted in the Titles list box. GetFile places the file name in the global variable EditFile.

Next, the routine checks to see if EditFile is empty (which it would be if no file was selected when GetFile was called). If so, it exits. Otherwise, it calls the Loaded routine, passing it the ClassName\$ variable, which holds Ami Pro's class name, to determine if Ami Pro is already running. If it is, then Loaded will return the handle to Ami Pro's main window. Otherwise, OpenDoc calls the LaunchApp routine to launch Ami Pro, passing it the variable EditFile to instruct it to use the name of the file as a command-line parameter and telling it to launch the program in run mode 1 (normal window with input focus). The other run-mode options are 2 (minimized with focus), 3 (maximized with focus), 4 (normal without focus), and 7 (minimized without focus).

If Ami Pro is already running, however, the routine jumps to the Else part of the If-Then construct and calls the Windows API function SetActiveWindow to make Ami Pro the active application. Then the routine calls the RestoreApp routine, in case Ami Pro is currently minimized, and then sends it the keypress Alt-FO followed by the contents of EditFile, thereby instructing it to open the document.

Finally, OpenDoc restores the mouse pointer to its normal state and exits.

The GetFile Procedure

GetFile is called by OpenDoc and other routines to obtain the file name of the document whose title is selected in the Titles list box.

```
Sub Getfile ()
Editfile = ""
Editfile = Recordvar.File
Editfile = RTrim$(Editfile)
End Sub
```

GetFile begins by clearing the variable Editfile. Then it sets Editfile equal to the File field in the current record variable. Next it uses the RTrim\$ function to strip off any extra spaces at the end of the Editfile variable, and then it exits, returning the current value of Editfile to the routine that called it.

The Loaded Procedure

Loaded is used to determine if Ami Pro is already running, so that DocMan can activate the current instance rather than launching a new one.

```
Function Loaded (ClassName$)
  Loaded = FindWindow(ClassName$, 0&)
End Function
```

Loaded passes Ami Pro's class name to the FindWindow API function. If the function returns a value of 0, then Ami Pro is not currently running. Otherwise, it returns the handle of the running instance.

The RestoreApp Procedure

RestoreApp is used to instruct the running instance of Ami Pro to restore its window if it is minimized.

```
Sub RestoreApp (AppLoaded)
  If IsIconic(AppLoaded) Then
    T = PostMessage(AppLoaded, WM_SYSCOMMAND, SC_RESTORE, 0)
    WaitSecs 1
  End If
End Sub
```

The AppLoaded parameter to the RestoreApp routine identifies the running instance of Ami Pro identified by the Loaded function. RestoreApp first calls the IsIconic function defined in DCGLOBAL.BAS to determine if Ami Pro is minimized. Then, if that function evaluates to True, RestoreApp uses the PostMessage function to send it the SC_RESTORE command, instructing Ami Pro to restore itself to its normal window state. Finally, RestoreApp calls the WaitSecs routine (described below), which pauses DocMan for one second to allow Ami Pro a chance to restore its window before any more commands are sent to it.

The LaunchApp Procedure

If Ami Pro isn't running when the Open, Print, or New button is clicked (or one of the corresponding menu items is selected), DocMan must launch AMIPRO.EXE. It does so with the LaunchApp routine.

```
Sub LaunchApp (RunProg$, RunParam$, RunMode%)
  T = Shell(RunProg$ + RunParam$, RunMode%)
  WaitSecs 1
End Sub
```

LaunchApp uses the Visual BASIC Shell command to execute RunProg\$ (which was previously set to equal "AMIPRO.EXE"), and, in the case of open and print routines, passes it the name of the document to open in the

RunParam\$ string. The RunMode% parameter is used to determine Ami Pro's initial window state, as described earlier. Again, the routine calls WaitSecs to give Ami Pro a chance to carry out the command before the next command is sent to it.

The WaitSecs Procedure

WaitSecs is used to pause execution of DocMan for a second or two, while Ami Pro reacts to commands that have been sent to it.

```
Sub WaitSecs (secs)
  Start! = Timer
  While Timer < Start! + Secs
    Temp = DoEvents()
  Wend
End Sub
```

The routine sets the variable Start! equal to the current value of Visual BASIC's Timer function (which always indicates the number of seconds past midnight), and then enters a While loop that continues until Timer equals the value of Start! plus the Secs parameter with which WaitSecs was called. The loop consists of a single statement, Temp = DoEvents(), which is executed repeatedly until the While condition is met.

The Recycler project described in Chapter 12 outlined one use of the DoEvents function in Visual BASIC: to monitor system messages such as drag-drop messages from File Manager. Another, more common use is to yield control to Windows so that any other applications waiting for processor time have a chance to execute. Visual BASIC applications yield control like this automatically whenever they're waiting for user input, but when they are performing a long list of instructions they can be processor hogs unless you remember to call the DoEvents function.

In the case of DocMan, the DoEvents calls in the WaitSecs routine allow Ami Pro to take control of the processor while it carries out the instructions that DocMan has given it. Without this statement, DocMan could keep sending Ami Pro more instructions before it was able to carry out the first ones.

The NewFile Procedure

NewFile is called when the user elects to create a new document, by either clicking the New File button or selecting the File New menu item.

```
Sub NewFile ()
  Screen.Mousepointer = 11
  AppLoaded = Loaded(Classname$)
  If AppLoaded = 0 Then
    LaunchApp RunProg$, "", 4
  Else
```

```

T = SetActiveWindow(AppLoaded)
RestoreApp AppLoaded
End If
SendKeys "%FN", TRUE
Screen.Mousepointer=1
End Sub

```

NewFile sets the cursor to an hourglass, then checks to see if Ami Pro is already running. If not (`AppLoaded = 0`), it calls `LaunchApp` to launch it. Otherwise, it activates Ami Pro's window and calls `RestoreApp` to restore it, if necessary. It then concludes by sending the Alt-FN keystrokes necessary to create a new file in Ami Pro.

The PrintFile Procedure

`PrintFile` is called when the user indicates that the file selected in the Titles list box on `OpenDM` is to be printed.

```

Sub PrintFile ()
OpenDoc
SendKeys "%FP", TRUE
WaitSecs 1
SendKeys "~", TRUE
End Sub

```

`PrintFile` calls the `OpenDoc` routine to load the document that is to be printed into Ami Pro, and then sends Ami Pro the Alt-FP command to print. It then calls `WaitSecs` to pause for one second before sending the carriage return needed to close the Ami Pro Print dialog box and print the document.

The DeleteFile Procedure

`DeleteFile` is called when the user clicks the Delete button or selects the Delete item on the File menu.

```

Sub DeleteFile ()
Getfile
If Editfile = "" Then Exit Sub
Boxtype = MB_OkCancel + MB_ICONEXCLAMATION
Msg$ = "Delete " + Editfile + "?"
Response = MsgBox(Msg$, Boxtype, "Delete File")
Select Case Response
Case IDOK
On Error GoTo Notfound
Kill Editfile
On Error GoTo Ø
DeleteRecord

```

```

Case IDCANCEL
Exit Sub
End Select
Exit Sub

```

```

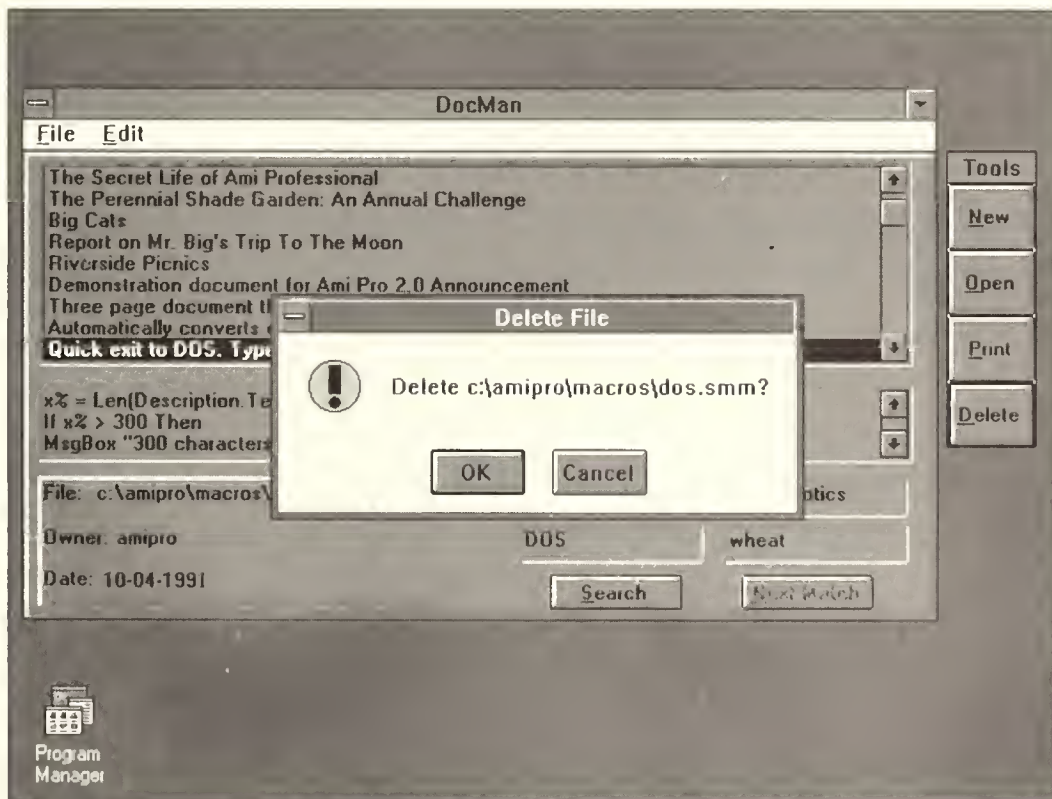
Notfound:
Resume Next
End Sub

```

DeleteFile starts by using GetFile to get the selected document's file name. Then it creates the message box shown in Figure 14.8, to double check that the user really wants to delete the selected document.

Figure 14.8

The Delete File confirmation box



The routine uses a Select Case routine to compare the user's response to the message-box response constants defined in DCGLOBAL.BAS.

If the user indicates that the file is to be deleted, the routine sets up an On Error condition, and then issues the Visual BASIC Kill command to delete the file. Then it cancels the On Error condition with the command On Error GoTo 0, and calls the DeleteRecord routine to delete the current document from DocMan's database, before exiting.

Otherwise, if the user elected not to delete the file, the routine exits immediately.

The code following the NotFound code is executed only if an error occurs during the Kill command. It simply instructs the routine to continue execution at the next line of code.

The DeleteRecord Procedure

DeleteFile calls the DeleteRecord routine to remove the current record from DocMan's database.

```
Sub DeleteRecord ()
Dim Tempvar As DocRec
Position = OpenDM.Titles.Listindex + 1
For I = Position To Lastrecord - 1
Get Filenum, I + 1, Tempvar
Tempvar.Recordnum = I
Put Filenum, I, Tempvar
Next I
Lastrecord = Lastrecord - 1
If Lastrecord = 0 Then Clearfields
OpenDM.Titles.Removeitem Position - 1
OpenDM.Titles.Listindex = Lastrecord - 1
OpenDM.Titles.Refresh
End Sub
```

DeleteRecord starts off by creating a temporary record variable called Tempvar, and then sets the integer variable Position equal to the Titles list box's Listindex property plus 1. Then it steps one record at a time through the database from that position until the next to last record, using the variable I as a counter. At each step, it reads record I + 1 into TempVar, then resets the RecordNum field in Tempvar (which will equal I + 1 at this point) to equal I, and then writes TempVar back to the DOCMAN.DAT file at position I. It then increments I and repeats the process.

To understand what's taking place here, let's suppose there are a total of ten records in the database, and that the seventh record was selected when you initiated the delete operation. The first time through the loop, DeleteRecord would overwrite the old contents of record 7 with the contents of record 8. Then the next time through, the old contents of record 8 would be overwritten with the contents of record 9, and the next and last time through the loop, the old contents of 9 would be overwritten with the contents of record 10.

Once the loop has concluded, the routine decrements the value of Lastrecord, the variable used to keep track of the number of records in the database. At this point, the record the routine was called to delete has been permanently eliminated.

All that remains now is to do a little cleaning up of the OpenDM screen. If the database is empty (Lastrecord = 0), DeleteRecord calls the ClearFields routine (below) to clear all the text fields on the OpenDM screen. Next it removes the deleted record's title from the Titles list box, and resets the list box's Listindex property to be equal to Lastrecord - 1. Finally, it issues the Refresh command, instructing DocMan to redraw the list box to display its new, abbreviated-by-one, list of titles.

The ClearFields Procedure

DeleteRecord uses the ClearFields subroutine to clear the text from the OpenDM text fields if the database is empty.

```
Sub Clearfields ()
OpenDM.Description.Text = ""
For X=0 To 7
If X=3 Then X=4
OpenDM.Text1(X).Text = ""
Next X
End Sub
```

The routine simply sets the Text properties of OpenDM's Description field and the seven text fields in the Text1 control array to empty strings, skipping Text1(3) because no such control exists.

The FileOpener Procedure

The OpenDM Form_Load routine and the Cleanup routine discussed later in this chapter both use the FileOpener function to open the DOCMAN.DAT file.

```
Function FileOpener (NameToUse$, Mode, RecordLen) As
Integer
OpenFileNum = FreeFile
On Error GoTo Openererror
Open NameToUse$ For Random As OpenFileNum Len = RecordLen
LastRecord = LOF(OpenFileNum) \ RecordLen
Fileopener = OpenFileNum
Exit Function
Openererror:
Action = FileErrors(Err, NameToUse$)
Select Case Action
Case 0
Resume
Case Else
Fileopener = 0
End
```

```
End Select
End Function
```

FileOpener starts by obtaining a free file handle, and then, after setting up an On Error routine, opens the file name specified by NameToUse\$ for random access using the handle returned by FreeFile. Then it determines the number of records in the file by dividing the total length of the file by the length of a single record, and returns the file's handle to the routine that called it.

The code following the Openerror label is executed only in the event of an error during the process of opening or reading the file. It calls the FileErrors function, passing it the name of the file and the error code, and then evaluating the result of that function to determine whether it should resume or end.

The FileErrors Procedure

FileErrors evaluates the runtime error code to determine the type of error that has occurred, and then opens a message box describing the error.

```
Function FileErrors (Errval As Integer, Filename As String) As Integer
Msgtype = MB_ICONEXCLAMATION + MB_ABORTRETRYIGNORE
Select Case Errval
Case 68
Msg$ = "Device unavailable "
Case 71
Msg$ = "Disk not ready"
Case 57
Msg$ = "Internal disk error."
Case 61
Msg$ = "Disk full."
Case 52
Msg$ = Filename + " is an illegal filename."
Case 75, 76
Msg$ = "The path " + Filename + " doesn't exist."
Case 54
Msg$ = "Can't open " + Filename + " for that kind of access."
Case 55
Msg$ = Filename + " already open."
Case 62
Msg$ = Filename + " has a nonstandard end of file marker."
Msg$ = Msg$ + " or an attempt was made to read beyond"
Msg$ = Msg$ + " the end of the file."
Case 53
Msg$ = Filename + " not found."
Case Else
Msg$ = "File or disk error associated with " + Filename + "! Error code: " +
    Str$(Errval)
End Select
Response = MsgBox(Msg$, Msgtype, "Disk Error")
Select Case Response
```



```

Case IDRETRY
FileErrors = 0
Case IDIGNORE
FileErrors = 1
Case IDABORT
FileErrors = 2
End Select
End Function

```

The message boxes created by FileErrors have three buttons: Abort, Retry, and Ignore, as shown in Figure 14.9.

Figure 14.9

A FileErrors message box



The function determines which button the user selects in the message box, and then returns a value to the routine that called it based on that response.

The ReadSelectedRecord Procedure

ReadSelectedRecord is called whenever the selection bar moves to a new record in the Titles list box on OpenDM.

```

Sub ReadSelectedRecord ()
If OpenDM.Titles.listindex < 0 Then Exit Sub
If (OpenDM.Titles.listindex + 1) <= LastRecord And LastRecord > 0 Then
Get Filenum, OpenDM.Titles.listindex + 1, Recordvar
OpenDM.Description.Text = RTrim$(Recordvar.Description)
OpenDM.Text1(0).Text = RTrim$(Recordvar.File)
OpenDM.Text1(1).Text = RTrim$(Recordvar.Owner)
OpenDM.Text1(2).Text = RTrim$(Recordvar.Date)
OpenDM.Text1(4).Text = RTrim$(Recordvar.Key1)
OpenDM.Text1(5).Text = RTrim$(Recordvar.Key2)
OpenDM.Text1(6).Text = RTrim$(Recordvar.Key3)
OpenDM.Text1(7).Text = RTrim$(Recordvar.Key4)
WhichRecChanged = -1
End If
End Sub

```

The routine starts by conducting a couple of checks to ensure that Titles.listindex is pointing at a valid record. Once it has made that determination, it reads record number Titles.listindex + 1 from Filenum (the handle of the DOCMAN.DAT file), and places the contents of that record into the Recordvar variable. Then it copies the contents of the various fields in

Recordvar into the corresponding fields on OpenDM. Next it sets the WhichRecChanged flag to -1 (indicating that the contents of the current record have not changed since it was read from disk) and exits.

The WriteChangedRecord Procedure

WriteChangedRecord is called to save the contents of the current record when WhichRecChanged is greater than -1, indicating that the contents of the record have been altered since it was read from disk.

The WriteChangedRecord routine gets called any time an OpenDM edit field whose contents have been changed loses the input focus. Thus, when you change the description of a file or one of its keywords, as soon as you tab to or click in another field, the changed record is saved to disk. However, the routine also gets called every time data being sent from Ami Pro is inserted into a field during the OpenDM Link_Execute routine, so Link_Execute sets the NewRecordFlag to True to prevent WriteChangedRecord from writing the new record to disk prematurely before all the data being received from Ami Pro has been entered into the appropriate fields on OpenDM.

```
Sub WriteChangedRecord ( )
  If NewRecordFlag = TRUE Then Exit Sub
  Recordvar.Title = OpenDM.Titles.list(WhichRecChanged)
  Recordvar.Description = OpenDM.Description.Text
  Recordvar.key1 = OpenDM.Text1(4).Text
  Recordvar.key2 = OpenDM.Text1(5).Text
  Recordvar.key3 = OpenDM.Text1(6).Text
  Recordvar.key4 = OpenDM.Text1(7).Text
  Put Filenum, WhichRecChanged + 1, Recordvar
  WhichRecChanged = -1
  Addkeys
End Sub
```

If NewRecordFlag is True, WriteChangedRecord exits immediately. Otherwise, it sets the various fields in the Recordvar variable equal to the contents of the corresponding fields on the OpenDM form. Then it issues the Visual BASIC Put command to write the changed record to disk, and then sets WhichRecChanged to -1 to indicate that the current record does not differ from the version stored on disk. Finally, it calls the AddKeys routine, which adds any new keywords that appear on the OpenDM form to the Keywords list.

The AddKeys Procedure

The purpose of AddKeys is to add new, unique keywords to the Keywords list box on the FindDlg form without creating duplicates of keywords that already appear in the list.

There were (at least) two possible ways to implement this. One was to have AddKeys search through the Keywords list four times (once for each keyword on OpenDM) to determine if the OpenDM keyword already appears in the list and, if not, to add it. The other was to simply add all four keywords on any changed record to the list, and then run through the list once, removing duplicate items. I opted for the second approach, so AddKeys begins by adding the contents of all four keyword fields on the OpenDM form to the Keywords list box on FindDlg.

```

Sub AddKeys ( )
For X = 4 To 7
FindDlg.List1.AddItem OpenDM.Text1(X).Text
Next X
Items = FindDlg.List1.Listcount
NewKeys = 0
Check = 0
FindDlg.List1.Refresh
Do While Check < (Items)
If UCase$(RTrim$(FindDlg.List1.list(Check))) <>
    UCase$(RTrim$(FindDlg.List1.list(Check + 1)))
    And RTrim$(FindDlg.List1.list(Check)) <> "" Then
Check = Check + 1
NewKeys = 1
Else
FindDlg.List1.RemoveItem Check
Items = Items - 1
End If
Loop
If NewKeys = 0 Then Exit Sub
WriteKeyFields
End Sub

```

Next AddKeys sets the integer variable Items equal to the number of items in the list box, and the variables NewKeys and Check equal to 0. Then it issues the FindDlg.List1.Refresh command, which tells DocMan to redraw the list box. Since the list box's Sorted property is set to True, this also causes the contents of the list box to be sorted in ascending alphabetical order.

Once the list box has been sorted, all the routine has to do to catch duplicate items is to make sure that no two consecutive items are identical. So it loops through the items in the list box, comparing each item to the one that follows it.

Because DocMan's searches are case-insensitive, "report", "Report", and "REPORT" are of identical effectiveness in finding a document that uses any of the three as a keyword. So the AddKeys routine uses the Visual BASIC UCase\$ function to compare uppercase versions of all keyword list entries (and the RTrim\$ function to ensure that a different number of trailing spaces doesn't prevent the routine from flagging duplicates).

If the list item being checked is not identical to the one that follows it, and isn't an empty string, AddKeys increments the value of the counter variable Check and sets NewKeys equal to 1 (indicating that the keyword list has changed and should be saved to disk). If, on the other hand, the item is identical to the one that follows it, AddKeys removes the current item from the list and decreases the variable Items (which reflects the total number of keywords in the list) by 1 before reading the next keyword.

Once the entire list box has been scanned for duplicates, AddKeys checks to see whether NewKeys is still equal to 0. If it is, then the keyword list hasn't changed and AddKeys exits. Otherwise, it calls the WriteKeyFields routine to save the list to disk.

The WriteKeyFields Procedure

WriteKeyFields reverses the process of reading keywords from disk that occurs during the FindDlg Form_Load procedure. It saves every item in the Keywords list box to the file DMKEYS.DAT, using the Print # command to write the sequential file and a error-handling routine called Unloaderror to handle any file errors that occur.

```

Sub WriteKeyFields ()
  KEYSFILENUM = FreeFile
  On Error GoTo Unloaderror
  KeysFile$ = ExePath + "DMKEYS.DAT"
  On Error Resume Next
  Kill KeysFile$
  On Error GoTo Unloaderror
  Open KeysFile$ For Output As KEYSFILENUM
  Items = FindDlg.List1.Listcount
  Check = 0
  Do While Check < (Items)
    Out$ = FindDlg.List1.List(Check)
    Print #KEYSFILENUM, Out$
    Check = Check + 1
  Loop
  Close KEYSFILENUM
  Exit Sub
Unloaderror:
  Action = FileErrors(Err, NewName$)
  Select Case Action
  Case 0
  Resume
  Case Else
  Exit Sub

```

```
End Select
End Sub
```

The ExitDocMan Procedure

ExitDocMan is called when the user selects the Exit item on OpenDM's File menu.

```
Sub ExitDocMan ()
  Cleanup
End
End Sub
```

ExitDocMan calls the Cleanup routine (below), and then issues the End command to stop program execution.

The Cleanup Procedure

Cleanup's job is to ensure that all data files have been saved to disk properly before DocMan exits.

It begins by calling WriteChangedRecord if WhichRecChanged is greater than -1.

```
Sub Cleanup ()
  If WhichRecChanged > -1 Then WriteChangedRecord
  NewName$ = ExePath + "DM.TMP"
  FMode = RANDOMFILE
  Recordcount = LastRecord
  On Error GoTo Cleanuperrs
  Cleanupfilenum = Fileopener(NewName$, fMode,
    Len(Recordvar))
  On Error Resume Next
  For I = 1 To Recordcount
    Get Filenum, I, Recordvar
    Put Cleanupfilenum, I, Recordvar
  Next I
  Close
  Kill ExePath + "DOCMAN.DAT"
  Name NewName$ As ExePath + "DOCMAN.DAT"
  Exit Sub
Cleanuperrs:
  Action = FileErrors(Err, NewName$)
  Select Case Action
  Case 0
  Resume
  Case Else
  End
```

```
Exit Sub
End Select
End Sub
```

Next, `CleanUp` opens a new file called `DM.TMP` and copies every record from 1 to `RecordCount` from `DOCMAN.DAT` to the new file. When that's done, it deletes `DOCMAN.DAT` and renames `DM.TMP` to `DOCMAN.DAT`, replacing the old file with the new one.

The purpose of replacing `DOCMAN.DAT` is to ensure that deleted records are permanently deleted. You'll recall that the initial value of `RecordCount` is determined by dividing the length of `DOCMAN.DAT` by the length of a single record. When you delete records from the database, the value of `RecordCount` goes down, but the data from the deleted records (or data from other records past the deleted records) still exists in the `DOCMAN.DAT` file. If `CleanUp` simply closed `DOCMAN.DAT`, the next time it was opened `RecordCount` would reflect the total length of the file, resulting in either duplicate records or previously deleted records appearing at the end of the file.

By copying only the number of records equal to `RecordCount` to the `DM.TMP` file, the routine ensures that the new file doesn't contain any data for deleted or duplicate records.

The code following the `Cleanuperrs` label is called only if a file error occurs during the cleanup process. If so, it calls `FileErrors`, and then either resumes or exits depending on the user's response to the `FileErrors` message box.

The FindRecord Procedure

`FindRecord` is called when you initiate a search by pressing either the `Go` button on the `FindDlg` form or the `Next Match` button on `OpenDM`.

```
Sub FindRecord ()
For Temp = 0 To 3
TestKey(Temp) = UCase$(RTrim$(FindDlg.Text1(Temp).Text))
Next Temp
For Match = OpenDM.Titles.listindex + 1 + FindNext To LastRecord
Get Filenum, Match, Recordvar
If TestField(UCase$(RTrim$(Recordvar.key1))) Then Exit For
If TestField(UCase$(RTrim$(Recordvar.key2))) Then Exit For
If TestField(UCase$(RTrim$(Recordvar.key3))) Then Exit For
If TestField(UCase$(RTrim$(Recordvar.key4))) Then Exit For
Next Match
If Match < LastRecord + 1 Then
FindDlg.Hide
OpenDM.Titles.listindex = Match - 1
OpenDM.Command2.Enabled = TRUE
Else MsgBox "Keywords not found", MB_OK, "DocMan"
OpenDM.Command2.Enabled = FALSE
```



```
End If
End Sub
```

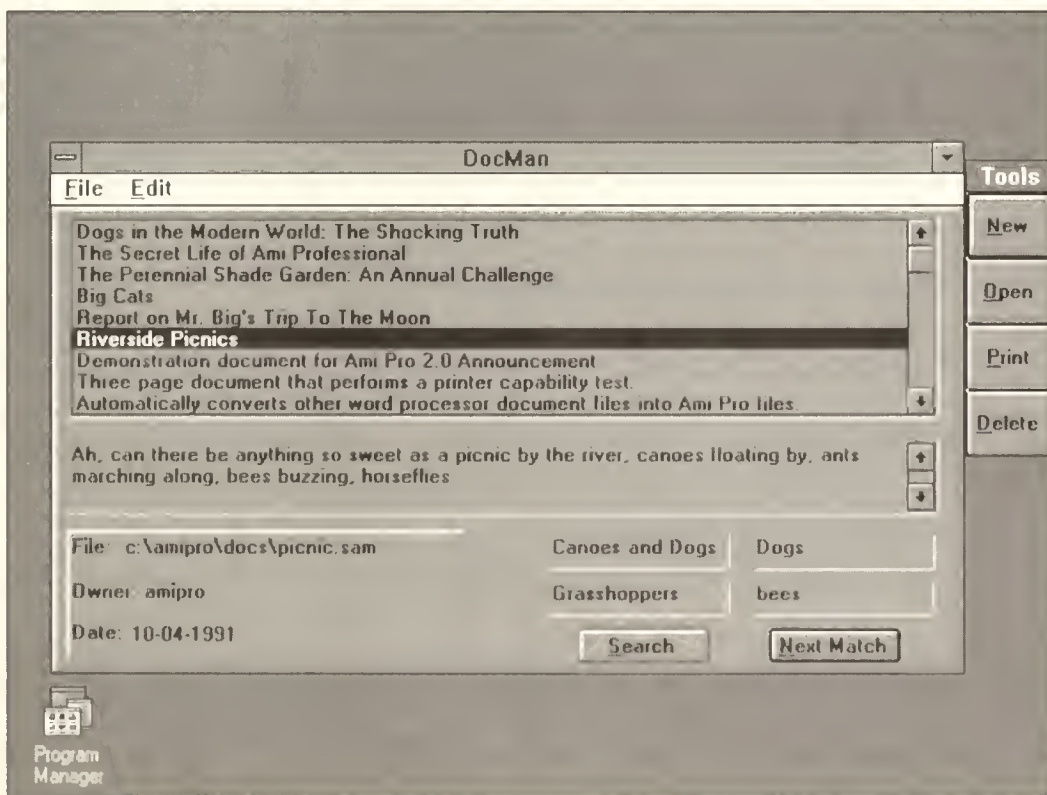
FindRecord begins by setting the variables Temp(0) through Temp(3) equal to the uppercase, trailing-spaces-trimmed contents of the four FindDlg keyword search fields. Then, starting with the document currently selected in the Titles list box on OpenDM (which will be the first record in the list box if FindNext = 0), the routine reads each record in turn until the TestField function (described below) returns a value of TRUE for one of the record's four keyword fields.

If no match is found, the value of Match will be equal to Lastrecord + 1 at the end of the For loop. But if a match was found, the Exit For command following the TestField statement that evaluated to TRUE would have made FindRecord exit the loop before the final Next statement incremented Match to that value. Consequently, the routine uses the value of Match at the end of the For Next loop to determine its next action.

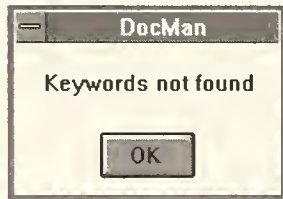
If a match was found (Match < LastRecord + 1), the routine hides the FindDlg form, sets the Titles list box Listindex property to Match - 1, highlighting the matching record (since Listindex is numbered from 0, whereas the record numbers start from 1), and enables the Next Match button, as shown in Figure 14.10.

Figure 14.10

After a successful search the Next Match button is enabled



Meanwhile, if no match was found, FindRecord informs the user through the message box shown below, and then disables the Next Match button before exiting the routine.



The TestField Procedure

TestField is used by FindRecord to determine if any of the four key fields in the current record variable match the search criteria. FindRecord calls TestField four times for each record that it tests, passing it one keyword field at a time.

```
Function TestField (FieldToTest As String)
If FieldToTest = "" Then Exit Function
For Temp = 0 To 3
If FieldToTest = TestKey(Temp) Then TestField = TRUE: Exit For
Next Temp
End Function
```

If the keyword field that FindRecord has passed to TestField is empty, the function exits immediately. Otherwise, it compares the value of the field being tested with each of the four TestKey(Temp) strings in turn. (You'll recall that the TestKey(Temp) strings hold the criteria for the current search.) If it obtains a match it returns a value of TRUE to the FindRecord function. Otherwise it exits, returning a value of FALSE.

The Sub GetPath Procedure

The OpenDM Form_Load procedure calls GetPath to determine the directory from which DOCMAN.EXE was run, so that it knows where to look for and store DocMan's data files.

```
Sub GetPath ()
Const GCW_HMODULE = (-16)
ExePath = String$(127, 0)
X = GetModuleFilename(GetClassWord(OpenDM.Hwnd, GCW_HMODULE), ExePath,
    Len(ExePath))
X = Len(ExePath)
Do While X > 0
If Mid$(ExePath, X, 1) = "\" Then Exit Do
X = X - 1
Loop
If X = 0 Then
ExePath = "\"
```

```
Else ExePath = Left$(ExePath, X)
End If
End Sub
```

GetPath combines a pair of Windows API functions here, passing GetModuleFilename the result of calling the GetClassWord function with the parameters OpenDM.Hwnd (the handle of the OpenDM window) and GCW_HMODULE (a request for a handle to the module). It also passes the fixed-length string ExePath to GetModuleFilename, along with an integer indicating the length of that string.

The result is that Windows places the fully qualified name of the running instance of DOCMAN.EXE in the string ExePath—so that if, for instance, DOCMAN.EXE is stored in the WINDOWS directory, ExePath will contain the string C:\WINDOWS\DOCMAN.EXE.

Next, GetPath starts at the last character in ExePath and works its way back toward the first, looking for a backslash (\). Once it finds one, it uses the Left\$ function to strip off the characters that follow the backslash, so that C:\WINDOWS\DOCMAN.EXE is cut down to C:\WINDOWS, yielding the path of the directory in which the DOCMAN.EXE application can be found.

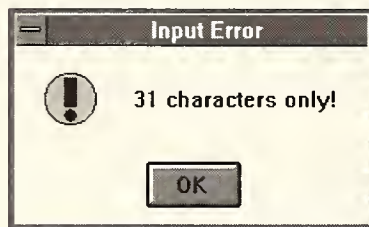
The TestLength Procedure

TestLength is used by the Keyword and Description edit fields on OpenDM to ensure that the user hasn't entered more text into the field than DocMan allows (31 characters in a Keyword field, or 300 characters in the Description field). The Keyword and Description fields call this routine every time the user types a character there, so it needed to be efficient and quick in order to keep up with a fast typist.

```
Sub TestLength (C As Control, L As Integer)
  Select Case Len(C.Text)
  Case Is <= L
    Exit Sub
  Case Else
    MsgBox Str$(L) + " characters only! ", MB_ICONEXCLAMATION,
      "Input Error"
    LeftText$ = Left$(C.Text, C.SelStart)
    RightText$ = Mid$(C.Text, C.SelStart + 1)
    LeftText$ = Left$(LeftText$, L - Len(RightText$))
    C.Text = LeftText$ + RightText$
  End Select
End Sub
```


The routine accepts two parameters: C, which indicates the control that called the routine, and L, the maximum number of characters the control will accept.

The routine consists entirely of a Select Case statement that evaluates the length of the text in control C. If the result of Len(C.Text) is less than or equal to L, the routine exits immediately. Otherwise, it creates the dialog box shown below, informing the user that he or she has exceeded the maximum number of characters allowed for the current control.



Next, the routine trims the control's text until it is equal to the maximum length specified by L. The easy way to do this would be to set C.Text = Left\$(C.Text,L), but that would simply strip extra characters from the end of the text, which might not be what you want if you went over the limit by inserting characters into the middle of the text. So instead, the routine does its trimming backwards from the current insertion point, removing enough characters from the text to the left of the insertion point to get the total length of the text under the limit.

First the routine assigns all of the control's text up to the current insertion point to the string LeftText\$, using the Left\$ function, and all the text to the right of the insertion point to the string RightText\$, using the Mid\$ function. Then it modifies LeftText\$ by determining the value of L minus the length of RightText\$ and then trimming LeftText\$ to that number of characters. So if L is 31 and RightText\$ contains 10 characters, LeftText\$ will be trimmed to 21 characters. Finally, the current text of the control is replaced by the combined contents of LeftText\$ and RightText\$.

The strength of this routine is that it removes only as many characters as necessary to get the control's text under the limit imposed by L. So if you go over the control's text-length limit by typing a single character into the control, the routine will remove just one character. But if you paste a string of characters into the control, the routine will remove as many characters as necessary to get the text length under the limit.

That concludes discussion of the routines in DOCMAN.EXE. Now, let's move to the four Ami Professional macros, which supply the data that gives DocMan something to do.

Inside Ami Pro

DocMan does not function in a vacuum. You can run it independent of any other application, but there is no point in doing so because DocMan doesn't have the power to create new records for its database by itself. To be useful, it requires the active intervention of another application—in this case Ami Pro.

Ami Pro Macros

I wrote four macros to provide support for DocMan in Ami Pro. The first, AutoExec, modifies Ami Pro's standard menus to allow them to interact with DocMan. The second, AutoNew, is used to obtain a document title, description, and four keywords from the user every time a new document is created. The third macro, Savemac, sends that information to DocMan the first time a document is saved and every time thereafter that it is saved under a new name. The fourth macro, DMInfo, is used to activate DocMan from within Ami Pro.

The AutoExec Macro

AutoExec adds a new menu item, called DocMan, to the AmiPro File menu and modifies the actions of the Save and Save As... items on the File menu so that they call the Savemac macro rather than the standard Ami Pro save-file routine. The modified menu is shown in Figure 14.11.

The AutoExec macro is executed automatically every time Ami Pro is run.

```
FUNCTION AutoExec()
IGNOREKEYBOARD(1)
INSERTMENUITEM (1. "&File",11, "Doc&man...", "DMINFO.SMM", "Call DocMan")
CHANGEMENUACTION(1. "&File". "&Save^S", "SAVEMAC.SMM", "Save current file")
CHANGEMENUACTION(1. "&File", "Save &As...", "SAVEMAC.SMM!SaveWithNewName()",
  "Save current file under a new name")
IF GETOPENFILENAME$()="" FILECLOSE()
ENDIF
IGNOREKEYBOARD(0)
END FUNCTION
```

The macro begins by telling Ami Pro to ignore keyboard input. Then it uses the INSERTMENUITEM command to add a new menu item labeled "DocMan" as item 11 on Ami Pro's File menu. Ami Pro executes the macro called DMINFO.SMM when the user chooses the DocMan item, and displays the text "Call DocMan" in its title bar when the menu selector bar passes over the DocMan item.

Next the macro uses the CHANGEMENUACTION command twice, first to change the Save menu item's action so that it calls the Savemac macro, and then to change Save As... so that it calls the SaveWithNewName subroutine of the Savemac macro. (Since no subroutine name is specified in

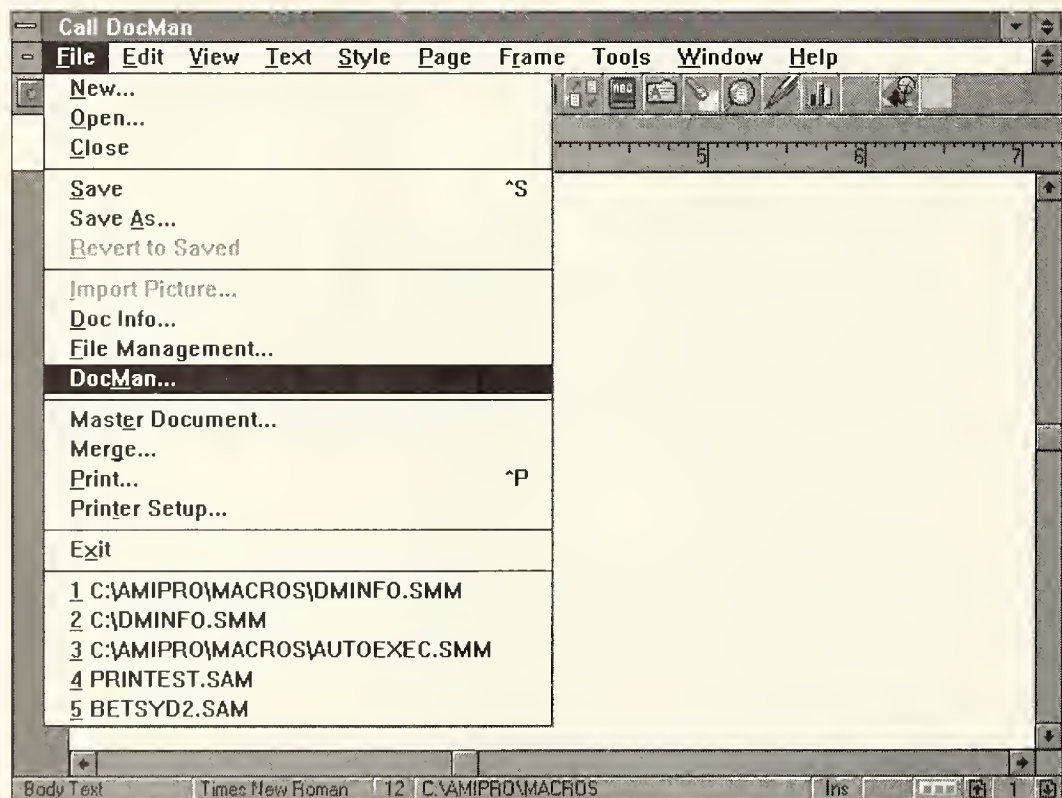
the call to Savemac made by the Save menu item, macro execution will begin at the first line of the macro, rather than at a named subroutine.)

When Ami Pro is loaded, it automatically opens a new file without executing the AutoNew macro. But since AutoNew hasn't run, no DocMan information has been collected for the open file, so the AutoExec macro's next step is to close the file. It does so by checking the name of the open file, using the GETOPENFILENAME function. If the file is unnamed, the macro issues the FILECLOSE command.

Finally, the macro tells Ami Pro to turn off the IGNOREKEYBOARD switch, and then exits.

Figure 14.11

The modified Ami Pro File menu



The AutoNew Macro

AutoNew executes automatically every time you use Ami Pro's File New command to create a new file. (There is no need to tell Ami Pro to look for this macro—it automatically looks for macros called AutoNew, AutoOpen, and AutoClose whenever a file is created, opened, or closed.)

The macro begins by displaying the Document Description dialog box, shown earlier in Figure 14.3. The instructions for drawing this dialog box appear in the macro following the END FUNCTION command. It was created using the dialog box editor in the Ami Professional Macro Developer's

Kit, although the standard Ami Pro dialog box editor would have served as well for this particular job.

```

FUNCTION AutoNew()
Box=DIALOGBOX(".", "DocDes")
IGNOREKEYBOARD(1)
SWITCH BOX
CASE 0
FILECLOSE ()
EXIT FUNCTION
CASE 1
HOURGLASS(1)
Title=GETDIALOGFIELD$(8001)
Descrip=GETDIALOGFIELD$(8002)
Key1=GETDIALOGFIELD$(8003)
Key2=GETDIALOGFIELD$(8004)
Key3=GETDIALOGFIELD$(8005)
Key4=GETDIALOGFIELD$(8006)
DOCINFO(TITLE,DESCRIP,0)
DOCINFOFIELDS(Key1,Key2,Key3,Key4,"","","")
HOURGLASS(0)
ENDSWITCH
IGNOREKEYBOARD(0)
END FUNCTION

```

Once the dialog box has been closed, the macro tells Ami Pro to ignore user input. Then it evaluates the result of the dialog box, which was assigned to the variable `Box`. If the user pressed the Cancel button, `Box` will be equal to 0, so the macro closes the newly created document and exits. Otherwise, it sets the mouse pointer to an hourglass to indicate that it is busy, and then uses the `GETDIALOGFIELD$` function to obtain the data the user typed into the Title, Description, and Keyword fields on the dialog box.

The variables to which the macro assigns this data are temporary and therefore cease to exist as soon as the `AutoNew` macro ends. This was a problem, since I didn't want to save the information in DocMan until the document was saved. Fortunately, Ami Pro provided an easy solution: a standard information sheet (Doc Info) for each document, which includes a 120-character Description field, a 300-character Keywords field, and eight 31-character DocInfo fields.

Ami Pro doesn't do much with this information, providing no way to search for documents on the basis of their keywords or description—hence the need for DocMan. But I was able to use the capacities of these fields as a guide in planning DocMan's field lengths, so that the 300-character capacity of the Keywords field, for instance, determined the 300-character limit of DocMan's Description field. I used the `DOCINFO` and `DOCINFO FIELDS`

commands to automatically stuff the data obtained from the Document Description dialog box into fields on the Doc Info sheet for safekeeping.

Finally, the macro restores the standard cursor, tells Ami Pro to start paying attention to the keyboard again, and exits.

The remaining text in the AutoNew macro is the code for creating the Document Description dialog box generated by the dialog box editor. It isn't intended to be comprehensible to anyone except Ami Pro, and I couldn't even begin to explain the significance of most of its code, other than to say that it creates the dialog box shown in Figure 14.3. But here it is anyway, for the morbidly curious:

```
DIALOG DOCDES
-2134900736 11 24 24 264 167 "" "" "Document Description"
FONT 10 "SYSTEM"
3 12 18 10 1001 1342177280 "STATIC" "&Title: " 0
25 12 198 12 8001 1350631552 "EDIT" "" 0
3 54 20 10 1004 1342177280 "STATIC" "&Desc:" 0
25 35 233 71 8002 1350631428 "EDIT" "" 0
26 109 231 44 11 1342177287 "BUTTON" "&Keywords" 0
45 121 77 12 8003 1350631552 "EDIT" "" 0
158 122 76 12 8004 1350631552 "EDIT" "" 0
45 136 76 12 8005 1350631552 "EDIT" "" 0
158 137 77 12 8006 1350631552 "EDIT" "" 0
230 3 29 11 1 1342242817 "BUTTON" "Okay" 0
230 19 29 11 2 1342242816 "BUTTON" "Cancel" 0
END DIALOG
```

The Savemac Macro

Savemac, or its subfunction SaveWithNewName, is called whenever you save a file.

```
FUNCTION Savemac()
IGNOREKEYBOARD(1)
Fname=GETDOCINFO$(DDfilename)
IF MID$(Fname,2,8) = "Untitled" THEN CALL SaveWithNewName()
ELSE SV=SAVE
ENDIF
IGNOREKEYBOARD(0)
END FUNCTION
```

The Savemac macro begins by telling Ami Pro to ignore keyboard input. Then it obtains the name of the file being saved, using the GETDOCINFO\$ command, and checks to see whether the file is untitled. If so, the macro calls the SaveWithNewName function. Otherwise it calls the standard Ami Pro SAVE function to save the file under its current name, then exits.

```

FUNCTION SaveWithNewName()
SV=SAVEAS
IF SV<1 THEN EXIT FUNCTION
ENDIF
Path=GETDOCINFO$(Ddpath)
Appname="amipro"
Fname=GETDOCINFO$(DDfilename)
Fname=STRCAT$(Path, "\", Fname)
Fname=LCASE$(Fname)
Title=GETDOCINFO$(DDDescription)
Key1=GETDOCINFO$(DDUSER1)
Key2=GETDOCINFO$(DDUSER2)
Key3=GETDOCINFO$(DDUSER3)
Key4=GETDOCINFO$(DDUSER4)
Descrip=GETDOCINFOKEYWORDS$()

```

The `SaveWithNewName` function calls the standard Ami Pro `SAVEAS` function, which obtains the name the user wants to use for the file and saves it under that name if the user doesn't select the Cancel button on the standard Save As dialog box. Then `SaveWithNewName` examines the result of the Save function and exits if it is less than 1 (which would indicate that the user canceled the operation). Otherwise, it extracts the data it wants to send to DocMan from the document's Doc Info sheet, using the `GETDOCINFO$` command, and adds the document's path to the document-name variable `Fname`.

```

ChannelID=DDEINITIATE ("DOCMAN" "DOCFORM")
IF (0 = ChannelID)
EXEC ("DOCMAN.EXE", "")
TIME1=NOW() + 3
R=NOW()
WHILE R < TIME1
R = NOW()
WEND
ChannelID=DDEINITIATE ("DOCMAN" "DOCFORM")
IF (0 = ChannelID)
MESSAGE ("CAN'T ESTABLISH LINK TO DOCMAN!")
EXIT FUNCTION
ENDIF
ENDIF

```

Next, the macro attempts to establish a DDE conversation with DocMan using the `DDEINITIATE` command. If the attempt is unsuccessful (`0 = ChannelID`), the macro uses the `EXEC` command to attempt to launch `DOCMAN.EXE` (which must be on the DOS path for this to work). Then it

waits three seconds, using Ami Pro's NOW function and a WHILE loop to create the delay, and then tries again to establish the DDE link. If it can't, it creates a message box saying so and then exits.

```
DDEEXECUTE(ChannelID, "titl={Title}")
DDEEXECUTE(ChannelID, "appl={Appname}")
DDEEXECUTE(ChannelID, "file={Fname}")
DDEEXECUTE(ChannelID, "desc={Descrip}")
DDEEXECUTE(ChannelID, "key1={Key1}")
DDEEXECUTE(ChannelID, "key2={Key2}")
DDEEXECUTE(ChannelID, "key3={Key3}")
DDEEXECUTE(ChannelID, "key4={Key4}")
DDEEXECUTE(ChannelID, "show")
DDETERMINATE (ChannelID)
END FUNCTION
```

If either attempt to create a DDE link is successful, Ami Pro issues a series of DDEEXECUTE commands to send DocMan the information it needs about the document being saved, including its title, description, and keywords. The Link_Execute routine on DocMan's OpenDM form intercepts the information and places it in a new record, as described above.

The SaveWithNewName macro terminates the DDEchannel and exits after sending the command string "show", which OpenDM's Link_Execute routine interprets as an instruction to make DocMan visible.

The DMInfo Macro

The final Ami Pro macro, DMInfo, is called when the user selects the DocMan item that the AutoExec macro added to Ami Pro's File menu.

```
FUNCTION DMInfo()
A=FILECHANGED(0,0)
IF A=1 THEN SAVE()
ENDIF
ChannelID=DDEINITIATE ("DOCMAN" "DOCFORM")
IF (0 = ChannelID)
EXEC ("DOCMAN.EXE", "")
TIME1=NOW() + 1
R=NOW()
WHILE R < TIME1
R = NOW()
WEND
ChannelID=DDEINITIATE ("DOCMAN" "DOCFORM")
IF (0 = ChannelID)
MESSAGE ("Can't Find Docman!")
EXIT FUNCTION
```

```
ENDIF
ENDIF
DDEEXECUTE (ChannelID, "show")
DDETERMINATE (ChannelID)
END FUNCTION
```

DMInfo starts by determining whether the current document has been changed since it was last saved. If so, it saves it automatically. Then it uses the same techniques as SaveWithNewName to establish a DDE conversation with DocMan and, once it has DocMan's attention, sends it the "show" command (to which OpenDM's Link_Execute routine responds by making DocMan visible). Finally, its job done, the DMInfo macro terminates the DDE channel and exits.

Wrapping Up DocMan

That's it. We've made it through every line of code in the DOCMAN.EXE application, and every line of Ami Pro macro code. But, of course, that's not the whole DocMan story.

As I said at the beginning of this chapter, DocMan was designed from the beginning to be extensible for use with applications other than Ami Pro. Although I may never get around to extending it to work with any other applications, you're more than welcome to take a crack at it—as long as you don't attempt to sell the finished product.

Here are the steps you'll have to take to modify DocMan for use with one or more additional applications:

1. Turn the RunProg\$ and ClassName\$ string variables in GLOBCODE.BAS into arrays, and add the executable file name and class name of any application that you want to use with DocMan to the array.
2. Modify the GetFile routine in GLOBCODE.BAS to make it examine the owner field in Recordvar to determine which application created the document and then have it choose the correct executable file name and class name from the arrays created in step 1 for use with that document.
3. Modify the NewFile routine in GLOBCODE.BAS to allow it to find out what kind of document you want to create.
4. Modify the Titles.Mousemove routine in OpenDM to have it load the appropriate application's icon from disk when you drag a document. You could, for instance, have it call GetFile to determine which application created the document, and then use the ExtractIcon function from the Windows 3.1 library SHELL.DLL to obtain that application's icon.

5. (This is the biggie.) Write macros in the new application's macro language that duplicate the functions of the Ami Pro macros in the current version of DocMan. The code will be different and you'll undoubtedly have to employ different methods, as well, but the result you want to achieve is the same: to have the application obtain the necessary information from the user and send it to DocMan via the DDE channel, using the command prefixes used by the SaveWithNewName macro (titl=, file=, show, and so on) to identify the data as it is sent through the DDE channel.

These steps outline but one of the many ways in which DocMan could be extended. Another method might be to use a pop-up dialog box created by DocMan to obtain the document title, description, and keywords. Doing so would reduce the amount of application macro programming that would be required to implement support for DocMan, but would also involve some major changes in DocMan's structure. In any case, no matter what you choose to do with it, DocMan should provide you with some good ideas about ways to extend standard applications and integrate them with your custom programs.

C H A P T E R

15

**Communicating with
Host Systems—
M.M.M.: the MCI
Mail Manager**

*Selecting the
Development Tool*

M.M.M.'s Capabilities

How M.M.M. Works

*Exploring the
AUTOMCI.DCP Script*

Exploring TM.DCP

Exploring EMAIL.DCP

Exploring PM.DCP

*Exploring
ONLINE.DCP*

Wrapping Up M.M.M.

M.M.M.: THE MCI MAIL MANAGER IS A COMPLETE, WINDOWS-BASED front end for the MCI Mail electronic mail service. Developed entirely in DynaComm's script language, it provides an automated solution to the tasks of creating, sending, receiving, and organizing MCI mail messages. Among its more powerful features are dual off-line address directories, support for up to ten different MCI accounts, and full off-line composition and editing of messages.

M.M.M. was developed out of my frustration with the standard MCI Mail user interface. Over the years I had grown dependent on MCI Mail as a convenient means of corresponding with dozens of professional and personal contacts across the country. But I had also grown increasingly intolerant of the standard MCI Mail user interface—including its obscure command structure and its primitive line editor for creating messages. There was an obvious need for a better, preferably Windows-based, interface for the mail service.

In addition, the more dependent I became on MCI Mail as a communications medium, the more I needed a way to organize and store the messages I had received and sent. Increasingly, the mail service became an important element of my ongoing projects, so it was no longer enough to simply type and read messages on line, and to then have them disappear into the ether once the on-line session was complete.

Consequently, as I set out to design and build a Windows interface for MCI Mail, I had a broad set of design goals: to simplify and automate the communications process, to escape the limitations of MCI's on-line editor, and to simplify the process of organizing the messages I sent and received.

Improving on an Existing Model

I also had a model to work from—not necessarily a good one, but one that on some level managed to achieve most of these goals in the character-mode DOS environment—Lotus Express.

Express is an unusual case. In many ways it is the best program available for what it does, and yet it remains a poor solution. It completely automates the process of sending and receiving MCI Mail messages, and it provides a good facility for storing messages off line. Moreover, because it can function as a memory-resident program, it can communicate with MCI in the background while you work on other applications. Nevertheless, every Express user I've ever talked to hates the program, condemning it as a slow, cumbersome memory hog with a user interface nearly as intractable as that of MCI Mail.

So, as I set out to build my MCI Mail application for Windows, I strove to duplicate the facilities and functions provided by Express, while taking advantage of both Windows' memory management capabilities and its user interface to avoid duplicating the DOS program's warts.

Selecting the Development Tool

I started developing the initial version of M.M.M. quite some time ago—several months before the release of Windows 3.0. That timing simplified the process of selecting a development tool, since most of the high-end tools that are discussed in this book were not yet available. Toolbook and Spinnaker Plus wouldn't be released for several months, and products like Visual BASIC, Realizer, and Turbo Pascal for Windows were then more than a year from release.

Thus, my choices soon boiled down to the macro languages of two Windows asynchronous communications programs: Crosstalk Communications' Crosstalk for Windows and Futuresoft Engineering's DynaComm. Not surprisingly, both languages were more than up to the task of automating the actual on-line session. An MCI Mail session really isn't that complex—you send a string of text to MCI, wait for its response, and then send the next string. Any macro language with even a modicum of decision-making capability (in the form of IF-THEN statements, WHILE loops, and so on) could handle that task.

The difference between the two languages lay in the capabilities they had that could be incorporated in the off-line portion of the application. DynaComm's script language offered the ability to create complex, dynamic dialog boxes and custom menus for interacting with the user, and included functions that could be used to build structured, disk-based tables to rapidly access lists of messages, address directories, and other information. In contrast, Crosstalk for Windows fell short in terms of both user interface and data-handling capabilities. Since I hoped to take advantage of the Windows interface, and because the ability to organize messages off line was of prime importance to the project, DynaComm was the obvious choice.

Other Possible Approaches

Today the range of high-level tools available for creating a project of this scope is much broader. Nevertheless, if I were starting the project over again today, I would probably still end up building it in DynaComm. Its macro language is powerful enough for nearly any communications-related task, and as noted, it provides remarkably powerful capabilities in the areas of user-interface design and data handling. Certainly it is hard to imagine that a general-purpose language would allow you to develop communications-based applications as quickly or easily as DynaComm does.

On the other hand, there are reasons why you might not want to use DynaComm—or any script language—for a project like this. For instance, if you were developing an application for a large number of users, it might make sense to develop it using a royalty-free high-level language such as Visual BASIC or Turbo Pascal for Windows so that you wouldn't have to

purchase a copy of DynaComm for each user. Doing so would be a more complex process than developing the application in a script language such as DynaComm's, because of the lack of communications-specific commands in a general-purpose language. However, you could overcome at least part of that complexity by making use of a commercial dynamic link library, such as the MicroHelp Communications Library, to provide those functions. The one-time fee for the DLL, if any, would soon pay for itself as you distributed the application.

In addition, a more general-purpose tool than DynaComm would undoubtedly provide even more control over the user interface and appearance of your application. For instance, although DynaComm allows you to create complex dialog boxes and custom menus, it does not allow you to place user-interface controls such as list boxes or buttons on its main window—they can only appear within dialog boxes. This, combined with DynaComm's inability to display more than one dialog box at a time, significantly restricts your user-interface design options. Since tools such as Turbo Pascal and Visual BASIC don't suffer from these limitations, they allow you to develop more creative interfaces than is possible with a script language such as DynaComm's.

M.M.M.'s Capabilities

M.M.M. has three major purposes: to simplify the creation and editing of MCI Mail messages, to automate the process of transmitting and receiving those messages, and to organize and manage messages that have already been sent or received.

Creating and Editing Messages

In order to simplify the creation and editing of messages, M.M.M. provides a full-screen editor. The editor offers the standard Windows cut, copy, and paste functions and other common text-editing capabilities, including word wrap, the ability to print some or all of the current document and to merge text files into the current document, and a search-and-replace function.

In addition, M.M.M. simplifies the process of creating the message's *envelope*—the addressee, subject, and special handling options for the message. It provides two point-and-click address books for storing and retrieving the names and MCI identifiers (IDs) of the people with whom you correspond. (The two lists, named Public and Private, are designed to be used as business (shared) and personal address books, respectively.) You can add an addressee to the message's To: or cc: list simply by picking the person's name from either address book.

M.M.M. also provides simple check-box controls for setting the handling options for a message. M.M.M. supports three of the most common MCI handling options: 4Hour, which assigns priority status to the message, speeding its delivery; DOC, which “hides” the list of addresses from recipients; and Receipt, which instructs MCI Mail to notify you when the message has been received.

All of M.M.M.’s message-creation and editing routines are used off line. As each message is created, it is added to your Outbox folder, a list of the messages that are to be delivered the next time M.M.M. connects to MCI Mail. (See “Organizing and Managing Messages,” below.)

Transmitting and Receiving Messages

M.M.M. offers two ways to transmit and receive messages: Send/Recv and AutoMCI.

Send/Recv is designed to be used when you want to either transmit messages immediately or know immediately what messages are waiting for you. In Send/Recv mode, M.M.M. signs on to MCI, transmits all the messages in your Outbox, retrieves any messages that are waiting for you, and then signs off, presenting you with a dialog box that tells you how many messages were transmitted and received. It also lets you know whether any messages could not be transmitted due to an addressing error, and adds those messages to an Unsent list, enabling you to correct the errors and send the messages later.

AutoMCI mode, on the other hand, is designed to operate in the background, checking your on-line mailbox at regular intervals, sending any messages in your Outbox and retrieving any messages that are waiting for you on line. It continues working this way, without interfering with your use of other Windows applications, until you tell it to stop doing so.

M.M.M. also provides a third communications mode, called Terminal, which logs you onto MCI Mail and then allows you to interact with MCI in a standard terminal session. The script’s address-book editing routines are available in this mode, allowing you to use MCI’s on-line facilities to look up someone’s correct account number and add that information to your address book. None of M.M.M.’s other automated features are available in this mode, however.

Organizing and Managing Messages

Finally, M.M.M. provides a number of facilities designed to help you manage the messages you have sent and received. These include a total of eight message lists, or *folders*.

The first five message folders are used for specific purposes by the M.M.M. program. They are *Inbox*, which is used to store messages that have been received; *Sent*, used to store messages that have been transmitted; *Outbox*,

which stores messages waiting for transmission; *Unsent*, for messages that can't be sent due to an error in their address; and *Drafts*, used to store messages in progress—ones you are not yet ready to transmit.

Another three message folders are designed to help you organize your incoming and outgoing messages more efficiently than is possible with the Inbox and Sent folders alone. Their default names are *UD1*, *UD2*, and *UD3*, but you can rename them at will (“UD” stands for *user-definable*). You can move messages from your Sent or Inbox folder into any of the three user-definable folders, which allows you to put all the messages pertaining to a particular project or topic in a single folder, providing an archival record of all the correspondence on that topic.

In addition to giving you the ability to move messages between folders, M.M.M. allows you to print messages, delete them, or save them under a different name. It also simplifies the processes of forwarding a message to another MCI user and replying to messages by providing single-click command buttons for both functions.

Finally, to simplify the use of M.M.M. itself, the script includes a completely automated setup routine, which guides you through the process of setting up M.M.M. for use with your account.

How M.M.M. Works

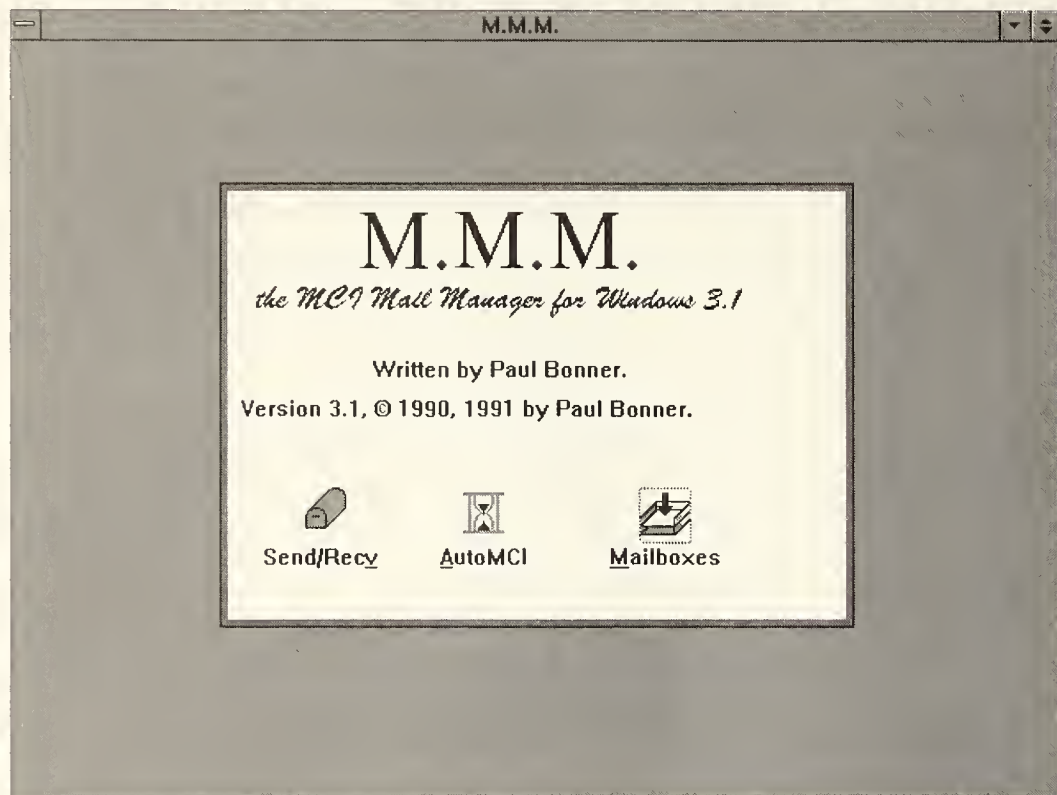
The M.M.M. application consists of five modules: AutoMCI, TM (To Module), Email, PM (Phonebook Module), and Online. The source file for each has the extension .DCP (as in AUTOMCI.DCP), and the compiled, executable file for each has the extension .DCT (as in AUTOMCI.DCT). The latter four modules serve as support for the AutoMCI module, and can only be executed by that module. (In fact, if you attempt to launch any of the support modules using DynaComm's Script Execute command, the support module will simply launch AutoMCI and then unload itself from memory.)

The first time you launch AutoMCI, the script will take you through the automated setup routine, which presents a series of dialog boxes requesting various information about your MCI account, including your logon name and password, the names you'd like to assign to the three user-definable message folders, and the location where the script should look for your Public address book. (Your Private address book is automatically stored in your \DYNACOMM\DAT directory, but your Public address book may be located on a network drive.)

Once you've finished entering that information, M.M.M. restarts, presenting you with its standard opening screen, shown in Figure 15.1. This dialog box gives you the choice of going to the Mailboxes screen (from which you can view and manage mail messages) or connecting to MCI Mail using either Send/Recv or AutoMCI mode.

Figure 15.1

The standard
M.M.M. welcome
message



The Mailboxes Screen

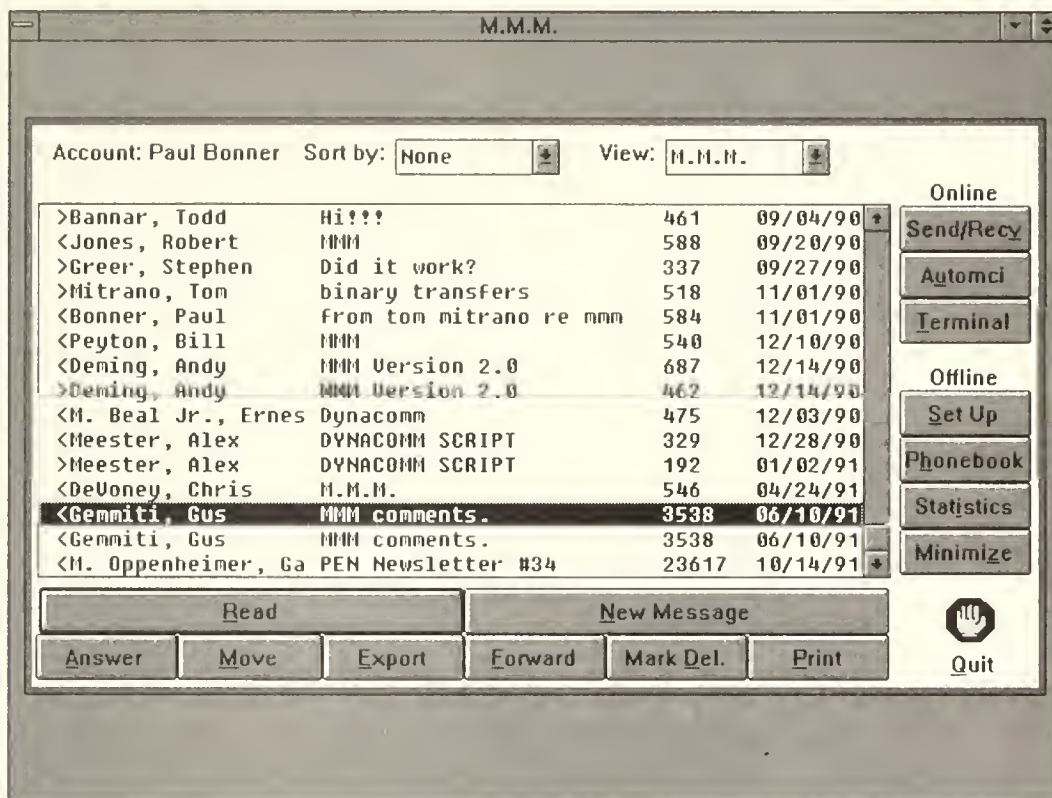
The Mailboxes screen, shown in Figure 15.2, is a large dialog box that provides access to, and tools for managing, messages in any of M.M.M.'s eight message folders.

The central feature of the Mailboxes screen is a large list box, which presents the messages in the currently selected mail folder and indicates whether each one was incoming or outgoing, who it was sent to or received from, its subject, the time and date it was received, and its size.

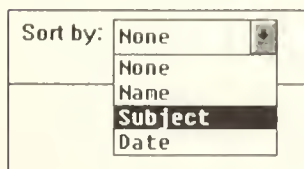
Within the Inbox folder, each message displays the name of its sender, whereas the Outbox, Drafts, Sent, and Unsent folders display the name of the addressee. In the three user-definable folders, the name displayed will be that of the message's sender or recipient, as appropriate; an incoming or outgoing indicator distinguishes between the two types of messages. Incoming messages are marked with a < before the sender's name, and outgoing messages have a > before the recipient's name.

Figure 15.2

The Mailboxes screen

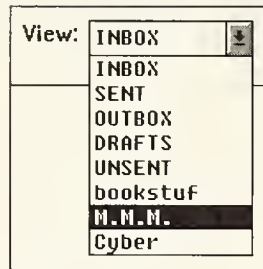


Above the central list box are two drop-down combo boxes. The first, labeled "Sort by:", presents four options for sorting the messages in the current folder, None, Name, Subject, and Date, as shown here:



Selecting None leaves the messages in their current order; Name sorts the messages by recipient or sender; Subject sorts them by subject; and Date sorts them by the date they were sent or received. (Note that Sort by Name will not sort correctly if senders' or recipients' names include a middle initial. To avoid this problem with recipients' names, enter the names in the address books without the middle initial, and then always use the address book entry to address messages, rather than using the Answer button.)

The second drop-down combo box, labeled “View:”, presents a list of the eight mail folders, as shown here:



Selecting any folder on the list immediately loads the list of the messages in that folder into the central list box.

That list box is surrounded by three groups of buttons: The one at the upper-right lists on-line options; the group below lists off-line options; and the buttons that run horizontally below the list box show message-handling options.

Online Options

There are three on-line option buttons: Send/Recv, AutoMCI, and Terminal. Their labels and actions correspond to the three methods M.M.M. provides for connecting to MCI Mail, as explained in “Transmitting and Receiving Messages,” above.

Offline Options

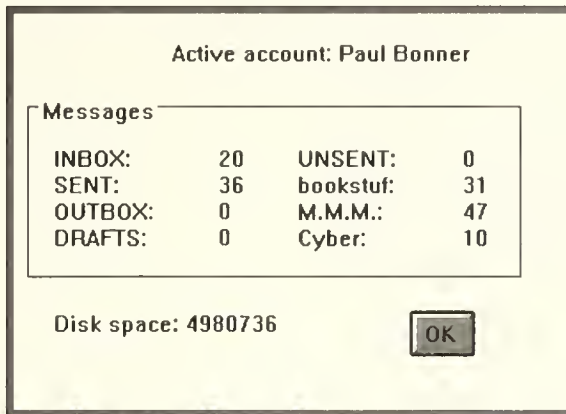
The Offline option buttons are labeled “Set Up,” “Phonebook,” “Statistics,” and “Minimize.” Pressing the Set Up button takes you to a screen that offers various program setup and maintenance options, as detailed in “The Setup Menu,” below. The Phonebook button loads the PM.DCP file, to allow you to add or edit names in one or both address books. The Statistics option opens a dialog box that displays the number of messages in each mail folder and the amount of unused storage space on the disk on which M.M.M. is installed, as shown in Figure 15.3.

The final off-line option, Minimize, is used to minimize the DynaComm/M.M.M. window. Normally, of course, applications don’t have minimize buttons—the user is expected to select the Minimize option from the application’s Control menu or the Minimize arrow at the upper-right corner of the application’s window. However, since DynaComm disables keyboard-shortcut access to its menus while a custom dialog box is displayed on screen, I added the Minimize button to the Mailboxes screen for the convenience of anyone using M.M.M. without a mouse. (You can toggle the input focus

between the dialog box and the menu by pressing Alt-F6, but that's the kind of arcane keystroke combination that is all too easy to forget.)

Figure 15.3

The Statistics dialog box



Message-Handling Options

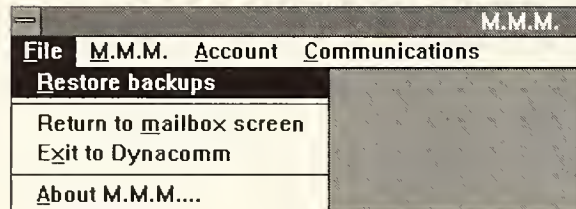
M.M.M.'s eight message-handling options appear in two rows of buttons at the bottom of the central list box. The top row has two large buttons: Read, which is used to open the message that is highlighted in the list box; and New Message, which loads the TM module, enabling you to address and compose a new message. The bottom row holds the remaining six, smaller buttons: Answer, which is used to address a reply to the current message; Move, which moves the current message to a different folder; Export, used to save the current message to disk under a different name; Forward, which forwards the message to another person; Mark Del., which marks the message for deletion; and Print, used for printing messages.

The Read button is the default button for the dialog box. Thus, anytime you want to read a message you simply move the list box selection bar until that message is highlighted, and press Enter to open the message.

When you select the Read command, the current message is loaded into DynaComm's memo editor as a read-only file. The editor, which the script fits with a custom menu bar, allows you to read the message, search it for a specified string, print the entire message or a selected passage, and copy segments of the message to the Windows Clipboard.

The Set Up Menu

When you select the button labeled “Set Up” on the Mailboxes screen, the dialog box disappears, and the script displays a custom menu offering access to various program setup and utility options, as shown here:



These options allow you to restore backup versions of mailbox lists (M.M.M. creates a backup whenever it modifies a mailbox list), create mailboxes to work with additional accounts (M.M.M. supports up to ten MCI accounts), switch to a different account, change the names of your user-defined mailboxes, and modify various modem and communications settings, such as baud rate and parity.

The Public Phonebook Location

The Set Up menu also allows you to specify the location of your Public phonebook. This setting is important because it determines whether you have the right to modify the Public phonebook. If the Public phonebook is located in the standard DynaComm data directory, along with your Private phonebook, then you are assumed to have administrator privileges. This means you can modify your Public phonebook, as well as your Private one, using the Phonebook Management dialog box (see below). If, on the other hand, the Public phonebook is located in a different directory, the assumption is made that you are accessing a shared phonebook on a network, so you are not given the option to modify it.

Thus, if you are a network administrator responsible for maintaining a Public phonebook for use by everyone on the network, you would make modifications to it in your standard DynaComm data directory, and then copy it to a shared disk drive for read-only access by other users.

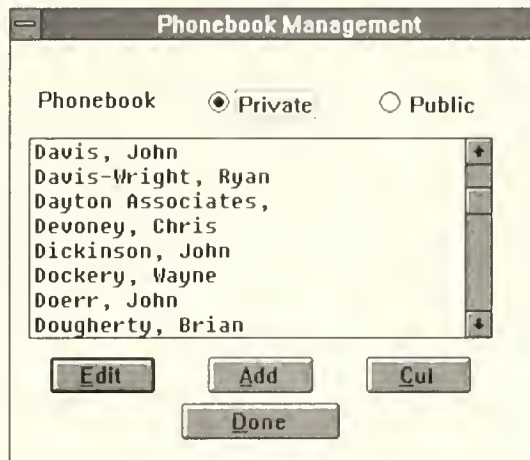
The Phonebook Management Routine

The Phonebook Management dialog box can be accessed in three ways: by pressing the Phonebook button on the main Mailboxes screen, by pressing a button that bears the same label on the dialog box used to address messages, or by selecting the function key button labeled “Phonebook” during an on-line terminal session.

The dialog box that appears after any of those events is shown in Figure 15.4. It offers you the ability to add, delete, or edit names in your Private

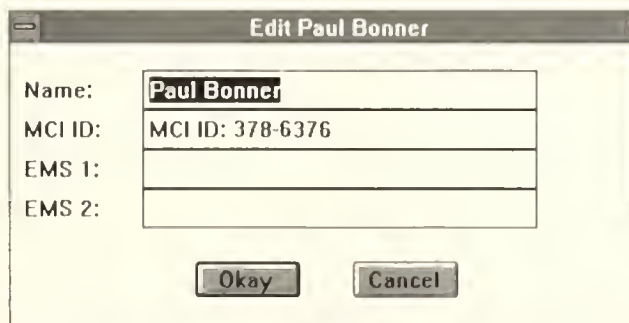
phonebook and in the Public phonebook if it is located in the DynaComm data directory, as explained above.

Figure 15.4
The Phonebook Management dialog box



Each entry in either phonebook can consist of up to four lines of information. Only the first two are used for MCI addresses: the first line holds the full name of the account holder, and the second line holds the person's account number, as shown in Figure 15.5. For EMS addresses ("EMS" stands for *electronic mail service*, and indicates that the addressee has an account on a mail service that interconnects with MCI) all four lines are used: The first holds the addressee's full name, the second holds the name of the EMS service, the third shows the EMS service's MCI account number, and the fourth holds the user's mailbox address on the EMS service.

Figure 15.5
Editing an MCI subscriber address



The key to the address book's value is that it will eliminate most MCI addressing errors. The biggest problem with automating the sending of MCI messages is eliminating the need for the script to make the kind of judgment calls that are so simple for a human user and so difficult for a computer.

For instance, say that you want to send a message to your friend Thomas Smith. You sign on to MCI and type **CR** (for *create*) and enter **Thomas Smith** at the “To:” prompt. Lo and behold, MCI replies by presenting you with the message “There is more than one:” and listing the names and account numbers of all 5 or 10 or 25 subscribers named Thomas Smith, along with their business affiliations, as shown in Figure 15.6. It then asks you to identify which of those names listed was the person with whom you wished to correspond.

Figure 15.6

MCI's “There is more than one:” error message

```

The MCI Mail Manager: Paul Bonner
2 characters or more are needed in "R".
Command: CREATE
TO:      Thomas Smith
There is more than one:
No.      MCI ID  Name                Organization  Location
0 NOT LISTED BELOW. DELETE.
1 NOT LISTED BELOW. ENTER AN ADDRESS.
2 REMOTE 470-4937 Thomas Smith    Digital      Chelmsford, MA
3        470-2492 Thomas Smith

```

If you know that Thomas Smith works for XYZ Corporation, that's an easy call; you simply look for the Thomas Smith listing that identifies XYZ as his business affiliation. But if an automated script doesn't “know” about XYZ, it has no way of identifying which Thomas Smith you meant, so it has to cancel the message-creation operation and add the message to your Unsent folder.

The address book helps solve this problem by encouraging you to enter each person's unique MCI account number. You can determine a person's account number by examining any message from that person (the sender's account number appears immediately after the sender's name in the message text) or by experimenting on line using the Create command. For instance, you could find the MCI account number of Thomas Smith of XYZ Corporation from the list of all Thomas Smiths, as described above.

Message-Composition Routines

Anytime you press the New Message, Answer, or Forward button on the Mailboxes screen, the script loads the TM module. This module presents you with a dialog box through which you enter all of the message's address and handling information, and then opens the message editor, in which you compose the message.

The message-addressing dialog box is a complex construct, as Figure 15.7 shows. The list box on the left lists all the names in either the Public or Private phonebook, depending on which of the radio buttons above the list box has been selected. Double-clicking on any name adds that name and its full MCI or EMS address to either the To list or cc list for the message, depending on which of those radio buttons has been selected. Alternately, you can type a name into the field labeled "Name:" and then press the Add button to add that name to the selected list.

Figure 15.7

The message-addressing dialog box

The screenshot shows a window titled "Edit Envelope". At the top left, there are two radio buttons: "Public" (selected) and "Private". Below them is a list box containing the following names: Richards, Katie; Ross, Randy; Seymour, Jim; Shipley, Chris; Steinberg, Don; Stinson, Craig; Taylor, Wendy; Tjelle, Linda; van Kirk, Doug; Wallace, Peggy; Ward, Patrick; Watts, Janice. To the right of the list box is a "Name:" text field and an "Add" button. Below the "Name:" field are two radio buttons: "To list" (selected) and "cc list". Underneath these is a large, empty rectangular text area. Below the text area are two buttons: "Edit name" and "Cut name". Below the "Cut name" button is a "Subject:" text field. At the bottom left, there is a "Cancel Message" button with a hand icon. At the bottom right, there is a "Done" button with a star icon. In the center-bottom area, there are three checkboxes: "Receipt", "Priority Delivery", and "Mask List Members". There is also an "Edit Phonebook" button located below the list box.

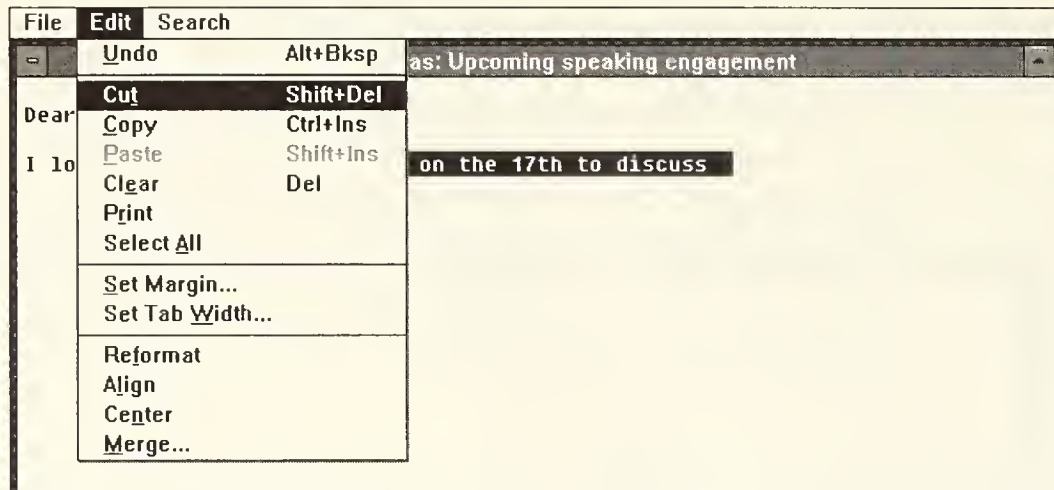
The other elements work more simply. You enter the message's subject in the field labeled "Subject:", and specify message-handling options by selecting the appropriate check boxes. Pressing the icon button labeled "Done" opens up the message editor, and pressing the icon button labeled "Cancel message" cancels the message and returns you to the Mailboxes screen.

The Message Editor menu used by the TM module is similar to that accessed via the Read button on the Mailboxes screen, but it includes options

for saving messages, cutting and pasting data, searching for and replacing data, and merging text files into the message, as shown in Figure 15.8.

Figure 15.8

The M.M.M.
Message Editor



That pretty much covers the basic capabilities and operations of M.M.M. Now let's look at how they are implemented in the five scripts that compose the M.M.M. application.

Exploring the AUTOMCI.DCP Script

AUTOMCI.DCP is a huge script, consisting of nearly 33k of source code and dozens of subroutines. Each subroutine tends to be fairly short, making it relatively easy to understand the purpose of each. But at the same time, the use of dozens of subroutines, combined with the event-driven nature of Windows applications in general, means the flow of action in the script is not a simple A-to-Z affair.

For the programmer, this difficulty is more than offset by the benefits to be gained from modularity, but for the casual reader, some perseverance may be required.

Initializing Global Settings and Variables

The AutoMCI script begins by executing a long series of statements that initialize various global variables and establish a variety of settings that control the program's operation and appearance.

The first few lines turn the standard arrow-shaped mouse pointer into a hourglass (to indicate that the program is working), cancel the standard DynaComm menu (by creating an empty menu with the MENU and MENU END commands), set DynaComm's window title to "M.M.M.", and set the

title to be used for the DynaComm terminal window to “The MCI Mail Manager”. Lines beginning with a semicolon are comments.

```
;AUTOMCI.DCT version 3.1 by Paul Bonner
SET POINTER WATCH
MENU
MENU END
TITLE "M.M.M."
SET TERMTITLE "The MCI Mail Manager"
```

Setting Up Directories

Next, the script initializes three directory pointers: \$Data_Dir, in which it will store universal data files; \$Mem_Dir, in which it will store messages and data files for individual mail boxes; and DIRECTORY SETTINGS, a DynaComm global variable used to store the location of the settings file, which in turn stores communications parameters and other information for use in on-line sessions.

The script uses the DynaComm SYSTEM command to determine its default Data and Task directories (the directories in which DynaComm looks for data files and script files until you tell it to look elsewhere), setting both \$Data_Dir and DIRECTORY SETTINGS to the default data directory and \$Mem_Dir to the default Task directory.

```
; ***** Set up directories
SET $Data_Dir SYSTEM(0x0F01,"DATA")
SET $Mem_Dir SYSTEM(0x0F01,"TASK")
SET DIRECTORY SETTINGS $Data_Dir
```

Once that has been done, the script checks to see whether the settings file it uses, MAILSET.DCS, is already in memory, which would indicate that M.M.M. is returning from another module. It does this by checking the value of the @S7 variable, one of eight global variables that are found in all DynaComm settings files.

A few lines further down, the script will set the value of @S7 to a string composed of “MMM” concatenated with the current data directory. Those values are retained when other M.M.M. modules are executed, but are discarded when you quit M.M.M. So if the first three characters in @S7 are “MMM”, then the script knows that MAILSET.DCS is already in memory and holds valid data. Thus, it changes the current data directory to the directory specified in @S7 following the letters “MMM”, and then jumps ahead to a procedure called Alt_Start, skipping the process of loading the settings file from disk.

If the first three characters in @S7 don't equal “MMM”, the script hides the terminal session screen and on-screen function keys and loads MAILSET.DCS.

```

IF SUBSTR(@S7,1,3)="MMM" SET %I LENGTH(@S7), SET $Data_Dir
    SUBSTR(@S7,4,%I-3), SET DIRECTORY DATA $Data_Dir, GOTO
    Alt_Start
SCREEN HIDE
LOAD "MAILSET.DCS"
FKEYS HIDE

```

Initializing Global Variables

In the Alt_Start routine, the script initializes a series of global variable names that are used to hold the names of up to ten MCI accounts (\$Account1 through \$Account10), the number of accounts in use (%AccountCounter), the user's MCI account name and password (\$Name and \$Pass), the title of the active mail folder (\$Mailbox), and the titles of the three user-definable mail folders for the active account (\$Ms6, \$Ms7, and \$Ms8).

```

*Alt_Start
SET $Account1 ""
SET $Account2 ""
SET $Account3 ""
SET $Account4 ""
SET $Account5 ""
SET $Account6 ""
SET $Account7 ""
SET $Account8 ""
SET $Account9 ""
SET $Account10 ""
SET %AccountCounter 0
SET $Account ""
SET $Name ""
SET $Pass ""
SET $Mailbox ""
SET $Ms6 ""
SET $Ms7 ""
SET $Ms8 ""

```

The above routine merely initializes all the string variables to empty strings, and %AccountCounter to zero. Later it will branch to subroutines that assign real values to the variables. However, it is necessary to initialize them here because of the way DynaComm's variable scoping rules operate. A variable that is initialized in a subroutine in DynaComm is active only in that subroutine and any subroutines that it calls, and ceases to exist as soon as the subroutine that initialized it returns to its calling routine. So in order to have these variables available throughout the program, it is important to initialize them here, not in a called subroutine.

The script then continues setting additional global variable names. The next sequence sets the names of the five standard mail folders.

```
SET $Ms1 "INBOX"
SET $Ms2 "SENT"
SET $Ms3 "OUTBOX"
SET $Ms4 "DRAFTS"
SET $Ms5 "UNSENT"
```

Next the script checks the value of the @S7 system variable again, this time to determine if it holds the value "newuser". The only time the value of @S7 will be equal to "newuser" is the first time you run the script after installing it on your system. The value "newuser" tells the script to jump to the first-time setup routine, which is called SetupMMM.

```
IF @S7 = "newuser" PERFORM SetupMMM
```

Otherwise, the script extracts the values of several global variables from the system variable @S6: %Account (the active-account indicator), %Init (a program-status indicator), and \$PubPath (the location of the Public phone-book). These variables are stored in @S6, separated by a tilde character (~), so the script obtains them by using the DynaComm PARSE command to break the contents of @S6 into two strings: the characters that precede the first tilde contained in the variable, and the characters that follow it.

```
SET $Stuff @S6
PARSE $Stuff $Stuff1 "~" $Stuff
SET %Account NUM($Stuff1)
PARSE $Stuff $Stuff1 "~" $Stuff
SET %Init NUM($Stuff1)
SET $Pubpath $Stuff
```

For example, if the initial contents of @S6 are "1~1~C:\DYNACOMM\DAT", then after the first PARSE command \$Stuff1 contains the string "1", which the subsequent NUM command converts to numeric form for use in the integer variable %Account. \$Stuff, meanwhile, contains "1~C:\DYNACOMM\DAT", so the script parses it again to break it into two strings, the first containing "1" (which it then converts to numeric form for assignment to the integer variable %Init), and the second containing "C:\DYNACOMM\DAT", which it assigns to \$PubPath.

Next, the same technique is used to extract the MCI Mail telephone number and the strings \$Pre_Dial and \$Post_Dial from the system variable @S3. Then the user-defined delay between AutoMCI connections (%Delay) is obtained from the variable @S4.

```

PARSE @S3 $NUM "`" $Pre_Dial
PARSE $Pre_Dial $Pre_Dial "`" $Post_Dial
SET %Delay NUM(@S4)

```

The script next sets the values of @S7 and @S8, assigning the concatenation of “MMM” and \$Data_Dir to @S7, and the value -1 to @S8. The value of @S8 is used by the EMAIL.DCT module to determine whether it should execute the SendRecv (@S8 = -1) or AutoMCI sequence (@S8 > -1). Setting it to -1 here establishes SendRecv as the default choice.

```

SET @S7 "MMM" | $Data_Dir
SET @S8 STR(-1)

```

Next, the script sets the value of \$Ms (used to identify the current mailbox) equal to \$Ms1, or “Inbox”, and then initializes several more global variables. Their use will be described below.

```

SET $Ms $Ms1
SET %Forward 0
SET %Answer 0
SET %Move 0
SET %MENU 0
SET %I 9999
SET %MarkDel 0
SET %NV 0

```

Obtaining Data

At this point the script executes a series of several subroutines that it uses to load data tables into memory and to obtain real values for some of the variables it has already initialized with dummy values. It starts with the Account_Data subroutine, from which it obtains the account name, log-on name, and password for the current account, the names of all the other accounts, and other account-related data. The Account_Data routine also calls the Tables routine, which loads into memory the tables that list the contents of the eight mail folders for the current account—designated as tables 0 through 7. Then, upon returning from Account_Data, the script performs the Get_Lists subroutine, in which it builds two text tables, named 10 and 11. It uses table 10 to store the mail folder names for the View: list on the Mailboxes screen, and table 11 to store the four choices for the Sort by: list.

```

PERFORM Account_Data
PERFORM Get_Lists

```

At this point the script turns the cursor back into an arrow; sets %Table (the pointer to the active table) to 0, making Inbox the active table; and then performs a routine called Stat_Check. If you have just executed the script

(Stat_Check determines this based on the contents of %Init), the Stat_Check routine displays a welcome message offering you the option of going to the Mailboxes screen or launching the Send/Recv or AutoMCI routine.

If, on the other hand, the script is returning from a Send/Recv or AutoMCI session in which mail was transferred, Stat_Check displays a dialog box that shows the number of messages sent, received, and unsent during the session. However, if no mail was transferred during the session, Stat_Check returns without displaying any dialog box.

```
SET POINTER ARROW
SET %Table 0
PERFORM Stat_Check
```

Then the script sets %Init to 0 and performs the Saveall routine, which saves the settings file and all the global variables to disk, concluding the initialization section of the script.

```
SET %Init 0
PERFORM Saveall
```

Analyzing the Main Routine

The Main routine presents the Mailboxes dialog box, which acts as the control center for the M.M.M. application. You saw it earlier in Figure 15.2.

Drawing the Dialog Box

The routine begins by using DynaComm's ICONIC() function to determine whether the M.M.M. program has been minimized. If so, it simply loops back to beginning of the Main routine, staying in this short loop until DynaComm is restored. This prevents it from drawing its dialog box on screen while you're using another program, and thus interrupting your work.

Otherwise, the routine uses the DIALOG command to start drawing the Mailboxes dialog box. The DIALOG command has the following syntax:

```
DIALOG (Horizontal, Vertical, Width, Height) "Title"
```

The horizontal and vertical coordinates control the dialog box's location within the DynaComm window, and the width and height coordinates control its size. The coordinates are specified in terms of a logical unit. Each character is four units wide, and each line eight units high. The point of origin is the upper-left corner of the screen. Thus, a position six characters to the right and five rows down from the top-left corner would be 24,40.

The size and location coordinates are optional, as is the title string. If you omit any of the coordinates, DynaComm will attempt to determine them automatically, based on the contents of the dialog box. Specifying

them gives you greater control, obviously, and usually yields a more attractive dialog box.

The presence or absence of a title string in the dialog statement determines the style of the dialog box. If you specify a title string, the dialog box has both a title bar (which makes it movable) and a Control menu. If you omit the title string, it has neither element.

The dialog statement in the Main routine omits the horizontal coordinate, allowing DynaComm to center the dialog box within the DynaComm window. It also omits the title string.

```
*MAIN
IF ICONIC() GOTO MAIN
DIALOG (,4,309,176)
```

The script then proceeds to draw the dialog box. It starts with a MESSAGE command, which simply displays the string “Account:” concatenated with the name of the current account as static text. The script then creates the central list box, instructing it to display table %Table and to highlight line %I of the table. Then it draws the two combo box controls, preceding them with the static text strings “Sort by:” and “View:”.

The Sort by and View Commands

The command for creating combo boxes has the syntax

```
LISTBOX (H,V,W,H) %Table %Pos COMBOBOX
```

Everything following the COMBOBOX statement is a command string that is only evaluated when you select an item from the list box.

The drop-down Sort by: list presents four options: None, Name, Subject, and Date, which appear as list-box entries 0 through 3. When the user selects any item on this list other than the zero element (None), the script jumps to the Sort subroutine. When that routine is finished, it updates the central list box to display the messages in the active folder in their new sort order.

```
MESSAGE (4,4,,) " Account: " | $Account
LISTBOX (2,24,258,112) %Table %I
MESSAGE (82,4,,) "Sort by:"
LISTBOX (110,4,50,44) 11 0 COMBOBOX SET %Sort LISTBOX(2), IF
%Sort>0 PERFORM Sort, DIALOG UPDATE LISTBOX 1 TABLE %Table %I
```

The View: drop-down list offers the user a choice of all eight message folders. When the user selects a new message folder from the View: list, the script calls the Ver_Clr routine, which determines whether any messages in the active folder have been marked for deletion and, if so, whether the user wishes to delete them. This routine is called whenever the script is about to execute a sequence of commands in which the active folder might change. Since the script only tracks the *number* of messages that have been marked

for deletion in the active folder, it must determine whether they should actually be deleted before it activates a different table. If it doesn't, the information about what is to be deleted is lost.

```
MESSAGE (172,4,,) "View:"
LISTBOX (192,4,50,78) 10 %Table COMBOBOX SET %New_TABLE LISTBOX(3).
    PERFORM Ver_Clr(%NV), IF %NV=0 SET %Table %New_Table, PERFORM
    Dialog_Update, ELSE IF %NV=1 SET %Table %New_Table, RESUME, ELSE IF
    %NV=2 RESUME
```

If the `Ver_Clr` routine returns `%NV` with a value of 0, there were no messages marked for deletion, so the routine simply sets the `%Table` variable, which indicates the active table, equal to the value of `%New_Table`, which indicates the table that the user selected from the drop-down list. Then it updates the central list box to show the new table.

If `Ver_Clr` returns a value of 1 for `%NV`, then there were messages marked for deletion, so `Ver_Clr` had to cancel the Mailboxes dialog box in order to display its own dialog box. Thus, the routine can't get away with simply updating the central table. Instead, it must issue the `RESUME` command, which results in the entire Mailboxes dialog box being repainted.

If `Ver_Clr` returns a value of 2, then the user canceled the operation. So the View-list command sequence comes to an end by issuing the `RESUME` command to repaint the Mailboxes dialog box.

The Message-Handling Commands

Next, the script creates the eight message-handling buttons, which appear below the central list box in the Mailboxes dialog box (Figure 15.2). The button-creation command has the syntax

```
"BUTTON (H,V,W,H) default title command
```

The optional *default* setting may consist of one of two keywords: `DEFAULT` or `CANCEL`. Specifying the `DEFAULT` keyword makes the button the default option for the dialog box, meaning it will automatically be selected if the user presses Enter without having tabbed to a different button. Specifying `CANCEL`, on the other hand, makes that button the cancel button for the dialog box, which means that it will be selected when the user presses the Esc key. The command sequence that follows the title string will be evaluated only when the button is selected.

```
BUTTON (2,140,129,14) DEFAULT "&Read" PERFORM Rec %I, IF %I = 1 DIALOG
    CANCEL, PERFORM Read, RESUME, ELSE SET %I 0
```

The `Read` command sequence, the default command, starts by performing the `Rec` subroutine, passing it the variable `I%`. The `Rec` routine determines which record is currently highlighted in the list box and then checks to make sure the record is not empty (which it would be if the list box were

empty). If the record is valid it returns the record number in the variable %I. Otherwise it sets %I to -1.

If %I comes back from the Rec subroutine with a value greater than -1, the Read subroutine is performed. Otherwise the command sequence sets %I to 0 and ends.

The New Message command sequence begins by calling the Ver_Clr routine to determine if any messages are marked for deletion and, if so, what the user wants to do with them. If Ver_Clr returns a value of less than 2, the routine continues. First it sets the value of %Init to 0, and then it launches the TM routine in the TM.DCT script, passing it the current values of \$Data_Dir, \$Mailbox, %Forward (which indicates whether a message is being forwarded), %Answer (which indicates whether the message is a reply), and \$Pubpath. When that routine concludes, the New Message routine continues by performing the Get_Lists routine, and then issuing the RESUME command, which redraws the Mailboxes dialog box.

```
BUTTON (131,140,129,14) "&New Message" PERFORM Ver_Clr(%NV), IF %NV < 2
    DIALOG CANCEL, SET %Init 0, PERFORM "TM*Tm" ($Data_Dir,
    $Mailbox,%Forward,%Answer, $Pubpath), PERFORM Get_Lists, RESUME,
    ELSE RESUME
```

If Ver_Clr returns a value of 2 for %NV, it means that the user elected to cancel the operation, so the routine simply issues the RESUME command to redraw the Mailboxes dialog box.

The Answer command sequence begins by performing the Rec subroutine. If Rec returns a value for %I that is greater than 0, the script then jumps to a subroutine called Ans_Button.

```
BUTTON (1,154,43,14) "&Answer" PERFORM Rec(%I), IF %I > 0 PERFORM
    Ans_Button, RESUME, ELSE SET %I 0
```

The command sequences for the Move, Export, and Forward buttons are all similar to that for the Answer button. They begin by performing the Rec subroutine, and then, if %I is greater than -1, jump to routines called Move, Export, and For_Button, respectively, and finally issue the RESUME command.

```
BUTTON (44,154,44,14) "&Move" PERFORM Rec(%I), IF %I > 0 PERFORM Move,
    RESUME, ELSE SET %I 0
BUTTON (88,154,43,14) "&Export" PERFORM Rec(%I), IF %I > 0 DIALOG CANCEL,
    PERFORM Export, RESUME, ELSE SET %I 0
BUTTON (131,154,43,14) "&Forward" PERFORM Rec(%I), IF %I > 0 PERFORM
    For_Button, RESUME , ELSE SET %I 0
```

The Mark Del command sequence is similar to those that immediately precede it. It performs Rec, then passes the values of %Table and %I to the Mark_Del routine, and finally performs the Dialog_Update command to

update the contents of the central list box without redrawing the entire dialog box.

```
BUTTON (174,154,43,14) "Mark &Del." PERFORM Rec(%I), IF %I > 0 PERFORM
  Mark_Del(%Table, %I), INCREMENT %I, PERFORM Dialog_Update, ELSE SET
  %I 0
```

The scheme of first marking messages for deletion and then later deleting them serves two purposes. First, it gives the user a chance to change his or her mind about deleting a message. Second, it speeds up the processing of the script because deleting messages one at a time can be a very slow process.

DynaComm doesn't offer a record deletion command. Instead, the only way to delete a record from a table is to copy the active table to a temporary table, and then copy the temporary table back to the active table minus those records you want to delete. Once a table has grown to hold several hundred records, this can be a quite time-consuming process, taking 20 seconds or more with very large tables. By using a batch deletion routine, however, the process need only be performed once, no matter how many records have been marked for deletion, rather than once for each record.

Finally, the Print command sequence jumps to the routine Print_Mess if Rec returns a non-negative value for %I.

```
BUTTON (217,154,43,14) "&Print" PERFORM Rec(%I), IF %I > 0 DIALOG CANCEL,
  PERFORM Print_Mess, RESUME, ELSE SET %I 0
```

The Online Commands

Next the script creates the buttons for the three on-line options, Send/Recv, AutoMCI, and Terminal.

```
MESSAGE(272,16,..) "Online"
BUTTON (263,24,40,14) "Send/Recv" PERFORM Ver_Clr(%NV), IF %NV < 2 SET
  @S8 STR(-1), SET %INIT 0, PERFORM Saveall, SET DIRECTORY DATA
  $Data_Dir | $Mailbox, SCREEN SHOW, EXECUTE 'email', ELSE RESUME
BUTTON (263,38,40,14) "A&utomci" PERFORM Ver_Clr(%NV), IF %NV < 2 SET
  @S8 STR(0), SET %Init 0, PERFORM Saveall, SET DIRECTORY DATA
  $Data_Dir | $Mailbox, SCREEN SHOW, EXECUTE "EMAIL" . ELSE RESUME
BUTTON (263,52,40,14) "&Terminal" PERFORM Ver_Clr(%NV), IF %NV < 2 SET
  %Init 0, SET @S7 "", PERFORM Saveall, SET DIRECTORY DATA $Data_Dir,
  EXECUTE "ONLINE", ELSE RESUME
```

The command sequences for the three on-line options are all very similar. Each starts by performing Ver_Clr. If the user doesn't cancel the operation (that is, %NV < 2), the script continues by setting the value of @S8 to -1 (in the case of the Send/Recv command) or to 0 (for AutoMCI).

The @S8 setting doesn't matter during terminal sessions, so the Terminal command sequence doesn't change it. However, it does set @S7 to an empty string, which will force the script to reload the settings file the next time AutoMCI is executed.

Next, all three command sequences set %Init to 0 and perform the Saveall routine. Then they set the DynaComm data directory setting to \$Data_Dir, and launch the script that will perform the actual communications session (EMAIL.DCT in the case of the Send/Recv and AutoMCI command sequences, and ONLINE.DCT in the case of the Terminal command sequence).

The Offline and Quit Commands

Next the script draws the Set Up, Phonebook, Statistics, Minimize, and Quit buttons.

The Set Up command sequence performs Ver_Clr, then sets %I to the current list box record, cancels the dialog box, and performs the Menu subroutine.

```
MESSAGE(272,72,,) "Offline"
BUTTON (263,80,40,14) "&Set Up"  PERFORM Ver_Clr(%NV), IF %NV < 2 SET %I
LISTBOX(1), DIALOG CANCEL, PERFORM Menu, RESUME, ELSE RESUME
```

The Phonebook command sequence sets %I equal to the current list box record, cancels the dialog box, executes the subroutine named Phon_Man in the PM script, and then resumes.

```
BUTTON (263,94,40,14) "P&honebook" SET %I LISTBOX(1), DIALOG CANCEL,
PERFORM "PM*Phon_Man" (0), RESUME
```

The Statistics command sequence sets %I to the current list box record, cancels the dialog box, and performs the Stats routine before resuming.

```
BUTTON (263,108,40,14) "Stat&istics" SET %I LISTBOX(1), DIALOG CANCEL,
PERFORM Stats, RESUME
```

The Minimize command sequence simply minimizes the DynaComm window, cancels the dialog box, and resumes. Since the RESUME command loops back to the beginning of the Main routine, where the IF ICONIC() command is found, the dialog box will be redrawn only when the window is restored.

```
BUTTON (263,122,40,14) "Minimi&ze" WINDOW MINIMIZE, DIALOG CANCEL, RESUME
```

The Quit option is actually presented in the form of an icon button. The “_stop” entry immediately after the button’s coordinates tells DynaComm to use the icon named _stop from the DCICON3.DLL icon library when it draws the button.

```
ICONBUTTON (267,142,36,14) "_stop" "&Quit" PERFORM Ver_Clr(%NV), IF %NV <
2 SET %Init 1, SET @S7 "", PERFORM Saveall, SET DIRECTORY DATA
$Data_Dir, PERFORM Close_Em, MENU CANCEL, QUIT, ELSE RESUME
```

The Quit command sequence performs Ver_Clr, then sets %Init to 1 and @S7 to an empty string before performing the Saveall subroutine. Then it

resets the data directory to \$Data_Dir, performs the Close_Em subroutine to close all open tables, cancels the current (empty) menu, and ends the script with the DynaComm QUIT command.

The End of Main

The Main routine concludes with the DIALOG END command (which tells DynaComm that the dialog box definition is complete) followed by a WAIT RESUME command (which tells DynaComm to pause script execution until it receives a RESUME command).

```
DIALOG END
WAIT RESUME
MENU
MENU END
GOTO Main
```

The RESUME command the script is waiting for is only issued when there is a need to redraw the dialog box, which generally occurs when a command button has launched a series of events that cancels the Mailboxes dialog box and substitutes another in its place. When the RESUME command is received, the script cancels any menus that may have been drawn by the processes launched from the dialog box's command buttons, and then issues the command GOTO Main, which jumps back to the beginning of the Mailboxes dialog box routine, redrawing the box and then once again waiting for a RESUME statement.

Mailboxes Screen Support Routines

The next five subroutines are support routines for the main Mailboxes dialog box: Dialog_Update, For_Button, Ans_Button, Rec, and Ver_Clr.

The Dialog_Update Routine

The Dialog_Update routine is used to update the contents of the central list box and the Sort by: combo box. Whenever this routine is called, it refreshes the two lists, setting the Sort by: list to element 0 (the None option) and instructing the central list box to display table %Table and to highlight element %I of that table.

```
*Dialog_Update
DIALOG UPDATE LISTBOX 2 TABLE 11 0
DIALOG UPDATE LISTBOX 1 TABLE %Table %I
RETURN
```


The For_Button Routine

The For_Button routine starts by performing Ver_Clr. Unless the user cancels the operation, it performs the For_Set routine, which creates a copy of the current message for forwarding, and then executes the Tm routine in TM.DCT module. When that routine is finished, the For_Button routine concludes by reloading the View: and Sort by: lists.

```
*For_Button
PERFORM Ver_Clr(%NV)
IF %NV < 2 SET %Forward 1,
PERFORM For_Set,
PERFORM "TM*Tm" ($Data_Dir, $Mailbox,%Forward,%Answer, $Pubpath),
PERFORM Get_Lists, RETURN,
ELSE RETURN
```

The Ans_Button Routine

The Ans_Button subroutine starts by performing Ver_Clr, and then, unless the user cancels the operation, it cancels the Mailboxes dialog box, performs the Ans_Set routine (which obtains information about the current message's sender and subject matter for use in addressing the reply), then calls the Tm routine in the TM module. When that routine concludes, Ans_Button finishes up by updating the Sort by: and View: lists before returning control to the routine that called it.

```
*Ans_Button
PERFORM Ver_Clr(%NV)
IF %NV < 2 DIALOG CANCEL,
SET %Answer 1,
PERFORM Ans_Set,
PERFORM "TM*Tm" ($Data_Dir,$Mailbox,%Forward,%Answer,$Pubpath),
PERFORM Get_Lists,
RETURN,
ELSE RETURN
```

The Rec Routine

The Rec subroutine, which is called by all the message-handling buttons, simply sets the value of %I to the record number of the message that is currently highlighted in the central list box. Then it checks to make sure that the active table actually has a valid record with that number. This is necessary because if the LISTBOX() function is performed on an empty list, it will return a value of 0, which is the same value returned by the first element in a list that does contain records. Rec determines the record's validity by reading the record and then examining the third character in the record variable, which in a valid record will never be an empty space. If the third character is a space, the record is not valid, so Rec sets %I to -1.

```

*Rec (%I)
SET %I LISTBOX()
RECORD READ %Table AT %I
IF SUBSTR(@R%Table,3,1)=" " %I=-1
RETURN

```

The Ver_Clr Routine

The Ver_Clr routine begins by setting the value of %NV to 0, and then checks the value of the variable %MarkDel, which is used to keep track of the number of messages in the current folder that have been marked for deletion. If no messages have been marked (%MarkDel=0) the routine ends. Otherwise, it sets the variable %CD equal to %NV and passes it to the Check_Del routine, which creates a dialog box asking the user how the messages that are marked for deletion should be treated.

```

*Ver_Clr(%NV)
SET %NV 0
IF %MarkDel=0 RETURN
SET %CD %NV
PERFORM Check_Del(%CD)
IF %CD=0 SET %NV 2,
ELSE SET %NV 1
RETURN

```

Check_Del will return %CD with one of two values. If %CD comes back with the value 0, it indicates that the user canceled the operation. In that case, %NV is set to 2, which will indicate to the calling routine that the operation is to be canceled. Otherwise, %CD will return with the value 1, indicating that the messages were either deleted or the marks were removed from them. If %CD has the value 1, then the operation that called Ver_Clr is to proceed, so Ver_Clr sets %NV to 1 and returns.

The Set Up Menu Routine

The Set Up Menu routine creates and displays the menu of setup and utility functions (shown below) that appears when you press the Set Up button on the Mailboxes dialog box.



The image shows a horizontal menu bar with a dark background. On the left, the word 'File' is highlighted in white. To its right, the text 'M.M.M. Account Communications' is displayed in a light color.

The routine begins by setting the mouse pointer to an arrow shape, and setting the DynaComm data directory to \$Data_Dir. Then the menu-definition process begins, with the command MENU.

```
*Menu
SET POINTER ARROW
SET DIRECTORY DATA $Data_Dir
MENU
```

The File Menu

The POPUP command creates the File item on the menu bar.



The first ITEM command beneath POPUP creates the first item on the File menu, labeled “Restore Backups”, and instructs the script to perform the Restore routine when that item is selected.

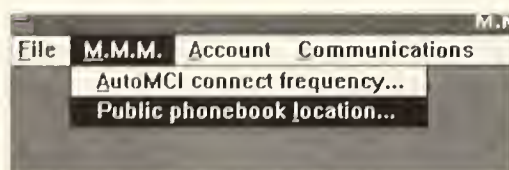
```
POPUP "&File"
ITEM "&Restore backups" PERFORM Restore
SEPARATOR
ITEM "Return to &Mailboxes screen" MENU, MENU END, RETURN
ITEM "E&xit to Dynacomm" SET %Init 1, SET DIRECTORY DATA $Data_Dir, SET
    @S7 "", PERFORM Saveall, PERFORM Close_Em, MENU CANCEL, CANCEL
SEPARATOR
ITEM "&About M.M.M....", PERFORM About
```

The next line creates a separator bar, which is followed by two items offering ways to leave the Menu screen. “Return to Mailboxes screen” cancels the menu and returns to the Main routine, whereas “Exit to DynaComm” cancels the script, leaving DynaComm running (in contrast to the Quit button on the Mailboxes screen, which ends both the script and DynaComm).

The next item on the File menu is another separator bar, which is followed by the final item on the menu, “About M.M.M....”. When selected, this item calls a subroutine called About, which displays information about the script and its authorship.

The M.M.M. Menu

The next menu is labeled “M.M.M.” and includes two menu items, as shown here:



The first option branches to a routine called Auto_Freq, which enables the user to determine the frequency of AutoMCI connections, and the second performs the Pub_Book routine, which is used to enter the location of the Public address book.

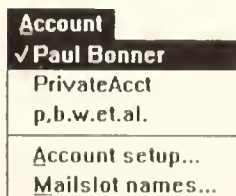
```
POPUP "&M.M.M."
ITEM "&AutoMCI connect frequency..." PERFORM Auto_Freq
ITEM "Public phonebook &location..." PERFORM Pub_Book
```

The Account Menu

The next popup menu, labeled "Account", is considerably more complex than the preceding two because its contents vary according to the number of MCI accounts you have set up within M.M.M. If you have set up only one account, the Account menu will list only two items: Account Setup..., which jumps to the Set Up routine to enable to you set up a new account or modify an existing one, and Mailslot names... which performs first the Slots routine through which you can enter or modify the names of your user-definable mail folders, and then Get_Lists to update the list of mail folder names used by the Mailboxes screen.

```
POPUP "&Account"
IF %AccountCounter > 1 ITEM $Account1 UNCHECKED SET %Choice 0, PERFORM Pick, RESUME
IF %AccountCounter > 1 ITEM $Account2 UNCHECKED SET %choice 1, PERFORM Pick, RESUME
IF %AccountCounter > 2 ITEM $Account3 UNCHECKED SET %choice 2, PERFORM Pick, RESUME
IF %AccountCounter > 3 ITEM $Account4 UNCHECKED SET %choice 3, PERFORM Pick, RESUME
IF %AccountCounter > 4 ITEM $Account5 UNCHECKED SET %choice 4, PERFORM Pick, RESUME
IF %AccountCounter > 5 ITEM $Account6 UNCHECKED SET %choice 5, PERFORM Pick, RESUME
IF %AccountCounter > 6 ITEM $Account7 UNCHECKED SET %choice 6, PERFORM Pick, RESUME
IF %AccountCounter > 7 ITEM $Account8 UNCHECKED SET %choice 7, PERFORM Pick, RESUME
IF %AccountCounter > 8 ITEM $Account9 UNCHECKED SET %choice 8, PERFORM Pick, RESUME
IF %AccountCounter > 9 ITEM $Account10 UNCHECKED SET %choice 9, PERFORM Pick, RESUME
IF %AccountCounter > 1 SEPARATOR
ITEM "&Account setup..." PERFORM Setup, RESUME
ITEM "&Mailslot names..." PERFORM Slots, PERFORM Get_Lists, RESUME
```

However, if you have more than one account, several additional items appear on the menu: a menu item for each of your accounts, and a separator bar separating the account names from the Account setup... and Mailslot names... options, as shown here:



When you select an account name from this list, the routine sets the value of %Choice to indicate the account you selected, and then calls the Pick routine, which processes your choice.

The Communications Menu

The next item to appear on the menu bar is labeled "Communications". The menu looks like this:



The Communications settings item calls the standard DynaComm Communications dialog box, shown in Figure 15.9, using the command SETTINGS COMMUNICATIONS. The Modem settings option makes use of the same method to access DynaComm's standard modem settings routine. The AutoMCI settings item calls a subroutine called Auto_Set to obtain various settings for the AutoMCI routine from the user. And the MCI phone number item calls the Access_Num routine to obtain the correct telephone number to use in accessing MCI. Here's what it looks like:

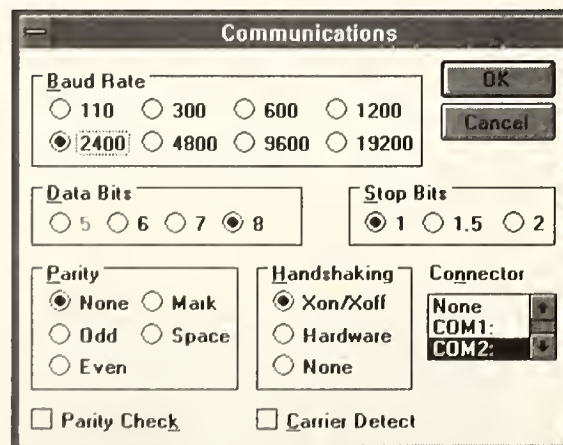
```

POPUP "&Communications"
ITEM "&Communications settings..." SETTINGS COMMUNICATIONS
ITEM "Mo&dem settings..." SETTINGS MODEM
ITEM "&AutoMCI settings..." PERFORM Auto_Set
ITEM "&MCI phone number..." PERFORM Access_Num

```

Figure 15.9

The standard DynaComm communications settings dialog box



The End of Menu

Finally, the Menu routine concludes with these lines:

```
MENU END
PERFORM View_Check
WAIT RESUME
GOTO MENU
```

Following the MENU END statement, which concludes the menu-definition process, the script immediately performs a subroutine called View_Check, which places a check mark beside the active account on the Accounts menu (when you have set up M.M.M. to work with more than one account). It then waits for a RESUME statement, to which it responds by jumping back to the start of the Menu routine, and thus redrawing the menu.

Menu-Support Routines

The next several routines in the AutoMCI.DCP script are menu-support routines that are called either during the menu-building process or in direct response to a menu choice made by the user. First come the View_Check routine and View_Check_2, a subroutine that it calls.

The View_Check Routine

The View_Check routine begins by initializing a variable called %Temp with a value of 2, and then compares the variable %AccountCounter, which keeps track of the number of active accounts, to %Temp.

```
*View_Check
%Temp=2
IF %AccountCounter < %Temp RETURN, ELSE IF %Account=0 MENU UPDATE 3 1
    CHECKED, ELSE MENU UPDATE 3 1 UNCHECKED
WHILE %Temp < 10 PERFORM VIEW_CHECK_2 (%Temp), INCREMENT %Temp
RETURN

*View_Check_2 (%Temp)
IF %AccountCounter < %Temp RETURN, ELSE IF %Account=%Temp-1 MENU UPDATE 3
    %Temp CHECKED, ELSE MENU UPDATE 3 %Temp UNCHECKED
RETURN
```

If there are less than two active accounts, View_Check simply returns to the Menu routine, because account names are only listed on the Accounts menu if more than one account has been established. Otherwise, it checks the current value of the variable %Account, which keeps track of the active account. Accounts are numbered 0 through 9. If %Account is equal to 0, View_Check places a check mark next to the first account listed on the menu, using the command MENU UPDATE 3 1, which tells DynaComm to

update the first item on the third pop-up menu. Otherwise, it removes the check mark from that item if one is already there.

Next, `View_Check` repeats this process eight more times, by calling the `View_Check_2` routine, passing it the value of `%Temp`, and then incrementing the value of `%Temp` after `View_Check_2` has returned, until `%Temp` is no longer less than 10. The increment command increases `%Temp`'s value by 1 each time it is executed.

The Pick Routine

The next routine, `Pick`, is called when the user selects a new account from those listed on the `Accounts` menu.

```
*Pick
IF %Choice = %Account RETURN
SET %Account %Choice
SET POINTER WATCH
MENU
MENU END
SET %MENU 1
PERFORM File_It
PERFORM Saveall
PERFORM Get_Lists
SET POINTER ARROW
RETURN
```

The routine begins by comparing the account selected by the user (`%Choice`) with the active account (`%Account`). If they are the same, it simply returns to the `Menu` routine. Otherwise, it sets the mouse pointer to an hourglass shape, cancels the menu, and sets the variable `%Menu` to a value of 1. (`%Menu` is a flag used by the `File_It` routine.) Then it performs the `File_It` routine, the `Saveall` routine, and the `Get_Lists` routine, in the process loading the eight mail folders for the newly selected account into memory, as well as the names of the user-defined mail folders and the account name and password for the newly selected account. Then it sets the mouse pointer back to an arrow shape and returns to the `Menu` routine.

The Pub_Book Routine

The next routine, called `Pub_Book`, is called by the `Public` phonebook location item on the `M.M.M.` menu.

```
*Pub_Book
DIALOG (.,160,) "Public Phonebook Location" modal
MESSAGE (8,8,,) "Enter drive and path for public phonebook:"
EDITTEXT (8,24,,) 130 "" $Pubpath
NEWLINE
BUTTON (40,,) DEFAULT "OK" RESUME
```

```

BUTTON CANCEL "Cancel" DIALOG CANCEL, RETURN
DIALOG END
WAIT RESUME
SET $Pubpath EDITTEXT(1)
DIALOG CANCEL
%Pp=LENGTH($Pubpath)
IF SUBSTR($Pubpath,%Pp,1)="\" SET $Pubpath SUBSTR($Pubpath,1,%Pp-1)
RETURN

```

Pub_Book simply opens a small dialog box that prompts the user to enter the location of the Public address book. The dialog box includes both OK and Cancel buttons. The Cancel button closes the dialog box and returns to the menu screen when it is selected, whereas the OK button issues a RESUME command, which results in the variable \$Pubpath being assigned the value of the dialog box's edit text field. The routine uses a combination of the LENGTH and SUBSTR commands to determine if the last character in the \$Pubpath variable is a \. If so, it strips off the last character before returning to the Menu routine.

The Access_Num Routine

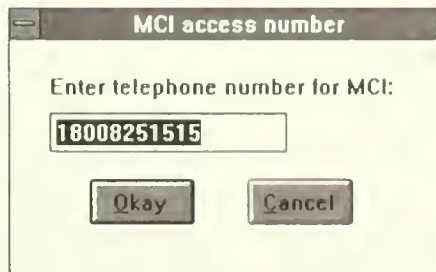
The Access_Num routine is called when the user selects the MCI phone number item on the Communications menu:

```

*Access_Num
DIALOG (..130,) "MCI access number" MODAL
MESSAGE (12,10..) "Enter telephone number for MCI: "
EDITTEXT (12,22..) 72 "" $Num LIMIT 17
BUTTON (24,..) DEFAULT "OK" SET $Num EDITTEXT(1). SET PHONENUMBER $Num. RESUME
BUTTON CANCEL "Cancel" RESUME
DIALOG END
WAIT RESUME
DIALOG CANCEL
RETURN

```

This routine simply opens a dialog box, shown below, that prompts the user to enter the telephone number it is to use to access MCI Mail.



When the user presses the OK button, the telephone number that has been entered is assigned both to DynaComm's PHONENUMBER global

variable (for use by the Terminal and Send/Recv routines) and to the variable \$Num for use by the AutoMCI routine. The RETURN following the DIALOG CANCEL statement returns to the Menu routine.

The Auto_Set Routine

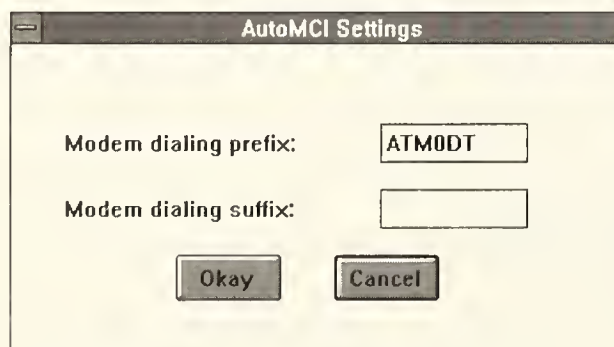
The following routine, Auto_Set, is used to obtain additional information needed by the script for use in the AutoMCI routine.

```
*Auto_Set
DIALOG (.,186,) "AutoMCI Settings" MODAL
EDITTEXT 45 "Modem dialing prefix: " $Pre_Dial
EDITTEXT 45 "Modem dialing suffix: " $Post_Dial
BUTTON (50,,) DEFAULT "OK" RESUME
BUTTON CANCEL "Cancel" DIALOG CANCEL, RETURN
DIALOG END
WAIT RESUME
SET $Pre_Dial EDITTEXT(1)
SET $Post_Dial EDITTEXT(2)
DIALOG CANCEL
RETURN
```

Auto_Set creates a dialog box, shown in Figure 15.10, to obtain the modem initialization and termination strings that are to be sent, respectively, prior to and immediately following the MCI-access telephone number during the AutoMCI dialing procedure. This information is needed because AutoMCI uses a custom dialing routine instead of the standard DynaComm dialing routine.

Figure 15.10

The AutoMCI
Settings dialog box



The reason for this is simple. The standard DynaComm dialing routine opens a small dialog box on screen that can't be closed until a connection is made. This is fine if you're operating DynaComm in the foreground, as you do with the Send/Recv or Terminal methods of connecting to MCI. But the

AutoMCI method is designed to dial into MCI in the *background*. If the DynaComm dialing routine were used whenever AutoMCI started to dial the modem, the Dialing dialog box would pop up in the middle of whatever other application you happened to be using, and keep you from doing anything else with that application until a connection was made.

The custom dialing routine used by AutoMCI is much less intrusive. However, because DynaComm's language doesn't provide any way to access the modem setup information gathered by its standard modem settings routine, AutoMCI needs the information gathered by the Auto_Set routine in order to send the proper modem-initialization string.

The Restore Routine

The Restore routine is called when the user selects the Restore backups item on the File menu.

```
*Restore
SET DIRECTORY DATA $Data_Dir | $Mailbox
DIALOG (.,148,144) "Restore Backups" modal
MESSAGE (8,20,.) "                Restore backup for:"
GROUPBOX (8,32,130,72) "Mailslots"
CHECKBOX (16,48,.) $Ms1
CHECKBOX (16,60,.) $Ms2
CHECKBOX (16,72,.) $Ms3
CHECKBOX (16,84,.) $Ms4
CHECKBOX (88,48,.) $Ms5
CHECKBOX (88,60,.) $Ms6
CHECKBOX (88,72,.) $Ms7
CHECKBOX (88,84,.) $Ms8
BUTTON (24,112,.) DEFAULT "OK" RESUME
BUTTON CANCEL "&CANCEL" DIALOG CANCEL, RETURN
DIALOG END
WAIT RESUME
PERFORM Restore_Checks (1,"INBOX")
PERFORM Restore_Checks (2,"SENT")
PERFORM Restore_Checks (3,"OUTBOX")
PERFORM Restore_Checks (4,"DRAFTS")
PERFORM Restore_Checks (5,"UNSENT")
PERFORM Restore_Checks (6,"UD1")
PERFORM Restore_Checks (7,"UD2")
PERFORM Restore_Checks (8,"UD3")
DIALOG CANCEL
RETURN
```

The Restore routine begins by opening a dialog box that contains eight check boxes, one for each of the eight mail folders, as shown in Figure 15.11. The user checks all those that should be restored from the backup file.

Figure 15.11

The Restore Backups dialog box



Once the user presses the OK button, the dialog box closes, and Restore performs the Restore_Checks routine once for each mail folder, passing the routine the check box number for the mail folder and its file name.

If a mail folder's check box is not checked, the Restore_Checks routine returns immediately to the Restore routine. Otherwise, it copies the mail folder's .BAK file over its mail folder file, performs first the Close_and_Clear routine to close the table representing the mail folder (thus saving the data from the .BAK file to disk under the mail folder's file name), and then performs the Table_Load routine to reload the table into memory before returning to the Restore routine.

```
*Restore_Checks (%Box, $Restore_File)
IF CHECKBOX(%Box)<> 1 RETURN
FILE COPY $Restore_File | ".BAK" $Restore_File
PERFORM Close_and_Clear(%Box, 1)
SET %ErrCount 0
PERFORM Table_Load (%Box, $Restore_File)
RETURN
```

The %ErrCount variable that the routine initializes immediately before performing the Table_Load routine is used by the Table_Load routine for error-checking purposes, as will be explained later.

The About Routine

The next routine, About, which is called whenever the user selects the About M.M.M.... item on the File menu, displays author and copyright information for the M.M.M. script in a modal dialog box, as shown in Figure 15.12.

Figure 15.12

The About M.M.M. dialog box



```
*About
DIALOG (.,184,114) "About M.M.M." MODAL
PICTURE (16,8) $Data_Dir | "MMM2.BMP"
MESSAGE (44,50,.) "Written by Paul Bonner."
MESSAGE (10,62,.) "Version 3.1, \251 1990, 1991 by Paul
    Bonner."
BUTTON (64,94,.) DEFAULT "OK", DIALOG CANCEL, RETURN
DIALOG END
WAIT RESUME
```

The Auto_Freq Routine

The Auto_Freq routine is used to determine how frequently the user wants the AutoMCI routine to dial into MCI Mail.

```
*Auto_Freq
DIALOG (.,152,100) "AutoMCI Frequency" MODAL
MESSAGE "Enter interval for AutoMCI"
```



```

MESSAGE "Connections (5-600 minutes)."  

NEWLINE  

EDITTEXT 16 "Frequency in minutes: " STR(%Delay) LIMIT 3  

NEWLINE  

BUTTON DEFAULT "OK" RESUME  

BUTTON CANCEL "Cancel" DIALOG CANCEL, RETURN  

DIALOG END  

WAIT RESUME  

SET $Temp EDITTEXT()  

SET %Delay NUM($Temp)  

IF %Delay < 5 SET %Delay 5  

IF %Delay > 600 SET %Delay 600  

PERFORM Saveall  

DIALOG CANCEL  

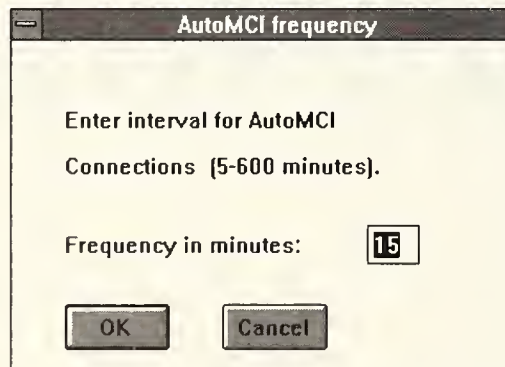
RETURN

```

The AutoMCI Frequency dialog box, shown in Figure 15.13, accepts values ranging between 5 and 600 minutes (ten hours). If the user enters a value of less than 5, the delay is set to 5 minutes. Values greater than 600 are adjusted down to 600. Once the value for the AutoMCI interval has been established, the Saveall routine is called to save it to disk.

Figure 15.13

The AutoMCI
Frequency dialog
box



The Slots Routine

The final menu-support routine, Slots, is used to name the three user-definable mail folders.

```

*Slots
SET DIRECTORY DATA $Data_Dir | $Mailbox
DIALOG (, ,120,132) "Mailslot names" MODAL
MESSAGE (16,12,,) "Mailslot 1:      " | $Ms1
MESSAGE (16,24,,) "Mailslot 2:      " | $Ms2
MESSAGE (16,36,,) "Mailslot 3:      " | $Ms3

```

```

MESSAGE (16,48,,) "Mailslot 4:      " | $Ms4
MESSAGE (16,60,,) "Mailslot 5:      " | $Ms5
EDITTEXT (16,72,,) 36 "Mailslot 6: " $Ms6 LIMIT 8
EDITTEXT (16,84,,) 36 "Mailslot 7: " $Ms7 LIMIT 8
EDITTEXT (16,96,,) 36 "Mailslot 8: " $Ms8 LIMIT 8
BUTTON (24,,,) DEFAULT "OK" RESUME
BUTTON CANCEL "Cancel" DIALOG CANCEL, RETURN
DIALOG END
WAIT RESUME
SET $Ms6 EDITTEXT(1)
SET $Ms7 EDITTEXT(2)
SET $Ms8 EDITTEXT(3)
FILE DELETE "SLOTS"
TABLE DEFINE 9 TEXT "SLOTS"
PERFORM WriteString(9, $Ms6)
PERFORM WriteString(9, $Ms7)
PERFORM WriteString(9, $Ms8)
PERFORM Close_and_Clear (9,0)
DIALOG CANCEL
SET DIRECTORY DATA $Data_Dir
RETURN

```

Slots consists of a simple dialog box, shown in Figure 15.14, that lists the names of the five standard mail folders as static text and the names of the three user-definable ones in editable text fields. Once the user closes the dialog box with the OK button, the existing SLOTS data file is deleted and a new one is created. The WriteString routine is used to write the new names of the three user-definable mail folders to the new SLOTS file.

Figure 15.14

M.M.M.'s Mailslot
Names dialog box

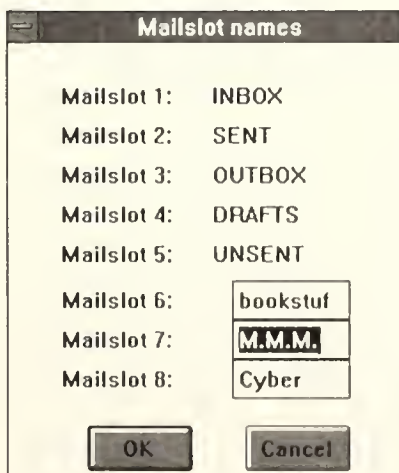


Table-Handling Routines

The next set of routines is used to define, open, close, clear, back up, and delete the structured tables that M.M.M. uses to store its mail folders.

Each mail folder is represented by a DynaComm structured table consisting of the following nine fields:

- An 81-character field used to store the incoming/outgoing indicator and the abridged name of the sender or addressee, an abbreviated subject description, the file size, and date information about each message that is displayed in the central list box of the Mailboxes screen
- A 28-character field used to store the message's subject
- A 6-character field for its file size
- A 10-character field that is reserved for future use
- A 10-character field used to store the date the message was sent or received
- A 15-character field used to hold the sender's or addressee's MCI account number
- A 1-character field used to indicate whether the message is a reply
- An 8-character field used for the name (without extension) of the file in which the message is stored
- A 45-character field used to hold the full name of the sender or addressee

The Tables Routine

The Tables routine is called during the M.M.M. start-up routine.

```
*Tables
SET DIRECTORY MEMO $Data_Dir | $Mailbox
SET DIRECTORY DATA $Data_Dir | $Mailbox
PERFORM Table_Def_And_Load (0, "INBOX")
PERFORM Table_Def_And_Load (1, "SENT")
PERFORM Table_Def_And_Load (2, "OUTBOX")
PERFORM Table_Def_And_Load (3, "DRAFTS")
PERFORM Table_Def_And_Load (4, "UNSENT")
PERFORM Table_Def_And_Load (5, "UD1")
PERFORM Table_Def_And_Load (6, "UD2")
PERFORM Table_Def_And_Load (7, "UD3")
SET DIRECTORY DATA $Data_Dir
RETURN
```


Tables resets the DynaComm pointers for its Data and Memo directories to the directory indicated by the concatenation of \$DataDir | \$Mailbox (which ends up pointing to a directory such as C:\WINDOWS\DYNACOMM\DATA\MB1), then performs the Table_Def_And_Load routine for each mail folder before resetting the data directory to \$Data_\$Dir.

The Table_Def_And_Load Routine

The Table_Def_And_Load routine accepts the number and name of the table to be defined and loaded into memory as parameters.

```
*Table_Def_And_Load (%Counter, $Tabfile)
SET %ErrCount 0
TABLE DEFINE %Counter FIELDS CHAR 81 CHAR 28 CHAR 6 CHAR 10 CHAR 10
    CHAR 15 CHAR 1 CHAR 8 CHAR 45 FILE
PERFORM Table_Load (%Counter, $Tabfile)
RETURN
```

The routine begins by resetting the %ErrCount variable to 0, and then uses the DynaComm TABLE DEFINE command to define the table specified by the variable %Counter. The TABLE DEFINE command specifies the size of each field in the table. The FILE command at the end of the definition statement instructs DynaComm to store the table's data on disk rather than in global memory.

Next, the routine calls the Table_Load procedure, passing the %Counter and \$Tabfile variables on to it.

```
*Table_Load (%Counter, $Tabfile)
IF %ErrCount > 5 CANCEL
TABLE LOAD %Counter FROM $Tabfile AS DYNACOMM
IF ERROR() INCREMENT %ErrCount, GOTO Table_Load
RETURN
```

The Table_Load procedure issues the command to load the table specified by %Counter from the file specified by \$Tabfile in DynaComm format. (DynaComm format is a structured record format supported by the DynaComm program.) The routine includes a primitive error-checking routine: In the event of a disk error it will try again to read the file from disk, but after five retries it will give up and cancel the script.

The Close_and_Clear Routine

The next routine, Close_and_Clear, serves the opposite purpose of the Table routine.

```
*Close_and_Clear (%Close, %Redefine)
TABLE CLOSE %Close
TABLE CLEAR %Close
IF %Redefine = 1 TABLE DEFINE %Close FIELDS CHAR 81 CHAR 28 CHAR 6
```

```
CHAR 10 CHAR 10 CHAR 15 CHAR 1 CHAR 8 CHAR 45 FILE
RETURN
```

It accepts two parameters: the number of the table to close and clear from memory, and %Redefine, which indicates whether the table should immediately be redefined.

The Table_Save Routine

The next routine, Table_Save, also accepts table number and file name parameters.

```
*Table_Save (%Counter, $Tabfile)
TABLE SAVE %Counter TO $Tabfile AS DYNACOMM
IF ERROR() PERFORM FastRest (%Table)
RETURN
```

Table_Save saves the table indicated by %Counter to the file indicated by \$Tabfile in DYNACOMM format. If an error occurs during the save operation, it calls the FastRest routine, which restores the table's backup file.

The Close_Em Routine

The next routine, Close_Em, is used to close all the tables at once.

```
*Close_Em
SET %I 0
WHILE %I < 15 PERFORM Close_And_Clear (%I,0), INCREMENT %I
RETURN
```

DynaComm supports the use of up to 15 tables, numbered 0 through 14, so Close_Em starts with table 0 and calls the Close_and_Clear routine 15 times, incrementing the variable %I (used to designate the table) each time. The routine doesn't bother checking to determine whether each table is in fact defined before it tries to close and clear it because DynaComm will accept instructions to close a table that isn't open without complaint.

The Save_Table Routine

The next routine, Save_Table, is used to save the contents of the active message-folder table to the appropriate disk file.

```
*Save_Table
SET DIRECTORY DATA $Data_Dir | $mailbox
IF %Table = 0 PERFORM Table_Save (0, "INBOX")
ELSE IF %Table = 1 PERFORM Table_Save (1, "SENT")
ELSE IF %Table = 2 PERFORM Table_Save (2, "OUTBOX")
ELSE IF %Table = 3 PERFORM Table_Save (3, "DRAFTS")
ELSE IF %Table = 4 PERFORM Table_Save (4, "UNSENT")
ELSE IF %Table = 5 PERFORM Table_Save (5, "UD1")
```

```

ELSE IF %Table = 6 PERFORM Table_Save (6, "UD2")
ELSE IF %Table = 7 PERFORM Table_Save (7, "UD3")
SET DIRECTORY DATA $Data_Dir
RETURN

```

This routine uses the value of the variable %Table to identify the active message folder and to determine the file to which it is to be saved.

The Back_Up Routine

Save_Table is followed in the listing by a routine called Back_Up, which is used to save a copy of the file that contains a specified mail folder to a file with a .BAK extension.

```

*Back_Up (%Back)
IF %Back1 = 0 PERFORM Table_SAVE (0, "INBOX.BAK")
IF %Back1 = 1 PERFORM Table_SAVE (1, "OUTBOX.BAK")
IF %Back1 = 2 PERFORM Table_SAVE (2, "SENT.BAK")
IF %Back1 = 3 PERFORM Table_SAVE (3, "UNSENT.BAK")
IF %Back1 = 4 PERFORM Table_SAVE (4, "DRAFTS.BAK")
IF %Back1 = 5 PERFORM Table_SAVE (5, "UD1.BAK")
IF %Back1 = 6 PERFORM Table_SAVE (6, "UD2.BAK")
IF %Back1 = 7 PERFORM Table_SAVE (7, "UD3.BAK")
RETURN

```

The Back_Up routine accepts a single parameter—the number of the table for which the backup operation should be performed.

The Get_List Routine

The Get_Lists routine, which appears next in the listing, is called whenever the script needs to refresh the tables used to display the Sort by: and View: list boxes on the main Mailboxes screen.

```

*Get_Lists
TABLE DEFINE 10 FIELDS CHAR 8
PERFORM WriteString(10, $MS1)
PERFORM WriteString(10, $MS2)
PERFORM WriteString(10, $MS3)
PERFORM WriteString(10, $MS4)
PERFORM WriteString(10, $MS5)
PERFORM WriteString(10, $MS6)
PERFORM WriteString(10, $MS7)
PERFORM WriteString(10, $MS8)
TABLE DEFINE 11 FIELDS CHAR 8
PERFORM WriteString(11, "None")
PERFORM WriteString(11, "Name")

```



```

PERFORM WriteString(11, "Subject")
PERFORM WriteString(11, "Date")
RETURN

```

Get_Lists begins by defining table 10, which is used to hold the list of tables used by the View: combo box. It then calls the WriteString procedure eight times, to write the names of each of the eight folders to the table. Next it defines table 11, which is used by the Sort by: combo box, and writes the name of the four sort options to it, again by calling the WriteString routine.

The WriteString Routine

WriteString accepts two parameters, %TableToWrite, the table number to which data is to be written, and \$Var, the string containing the data to write. It then sets the current record variable for the designated table equal to \$Var, and issues the RECORD WRITE command.

```

*WriteString (%TableToWrite, $Var)
SET @R(%TableToWrite) $Var
RECORD WRITE %TableToWrite
RETURN

```

The ReadString Routine

The final table-handling routine, ReadString, performs the opposite function, reading a single record from the designated table.

```

*ReadString (%Table_To_Read, $Var)
Record Read %Table_To_Read
Set $Var TRIM(@R(%Table_To_Read)," "," ")
RETURN

```

Mailboxes Screen Action Routines

The next set of routines is called by command sequences associated with the buttons on the main Mailboxes screen. The first of these is the Sort routine.

The Sort Routine

The Sort routine makes use of the variable %Sort, which will be equal to whichever item was selected from the Sort by: list. If %Sort is equal to 0, then the None option was selected, and so the routine returns without sorting the active folder. Otherwise, the routine determines the field on which to sort the table that holds the folder's data (field 9 for a by-name sort, field 2 for a by-subject sort, and field 5 for a by-date sort).

```

*Sort
IF %Sort=0 RETURN
SET %Errorcount 0

```

```

*Resort
IF %Errorcount > 5 CANCEL
IF %Sort=1 %S1=9
ELSE IF %Sort=2 %S1=2
ELSE IF %Sort = 3 %S1=5
ELSE RETURN
PERFORM Back_Up (%Table)
TABLE SORT %Table %S1 ASCEND
IF ERROR() PERFORM Fastrest (%Table), INCREMENT %Errorcount, GOTO Resort
SET %Errorcount 0
RETURN

```

It then calls the `Back_Up` routine to create a copy of the current folder in case an error occurs during the `Sort` routine, and then issues the `TABLE SORT` command, instructing `DynaComm` to sort table `%Table` on field `%S1` in ascending order. If an error occurs, the `FastRest` routine is called to restore the original table, and the `SORT` command is repeated. If more than five errors occur, the script ends.

The Stats Routine

The next routine, called `Stats`, is called when the user selects the `Statistics` button on the main `Mailboxes` screen.

```

*Stats
DIALOG (, ,168, )
MESSAGE (48,8,,) "Active account: " | $Account
GROUPBOX (4,24,150,56) "Messages"
PERFORM Stat_Compose (0, $Ms1, 12, 62,40)
PERFORM Stat_Compose (1, $Ms2, 12, 62,48)
PERFORM Stat_Compose (2, $Ms3, 12, 62,56)
PERFORM Stat_Compose (3, $Ms4, 12, 62,64)
PERFORM Stat_Compose (4, $Ms5, 86,136, 40)
PERFORM Stat_Compose (5, $Ms6, 86,136, 48)
PERFORM Stat_Compose (6, $Ms7, 86,136, 56)
PERFORM Stat_Compose (7, $Ms8, 86,136, 64)
MESSAGE (12,90,,) "Disk space: " | STR(DISKSPACE())
BUTTON (120,90,20,) DEFAULT "OK", DIALOG CANCEL, RETURN
DIALOG END
WAIT RESUME

```

`Stats` creates a dialog box identifying the current account, and then issues eight calls to the `Stat_Compose` routine, which creates and displays a message about a specified mail folder. `Stats` passes `Stat_Compose` the table number and name for each folder and the coordinates to be used to display the message about it. Then, after `Stat_Compose` has displayed the status message about each of the eight folders, `Stats` displays a string identifying the

amount of disk space remaining on the default drive (which it determines using DynaComm's DISKSPACE() function), and an OK button with which the dialog box can be closed.

Stat_Compose assembles and displays a status message for the specified mail folder.

```
*Stat_Compose (%About, $Label, %H1, %H2, %V)
SET %Count 0
RECORD READ %About at 0
WHILE NOT EOF
BEGIN
RECORD READ %About
INCREMENT %Count
END
MESSAGE (%H1,%V,,) $Label | ":"
MESSAGE (%H2,%V,,) STR(%Count)
RETURN
```

Stat_Compose counts the records in the specified folder by starting at 0 and reading each record until it encounters an end-of-file (EOF) marker. Then it displays a static text message displaying the name of the current table, as identified by the variable \$Label, using the coordinates specified in %H1,% V, followed by a second message presenting the number of records in the table at coordinates %H2,% V.

The Move Routine

The next routine, Move, is called when the user selects the Move button on the Mailboxes dialog box.

```
*Move
DIALOG CANCEL
SET DIRECTORY DATA $Data_Dir | $Mailbox
IF %Table=0 $Mbx = $Ms1
ELSE IF %Table=1 $Mbx=$Ms2
ELSE IF %Table=5 $Mbx=$Ms6
ELSE IF %Table=6 $Mbx=$Ms7
ELSE IF %Table=7 $Mbx=$Ms8
DIALOG (.,150,92)
IF %Table=2 MESSAGE "Move message from OUTBOX to Drafts?", GOTO D1
IF %Table=3 or %Table=4 MESSAGE "Edit envelope and move to OUTBOX?", GOTO D1
RADIOGROUP (4,8,,) 0 " Move Message from " | $Mbx | " to:"
RADIOBUTTON (14,24,,) "81 " | $Ms6
RADIOBUTTON (14,36,,) "82 " | $Ms7
RADIOBUTTON (14,48,,) "83 " | $Ms8
^D1
NEWLINE
BUTTON (44,68,,) DEFAULT "OK" RESUME
BUTTON (84,68,,) CANCEL "Cancel" DIALOG CANCEL, RETURN
```



```

DIALOG END
WAIT RESUME
DIALOG CANCEL
IF %Table = 3 or %Table=4 PERFORM Edit_Out, RETURN
IF %Table=2 %Destination = 3,
ELSE SET %Destination RADIOGROUP()+4
IF %Destination = %Table RETURN
PERFORM Back_Up (%Table)
PERFORM Back_Up (%Destination)
WHILE NOT EOF
BEGIN
RECORD READ %Destination
END
SET @R%Destination @R%Table
SET %Errorcount 0
*Jump1
IF %Errorcount>5 CANCEL
RECORD WRITE %Destination
IF ERROR() INCREMENT %Errorcount, PERFORM FastRest (%Destination), GOTO Jump1
SET %Errorcount 0
SET %Move 1
GOTO Delete

```

The Move routine is fairly complex because the choices it offers the user as a destination for the message vary based on the folder that is active at the time the Move button is selected. If the Outbox folder is active, the only choice is to move the message to the Drafts folder. If the Unsent or Drafts folder is active, the only choice is to move the message to the Outbox. Otherwise, the routine offers the ability to move messages to any of the three user-defined folders, as shown in Figure 15.15.

The intent of these restrictions is to ensure that all messages in the user-defined folders have in fact been sent or received, that all messages in the Inbox have been received, and that all messages in Sent have been transmitted. Without these restrictions, the user would have no way to tell by looking at a message whether it was actually ever sent, or if it was merely a draft message that had found its way accidentally into another folder.

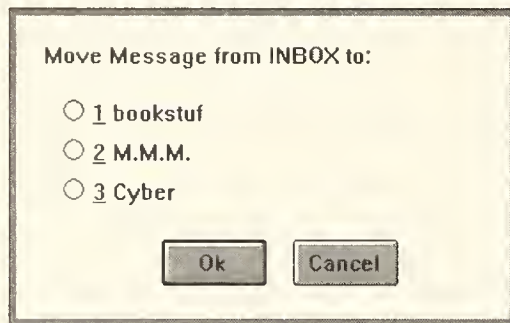
After canceling the Mailboxes dialog box and resetting the data directory, the Move routine defines the value of \$Mbx, which will be used to identify the active folder, based on the value of %Table. Then it starts to build its own dialog box. At that point, if %Table is equal to either 2 (Outbox) or 3 or 4 (Unsent or Drafts), it displays a one-line question offering the user a single possible destination and jumps ahead to the marker D1 to display the OK and Cancel buttons. Otherwise, it displays a radio group offering the user a choice of the three user-defined mail folders as possible destinations.

The routine's handling of the results of the Move dialog box is also based on the active folder. If Unsent or Drafts is active, the routine performs the Edit_Out routine, which moves the selected message into the Outbox folder and launches the TM routine to allow the user to edit the message's address,

handling options, and contents. If the Outbox folder is active, the routine sets the variable %Destination to 3, the table number of the Drafts folder. Otherwise, it sets %Destination equal to the number of the radio button that was selected, plus 4 (since the radio buttons are numbered 1 through 3 and the user-definable mail folders they represent are numbered 5 through 7).

Figure 15.15

The Move Options dialog box



Then, after checking to make sure that the destination table isn't the same as the source table, the Move routine calls the Back_Up routine twice, once for the source table and once for the destination table. It then reads every record in the destination table to set that table's record pointer to the end of the file (in order to avoid overwriting an existing record). Then it copies the current record from the source table to the destination table's record variable, and then issues a RECORD WRITE command for the destination table. Finally, it calls the Delete routine to delete the message from the source table.

The FastRest Routine

If an error occurs during the process of writing the destination table, the Move routine, like many others, calls the FastRest routine.

```
*FastRest (%Restore)
SET %ErrCount 0
SET DIRECTORY DATA $Data_Dir | $Mailbox
SWITCH %Restore
CASE 0
FILE COPY "INBOX.BAK" "INBOX", PERFORM Table_Load (0, "INBOX")
CASE 1
FILE COPY "SENT.BAK" "SENT", PERFORM Table_Load (1, "SENT")
CASE 2
FILE COPY "OUTBOX.BAK" "OUTBOX", PERFORM Table_Load (2, "OUTBOX")
CASE 3
FILE COPY "DRAFTS.BAK" "DRAFTS", PERFORM Table_Load (3, "DRAFTS")
CASE 4
FILE COPY "UNSLNT.BAK" "UNSENT", PERFORM Table_Load (4, "UNSENT")
```

```

CASE 5
FILE COPY "UD1.BAK" "UD1", PERFORM Table_Load (5, "UD1")
CASE 6
FILE COPY "UD2.BAK" "UD2", PERFORM Table_Load (6, "UD2")
CASE 7
FILE COPY "UD3.BAK" "UD3", PERFORM Table_Load (7, "UD3")
SWITCH END
RETURN

```

FastRest uses DynaComm's SWITCH CASE procedure to act on the value of %Restore (the table to restore). It copies the specified table's .BAK file to its data file, and then returns.

The Edit_Out Routine

As described above, the Edit_Out routine is used to move messages from the Drafts or Unsent folder to the Outbox folder.

```

*Edit_Out
SET %Destination 2
PERFORM Back_Up (%Table)
PERFORM Back_Up (%Destination)
SET %Move 1
SET %Answer 2
PERFORM Get_Filename
SET $F @R9.8
SET $F FILTER($f," ".)
SET DIRECTORY DATA $Data_Dir | $Mailbox
FILE DELETE "MOVEFILE.DCM"
FILE DELETE "MOVEFILE.ENV"
FILE RENAME $F | ".DCM" "MOVEFILE.DCM"
FILE RENAME $F | ".ENV" "MOVEFILE.ENV"
PERFORM Close_and_Clear (9,0)
PERFORM Delete
PERFORM Saveall
PERFORM "TM*TM" ($Data_Dir,$Mailbox,%Forward,%Answer, $Pubpath), PERFORM
  Get_Lists, RETURN

```

Edit_Out begins by backing up both the source and destination (Outbox) tables. Then it sets the values of the %Move and %Answer variables, which are used by the Delete routine, and calls the Get_Filename routine to identify the name of the file in which the selected message's contents are stored. Get_Filename reads the selected message's data into table 9, so that the name of the file is in @R9.8 (field 8 of the record variable for table 9).

Next, Edit_Out sets the variable \$F equal to the contents of @R9.8, and then uses DynaComm's FILTER command to remove any spaces from \$F, since spaces are not allowed in a file name. Then it instructs DynaComm to delete the files MOVEFILE.DCM and MOVEFILE.ENV, and then renames the .DCM and .ENV files for the selected message to MOVEFILE.DCM and MOVEFILE.ENV. (The .DCM file holds the message's contents, .ENV

its address information. Only messages in the Drafts, Outbox, and Unsent folders have .ENV files, since they are the only messages for which M.M.M. needs to track address information.)

Next Edit_Out calls Close_and_Clear to close table 9, calls Delete to remove the selected message's record from the active folder, calls Saveall to save all current settings, and then calls the TM routine in TM.DCT to allow the user to address and edit the message. When that routine concludes, Edit_Out finishes up by using Get_Lists to refresh its Sort by: and View: lists, and then returns to the Mailboxes screen.

The Check_Del Routine

The Check_Del routine is called by the Ver_Clr routine when the user initiates an action that could result in the active mail folder changing, and the active folder contains messages that are marked for deletion.

```
*Check_Del (%CD)
DIALOG (,,168,70) "AutoMCI" MODAL
MESSAGE (4,8,,) "Okay to purge all messages marked for deletion?"
BUTTON (14,28,,11) DEFAULT "OK" SET %CD 1, RESUME
BUTTON (,,11) "&No" SET %CD 2, RESUME
BUTTON (,,11) CANCEL "Cancel" SET %CD0, RESUME
MESSAGE (4,50,,) "Select 'OK' to proceed, 'No' to clear all marks,"
MESSAGE (4,58,,) "or 'Cancel' to return to the mailbox screen."
DIALOG END
WAIT RESUME
DIALOG CANCEL
IF %CD = 0 RETURN
SET POINTER WATCH
IF %CD = 2 PERFORM Clear_Marks, SET %MarkDel 0, RETURN
PERFORM Purge_Marked, SET %MarkDel 0, RETURN
SET POINTER ARROW
RETURN
```

The routine creates a dialog box, shown in Figure 15.16, that informs the user that some of the messages in the active folder are marked for deletion and asks if they should be deleted. The user has three options: to click the OK button, signaling that the messages should be deleted; to click the No button, to clear the deletion marks; or to press the Cancel button, canceling the operation that called Ver_Clr. If the choice is Cancel, the routine returns a value of 0 to Ver_Clr. Otherwise, it performs the Clear_Marks routine if the user's choice is No, or the Purge_Marked routine if the choice is the OK button.

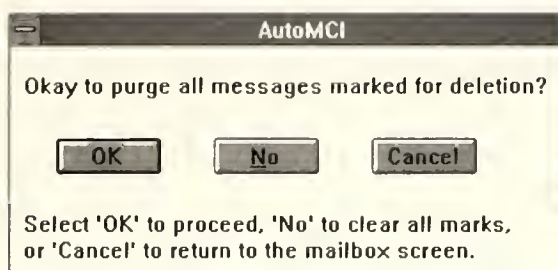
The Clear_Marks Routine

The Clear_Marks routine removes deletion marks from all messages in the active folder.

```
*Clear_Marks
```

Figure 15.16

The Purge
Messages dialog
box



```

SET DIRECTORY DATA $Data_Dir | $mailbox
SET %Counter 0
RECDRD READ %Table AT %Counter
SET PDINTER WATCH
WHILE NOT EDF
BEGIN
RECDRD READ %Table AT %Counter
IF EDF LEAVE
IF SUBSTR(@R(%Table).1.1.1)="D" SET @R(%Table).1 " " | SUBSTR(@R(%Table).1.2.80)
RECORD WRITE %Table AT %Counter
INCREMENT %Counter
END
SET DIRECTDRY DATA $Data_Dir
SET PDINTER ARRDW
RETURN

```

Messages are marked for deletion by replacing the first character in the first field of their table record with a D (normally the first character is an empty space). Thus the Clear_Marks routine reads each record in the active table and checks the first character in the first field of each record. If that character is a D, the routine replaces it with a space and writes the changed record to disk.

The Mark_Del Routine

The next routine performs the opposite function, marking the selected message for deletion when the user selects the Mark Del. button.

```

*Mark_Del (%Table, %I)
IF %I < 0 RETURN
RECORD READ %Table AT %I
IF SUBSTR(@R(%Table).1.1.1)="D" SET @R(%Table).1 " " |
SUBSTR(@R(%Table).1.2.80), RECORD WRITE %Table AT %I, DECREMENT
%MarkDel,
ELSE SET @R(%Table).1 "D" | SUBSTR(@R(%Table).1.2.80), RECORD WRITE
%Table AT %I, INCREMENT %MarkDel
RETURN

```

If the selected message is already marked for deletion, the Mark_Del routine removes the deletion mark.

The Purge_Marked Routine

The next routine, `Purge_Marked`, is used to remove all records that have been marked for deletion from the active folder.

```
*Purge_Marked
SET DIRECTORY DATA $Data_Dir | $mailbox
IF %MarkDel < 1 RETURN
SET %Counter 0
RECORD READ %Table AT %Counter
WHILE NOT EOF
BEGIN
SET POINTER WATCH
RECORD READ %Table AT %Counter
IF SUBSTR(@R(%Table).1,1,1)="D" PERFORM Del_One
INCREMENT %Counter
END
PERFORM Back_Up (%Table)
TABLE DEFINE 9 FIELDS CHAR 81 CHAR 28 CHAR 6 CHAR 10 CHAR 10 CHAR 15 CHAR
    1 CHAR 8 CHAR 45 FILE
TABLE COPY %Table TO 9
PERFORM Close_and_Clear (%Table, 1)
TABLE COPY 9 TO %Table EXCLUDE "D"
TABLE CLOSE 9
SET POINTER ARROW
PERFORM Save_Table
RETURN
```

`Purge_Marked` reads each record in the table that represents the active folder. If a record is marked for deletion, `Purge_Marked` calls the `Del_One` routine to delete the `.DCM` and `.ENV` files associated with that record, and then proceeds to read and examine the next record. When all messages have been read and examined, it backs up the current table. Then it defines table 9 using the same field structure as the eight message folder tables and copies the current table to it using DynaComm's `TABLE COPY` command.

Then it copies table 9 back to the active table using the DynaComm `EXCLUDE` command to leave out those messages that begin with the letter `D` (the messages that are marked for deletion). Finally, it closes table 9 and saves the active table, from which the marked messages have been excised, before returning to the Mailboxes dialog box.

The `Del_One` routine is called by the `Purge_Marked` routine to delete the files associated with a record that has been marked for deletion.

```
*Del_One
SET $F @R(%Table).8
SET $F1 TRIM($F, ,) | ".DCM"
SET $F2 TRIM($f, ,) | ".ENV"
FILE DELETE $F1, FILE DELETE $F2
RETURN
```


Del_One sets \$F equal to the file name specified in field 8 of the current record, and then creates two new file names by concatenating that file name with the strings “.DCM” and “.ENV”. (The TRIM function removes any trailing spaces from \$F, in order to ensure that “J424” becomes “J424.DCM” after the concatenation, not “J424 .DCM”, which would be an illegal file name.) Then it issues the FILE DELETE command to delete both files in turn.

The Delete Routine

The Delete subroutine is called by the Move routine to delete, in the active folder, the record of a message that has been moved to another folder.

```
*Delete
IF %I < 0 RETURN
SET DIRECTORY DATA $Data_Dir | $Mailbox
TABLE DEFINE 9 FIELDS CHAR 81 CHAR 28 CHAR 6 CHAR 10 CHAR 10 CHAR 15
    CHAR 1 CHAR 8 CHAR 45 FILE
TABLE COPY %Table to 9
RECORD READ 9 at %I
PERFORM Back_Up (%Table)
TABLE CLEAR %Table
TABLE COPY 9 to %Table EXCLUDE @R9
TABLE CLEAR 9
PERFORM Save_Table
I%Move=0
SET DIRECTORY DATA $Data_Dir
RETURN
```

The Delete routine begins by defining 9 as a temporary table and copying the active table to it. Then it reads record %I (representing the selected record in the active table) in table 9, and copies all of table 9 except that record back to the active table. Then it clears table 9, saves the active table, and returns.

The Saveall Routine

The next routine, Saveall, is called by many routines when they need to save current script settings.

```
*Saveall
SET @S1 $Name
SET @S2 $Pass
SET @S3 $Num | "`" | $Pre_Dial | "`" | $Post_Dial
SET @S4 STR(%Delay)
SET @S6 STR(%Account) | "~" | STR(%Init) | "~" | $Pubpath
SAVE
SET POINTER ARROW
RETURN
```

The Saveall routine sets the values of the settings variables @S1, @S2, @S3, @S4, and @S6, and then issues the SAVE command, which saves the settings file to disk.

Message-Handling Routines

The next several routines are called by the message-handling command buttons on the Mailboxes screen.

The Read Routine

First among these routines is Read, which, as you might suppose, is called when you select the Read button.

```
*Read
PERFORM Get_Filename
IF @R9="" RETURN
SET $F @R9.8, SET $Subject1 @R9.2, SET $Who @R9.9
IF SUBSTR(@R9.1.1.1)="*" SET @R9.1 " " | SUBSTR(@R9.1.2.80), SET @R%Table
    @R9, RECORD WRITE %Table at %I
MENU CANCEL
MENU
POPUP "&File"
ITEM "Read &next" INCREMENT %I, WINDOW CLOSE %Win1, GOTO Read
ITEM "Save &as..." SYSTEM 0x0100 0x0112
SEPARATOR
ITEM "&Print all" SYSTEM 0x0100 0x0211, SYSTEM 0x0100 0x0243
ITEM "Print se&lection" SYSTEM 0x0100 0x0243
ITEM "Printer Set&up..." SYSTEM 0x0100 0x0122
SEPARATOR
ITEM "&Close" WINDOW CLOSE %Win1, RESUME
POPUP "&Edit"
ITEM "&Copy" SYSTEM 0x0100 0x0222
ITEM "Select &all" SYSTEM 0x0100 0x0211
POPUP "&Search"
ITEM "&Find..." PERFORM FindRoutine
ITEM "Find &next..." PERFORM FindNextJump
POPUP "&Help" SYSTEM 9
MENU END
SET MEMOTITLE TRIM($Who) | ": " | TRIM($Subject1)
SET $File FILTER($F," ",)
WINDOW OPEN MEMO $File | ".DCM" TRUE %WIN1
WHEN WINDOW %HWND %MSG %WPARAM %LPARAM
BEGIN
WINDOW DEFAULT %HWND %MSG %WPARAM %LPARAM
IF ((%HWND=%Win1) AND (%MSG=0X0002)) RESUME
END
WAIT RESUME
WHEN CANCEL WINDOW
MENU CANCEL
RETURN
```

As you can see, the Read routine is long and involved. It begins by calling the Get_Filename routine, which copies the selected record to table 9 and returns. Then the Read routine examines the record in table 9, extracting the file name, subject, and sender from it, and changing the read/unread indicator (an asterisk as the first character of the first record) to a space, indicating that the message has been read. It then copies the table 9 record back to the active table.

Next, the Read routine defines a new menu to be used for the message reader. The first pop-up menu on the new menu bar is labeled "File", and includes the items Read Next, Save as, Print all, Print selection, Printer Setup, and Close. Most of these make use of the DynaComm SYSTEM command to access internal DynaComm functions. For instance, Print all issues the command SYSTEM 0x0100 0x0211, SYSTEM 0x0100 0x0243, which translates to a Select All command, followed by a Print Selection command.

The Read Next command is an exception. It begins by incrementing %I, so that it points at the next record in the active table, then closes the File window and jumps back to the beginning of the Read command to read the next message.

The next pop-up menu, Edit, also uses DynaComm SYSTEM commands to implement standard Copy and Select All functions. The Find and Find next items, on the other hand, call script routines called FindRoutine and FindNextJump, respectively.

The third pop-up menu, Help, makes the standard DynaComm Help menu available to the user. That concludes the menu definition.

Next the Read routine sets the DynaComm MEMOTITLE variable to display the sender and subject of the selected message, and then issues the WINDOW OPEN MEMO command to open the message file. The TRUE command that appears after the file name sets the read-only status of the file to TRUE, and the last parameter in the line assigns the memo window's handle to the variable %Win1.

The WINDOW OPEN command opens the file in a memo window, allowing the user to read it and to make use of any of the items on the menu that was just created. While the user reads the file, the script goes into a loop in which it monitors the messages that Windows sends to DynaComm, looking for a window-destroy message for the memo window that was just created (such as would be issued if the user double-clicked on the window's Control menu, or selected the Close item on the File menu). When the destroy message, (%HWND=%Win1) AND (%MSG=0X0002), is received, the script exits the loop. Meanwhile, the WINDOW DEFAULT... statement immediately before the IF...RESUME statement instructs DynaComm to process all Windows messages, so the window is closed as the script exits the loop.

Finally, the Read routine cancels the WHEN WINDOW condition and the menu it created and returns to the Mailboxes screen.

The Get_Filename Routine

The next routine, Get_Filename, is used by the Read routine and others to obtain information about the selected message.

```
*Get_Filename
IF %I < 0 RETURN
TABLE DEFINE 9 FIELDS CHAR 81 CHAR 28 CHAR 6 CHAR 10 CHAR 10 CHAR 15 CHAR
    1 CHAR 8 CHAR 45 FILE
RECORD READ %Table at %I
SET @R9 @R%Table
RECORD WRITE 9
RETURN
```

All the Get_Filename routine actually does is copy the selected record to table 9 so that the routine that called Get_Filename can read and manipulate it more easily.

The FindRoutine Routine

Get_Filename is followed in the listing by FindRoutine, which is called by the Find item on the Read routine's Edit menu.

```
*FindRoutine
DIALOG (.,179,61) "Find"
MESSAGE (5,6,37,10) "&Find what:"
EDITTEXT (45,5,127,12) 127 "" $Find
CHECKBOX (3,20,95,12) %CB "&Match upper/lowercase"
GROUPBOX (3,32,116,26) ""
RADIOGROUP (4,37,113,11) %RG ""
RADIOBUTTON (6,41,50,12) "&Forward"
RADIOBUTTON (61,41,50,12) "&Backward"
BUTTON (130,23,35,12) DEFAULT "OK" RESUME
BUTTON (130,40,35,12) CANCEL "Cancel" DIALOG CANCEL, RETURN
DIALOG END
WAIT RESUME
$Find=EDITTEXT(1)
%CB=CHECKBOX(1)
%RG=RADIOGROUP(1)
DIALOG CANCEL
*FindNextJump
IF $Find= "" RETURN
IF %CB = 1 AND %RG = 2 EDIT FIND $Find CASE REVERSE,
ELSE IF %CB = 1 AND %RG=1 EDIT FIND $Find CASE,
ELSE IF %CB = 0 AND %RG=1 EDIT FIND $Find,
ELSE IF %CB = 0 AND %RG=2 EDIT FIND $Find REVERSE
RETURN
```

FindRoutine begins by essentially duplicating the Find dialog box from the Windows 3.0 version of Notepad. It offers the options for matching the case of the search string and searching forward or backward through the document. When the user clicks on the OK button, the routine issues the Dyna-Comm EDIT FIND command, adding the options CASE (case-sensitive search) and REVERSE (search backwards) as necessary.

The FindNextJump routine, which appears after the DIALOG CANCEL statement in FindRoutine, simply repeats the previous search.

The Ans_Set Routine

The next routine, Ans_Set, is called by the Ans_Button routine, discussed earlier.

```
*Ans_Set
PERFORM Get_Filename
SET $Id ""
SET $F TRIM(@R9.9)
SET $F1 @R9.2
IF @R9.6 <>" " SET $ID @R9.6
PERFORM Close_and_Clear (9,0)
PARSE $F $Ftemp1 "." $Ftemp2
IF $Ftemp2 <>" " SET $F FILTER($Ftemp2," ".) | " " | FILTER($Ftemp1. " ".)
IF $Id<> "" SET $F TRIM($F) | " /" | $Id
SET DIRECTORY DATA $Data_Dir | $Mailbox
TABLE DEFINE 9 TEXT "ANSWER"
PERFORM WriteString(9, $F)
PERFORM WriteString(9, "re: " | $F1)
TABLE CLOSE 9
SET DIRECTORY DATA $Data_Dir
RETURN
```

Ans_Set calls the Get_Filename routine, then reads the record it creates in table 9 to obtain the message file, sender, and subject matter of the selected message. It then writes that information, together with the MCI ID (\$Id) of the sender, if available, to a text file called ANSWER, before returning to the Ans_Button routine.

The For_Set Routine

The For_Set routine, called by the For_Button routine (discussed above), prepares the selected message for forwarding.

```
*For_Set
PERFORM Get_Filename
SET DIRECTORY DATA $Data_Dir | $Mailbox
SET $F @R9.8
FILE COPY FILTER($F," ".) | ".DCM" TO "FORWARD.DCM"
PERFORM Close_and_Clear (9,0)
```

```

SET DIRECTORY DATA $Data_Dir
RETURN

```

After calling `Get_Filename`, `For_Set` copies the .DCM file from the selected message to a new message called `FORWARD.DCM`, then returns to the `For_Button` routine.

The Export Routine

The `Export` routine is called by the `Export` button on the `Mailboxes` screen.

```

*Export
SET DIRECTORY DATA $Data_Dir | $Mailbox
PERFORM Get_Filename
SET $F @R9.8
PERFORM Close_and_Clear (9,0)
MENU CANCEL
MENU
SET $File FILTER($F," ",) | ".DCM"
FILE CREATENAME $File1 TYPE "*.TXT"
FILE COPY $File $File1
SET DIRECTORY DATA $Data_Dir
RETURN

```

`Export` uses `DynaComm`'s `FILE CREATENAME` function, which prompts the user to supply a name for the file being created in order to create a text file. Then it copies the current message file to the new file before returning to the `Mailboxes` screen.

The Print_Mess Routine

The `Print_Mess` routine is called when the user presses the `Print` button on the `Mailboxes` screen.

```

*Print_Mess
SET DIRECTORY DATA $Data_Dir | $Mailbox
PERFORM Get_Filename
SET $F @R9.8
MENU CANCEL
MENU
SET $File FILTER($F," ",) | ".DCM"
DIALOG
MESSAGE "Printing " | $File | "..."
DIALOG END
PRINT OPEN
PRINT FILE $File
PRINT CLOSE

```



```

WAIT DELAY "2"
DIALOG CANCEL
SET DIRECTORY DATA $Data_Dir
RETURN

```

Print_Mess makes use of the DynaComm PRINT FILE command to print the selected file.

Welcome-Message Routines

The next two routines are used to present various welcome messages to the user when the AutoMCI script is run.

The Stat_Check Routine

The Stat_Check routine is called immediately before the main Mailboxes screen appears.

```

*Stat_Check
IF ICONIC() GOTO Stat_Check
SET $Stat @S5
IF $Stat = "" RETURN
PARSE $Stat $Count "&" $Stat
PARSE $Stat $Received "&" $Stat
SET $UNSENT $Stat
SET %Count NUM($Count)
SET %Received NUM($Received)
SET %Unsent NUM($Unsent)
IF %Count < 1 and %Received < 1 and %Unsent < 1 GOTO Welcome
DIALOG (..140.)
MESSAGE (40,8..) "M.M.M., version 3.1"
NEWLINE
NEWLINE
MESSAGE "Dynacomm script by Paul Bonner"
MESSAGE " \251 1990, 1991 by Paul Bonner."
NEWLINE
IF %Count > 0 MESSAGE "          " | $Count | " Message(s) sent."
IF %Received > 0 MESSAGE "          " | $Received | " Message(s)
received."
IF %Unsent > 0 MESSAGE "          " | $Unsent | " Message(s) not sent." .
Message "          Check your UNSENT folder."
NEWLINE
BUTTON (54,..) DEFAULT "OK" RESUME
DIALOG END
WAIT RESUME
DIALOG CANCEL
SET @S5 "0&0&0"
SAVE
RETURN

```

Stat_Check begins by ensuring that the DynaComm window has not been minimized. If it has been, the routine loops until the window is restored. Otherwise, it sets the variable \$Stat equal to the contents of the settings variable @S5, and then parses \$Stat and converts the results to numeric form to obtain three integer variables: %Count, %Received, and %Unsent. If the AutoMCI script is being executed following an on-line session in which at least one message was transmitted or received, at least one of these values will be greater than zero. If not, the Stat_Check routine jumps to the routine called Welcome.

If at least one message was transmitted or received or an attempt was made to transmit at least one, then Stat_Check creates a dialog box that reports the number of messages that were sent, received, and not sent, followed by an OK button. When the user selects the OK button, the dialog box closes and the routine returns to the calling routine, after setting the value of @S5 to "0&0&0" so that all three counts will start at zero the next time an on-line session takes place.

The Welcome Routine

The Welcome routine is called if no messages were sent or received in the previous on-line session, or if an entirely new M.M.M. session is starting (as opposed to AutoMCI being executed following an on-line session).

```
*Welcome
IF %Init=0 RETURN
DIALOG (.,168,130)
PICTURE (14,8) $Data_Dir | "MMM2.BMP"
MESSAGE (44,50,.) "Written by Paul Bonner."
MESSAGE (10,62,.) "Version 3.1, \251 1990, 1991 by Paul Bonner."
ICONBUTTON (109,90,.) "_INBOX" "&Mailboxes" DIALOG CANCEL, RETURN
ICONBUTTON (4,90,.) "_SENDMAIL" "Send/Rec&v" SET %Init 0, SET @S8 "-1",
    PERFORM Saveall, DIALOG CANCEL, SET DIRECTORY DATA $Data_Dir |
    $Mailbox, SCREEN SHOW, PERFORM Close_Em, EXECUTE "Email"
ICONBUTTON (58,90,.) "_HOURL4" "&AutoMCI" SET %Init 0, SET @S8 "0",
    PERFORM Saveall, DIALOG CANCEL, SCREEN SHOW, SET DIRECTORY DATA
    $Data_Dir | $Mailbox, PERFORM Close_Em, EXECUTE "EMAIL"
, DIALOG END
WAIT RESUME
```

The Welcome dialog box simply displays copyright and authorship information for M.M.M., along with three icon buttons that can be used to launch a Send/Recv session or an AutoMCI session or to go to the Mailboxes screen.

Account Setup Routines

The final set of subroutines in AUTOMCI.DCP consists of several routines that are used to set up and manage MCI accounts within M.M.M.

The Setup Routine

The first of these accounts routines is the Setup routine.

```
*Setup
SET DIRECTORY DATA $Data_Dir
TABLE DEFINE 9 FIELDS CHAR 12 CHAR 12 CHAR 99 CHAR 8 FILE
TABLE LOAD 9 FROM "ACCOUNTS" AS DYNACOMM
DIALOG (.,128,) "Account setup" MODAL
MESSAGE (12,12..) "Select Account"
LISTBOX (36,32,42,32) 9 %Account
BUTTON (8.,24,) DEFAULT "&OK" SET %Choice LISTBOX(), IF %Choice =
    %Account DIALOG CANCEL, RETURN, ELSE SET %Account %Choice, PERFORM
    File_It, PERFORM Saveall, PERFORM Get_Lists, DIALOG CANCEL, RETURN
BUTTON (.,24,) "&Edit" SET %Choice LISTBOX(), PERFORM Edit, RESUME
BUTTON (.,24,) "&New" PERFORM New, RESUME
DIALOG END
WAIT RESUME
DIALOG CANCEL
GOTO Setup
```

The Setup routine is called when the user selects the Account setup... option on the Accounts menu. It opens a dialog box that lists the current accounts, and gives you the option of selecting or editing any of those accounts or creating a new one.

Selecting an account and pressing the OK button on this dialog box is the same as selecting an account from the drop-down Accounts menu. Selecting the Edit button calls the Edit routine.

The New Routine

Selecting the New button on the Account Setup dialog box calls the following routine.

```
*New
SET $Account "Name me"
SET $name "MCI ID"
SET $pass "password"
DIALOG (.,140,) "New account" MODAL
NEWLINE
EDITTEXT (8,16..) 52 "Account Name: " $Account LIMIT 12
EDITTEXT (8,32..) 52 "Log on: " $Name LIMIT 12
EDITTEXT (8,50..) 52 "Password: " $Pass LIMIT 12 PASSWORD
BUTTON "OK" SET $Account EDITTEXT(1), SET $Name EDITTEXT(2), SET $Pass
    EDITTEXT(3), RESUME
BUTTON "Cancel" DIALOG CANCEL, RETURN
DIALOG END
WAIT RESUME
WHILE NOT EOF
BEGIN
RECORD READ 9
END
```



```

SET %I 1
*Name_Dir
SET $Newdir $Data_Dir | "MB" | STR(%I)
SET DIRECTORY MEMO CREATE $Newdir
IF ERROR() INCREMENT %I, GOTO Name_Dir
FILE COPY $Data_Dir | "BLANK.DCM" $Newdir | "\BLANK.DCM"
FILE COPY $Data_Dir | "FORWFORM.DCM" $Newdir | "\FORWFORM.DCM"
SET @R9.1 TRIM($Account, " ", " ")
SET @R9.2 TRIM($Name, " ", " ")
SET $Pass TRIM($Pass, " ", " ")
PERFORM Code
SET @R9.3 $Pass
SET @R9.4 "MB" | STR(%I)
RECORD WRITE 9
INCREMENT %A
SET %Account %A-1
IF %Account = 0 SET $Account1 $Account,
ELSE IF %Account = 1 SET $Account2 $Account,
ELSE IF %Account = 2 SET $Account3 $Account,
ELSE IF %Account = 3 SET $Account4 $Account,
ELSE IF %Account = 4 SET $Account5 $Account,
ELSE IF %Account = 5 SET $Account6 $Account,
ELSE IF %Account = 6 SET $Account7 $Account,
ELSE IF %Account = 7 SET $Account8 $Account,
ELSE IF %Account = 8 SET $Account9 $Account,
ELSE IF %Account = 9 SET $Account10 $Account
RETURN

```

The New routine is fairly straightforward, if a little long. It begins by creating a dialog box with edit fields in which you can enter a name for the new account, the account's MCI log-on name, and the account's password, as shown in Figure 15.17. These fields are all limited to 12 characters, and the PASSWORD keyword following the LIMIT 12 statement in the Password field tells DynaComm to echo anything typed into that field as a series of asterisks, rather than as the actual letters that are being typed.

Once the dialog box has been filled out and closed, the routine attempts to create a new directory in which to store the new account's messages and data files. It begins by attempting to create a directory called MB1 under the DynaComm data directory. If an error occurs in that operation, indicating that MB1 already exists, it increments the value of %I and tries again (with MB2 this time). It continues to increment %I in this manner until it finds an acceptable name. It then copies the files BLANK.DCM and FORWFORM.DCM to the new directory, uses the Code routine to encode the account's password, and then saves the account data before returning.

Figure 15.17

The New Account dialog box

The image shows a dialog box titled "New account". It contains three input fields: "Account Name:" with the text "Name mc", "Log on:" with the text "MCI ID", and "Password:" with eight asterisks. At the bottom of the dialog box are two buttons: "Okay" and "Cancel".

The Account_Data Routine

The next routine, `Account_Data`, is used to get information about the active account during the M.M.M. start-up process and whenever the user selects a different account from the Accounts menu or the Account Setup dialog box.

```
*Account_Data
SET DIRECTORY DATA $Data_Dir
SET %MENU 0
TABLE DEFINE 9 FIELDS CHAR 12 CHAR 12 CHAR 99 CHAR 8 FILE
TABLE LOAD 9 FROM $Data_Dir | "ACCOUNTS" AS DYNACOMM
RECORD READ 9
SET %AccountCounter 0
WHILE NOT EOF
BEGIN
RECORD READ 9 AT %A
SET $Record TRIM(@R9.1,,)
IF %AccountCounter = 0 SET $Account1 $Record,
ELSE IF %AccountCounter = 1 SET $Account2 $Record,
ELSE IF %AccountCounter = 2 SET $Account3 $Record,
ELSE IF %AccountCounter = 3 SET $Account4 $Record,
ELSE IF %AccountCounter = 4 SET $Account5 $Record,
ELSE IF %AccountCounter = 5 SET $Account6 $Record,
ELSE IF %AccountCounter = 6 SET $Account7 $Record,
ELSE IF %AccountCounter = 7 SET $Account8 $Record,
ELSE IF %AccountCounter = 8 SET $Account9 $Record,
ELSE IF %AccountCounter = 9 SET $Account10 $Record
INCREMENT %A
END
DECREMENT %A
```

`Account_Data` loads a file called `ACCOUNTS` into table 9 and reads through it, obtaining the names of each account that the user has set up from it.

The File_It Routine

Account_Data then performs the File_It routine, which is also called by other routines in the script, including the Pick routine.

```
*File_It
SET POINTER WATCH
IF %Menu = 1 SET DIRECTORY DATA $Data_Dir,
TABLE DEFINE 9 FIELDS CHAR 12 CHAR 12 CHAR 99 CHAR 8 FILE,
TABLE LOAD 9 FROM $Data_Dir | "ACCOUNTS" AS DYNACOMM
RECORD READ 9 at %Account
SET $Account TRIM(@R9.1,,)
SET TERMTITLE "The MCI Mail Manager: " | $Account
SET $Name @R9.2
SET $Pass @R9.3
SET $Mailbox @R9.4
PERFORM Close_and_Clear (9,0)
SET @S1 $Name
SET @S2 $Pass
SET DIRECTORY MEMO $Data_Dir | $Mailbox
SET DIRECTORY DATA $Data_Dir | $Mailbox
SET %I 0
TABLE DEFINE 9 TEXT "SLOTS"
PERFORM ReadString(9, $MS6)
PERFORM ReadString(9, $MS7)
PERFORM ReadString(9, $MS8)
PERFORM Close_and_Clear (9,0)
PERFORM Tables
RETURN
```

File_It reopens the ACCOUNTS file to obtain the user name, password, account name, and mailbox directory (the MB *number* directory in which the account's messages and data files are stored) of the active account. It then reads the SLOTS file from the active account's mailbox directory to obtain the names of the active account's three user-definable mail folders.

The Edit Routine

The Edit routine is called when the user selects the Edit option in the Account Setup dialog box.

```
*Edit
RECORD READ 9 at %Choice
SET $Account @R9.1
IF SUBSTR($Account,1,1)=" " or SUBSTR($Account,1,1)="" RETURN
SET $Name @R9.2
SET $Pass @R9.3
PERFORM Decode
```



```

SET $Mailbox @R9.4
DIALOG (.,180.) "Edit Account" MODAL
EDITTEXT (8,16..) 52 "Account Name: " $Account LIMIT 12
EDITTEXT (8,32..) 52 "Log on: " $Name LIMIT 12
EDITTEXT (8,50..) 52 "Password: " $Pass LIMIT 12 PASSWORD
NEWLINE
MESSAGE "Messages are stored in "
MESSAGE " " | $Data_Dir | $Mailbox
NEWLINE
BUTTON (40,..) "OK" SET $Account EDITTEXT(1), SET $Name EDITTEXT(2), SET
    $Pass EDITTEXT(3), RESUME
BUTTON "Cancel" DIALOG CANCEL, RETURN
DIALOG END
WAIT RESUME
SET @R9.1 $Account
SET @R9.2 $Name
PERFORM Code
SET @R9.3 $Pass
SET @R9.4 $Mailbox
RECORD WRITE 9 at %Choice
RETURN

```

The Edit routine creates a dialog box that allows the user to modify the account name, log-on name, and password for any account, as shown in Figure 15.18. It makes use of the Decode routine (below) to decrypt the account's password.

Figure 15.18

The Edit Account dialog box



The Code Routine

The Code routine uses the DynaComm ENCRYPT function to scramble the account password, passing it the word PASS as an encryption key.

```

*Code
SET $Pass ENCRYPT($Pass,"PASS")
SET $PL STR(LENGTH($Pass))
SET $Pass $PL | $Pass
RETURN

```

After encrypting the password, the Code routine determines its length, converts that length to a string called \$PL, and concatenates \$PL with the \$Pass string. This is necessary because of a glitch in DynaComm's handling of encrypted strings that makes them tend to grow in length and become undecryptable after they have been stored in a fixed-length record table.

The Decode Routine

The Decode routine uses the value of \$PL to circumvent the encryption problem by only attempting to decrypt that many characters of the \$Pass string.

```

*Decode
SET %PL NUM(SUBSTR($Pass,1,2))
SET $Pass SUBSTR($Pass,3,%PL)
SET $Pass DECRYPT($Pass,"PASS")
RETURN

```

The SetupMMM Routines

The final two routines in the AUTOMCI.DCP script are used to set up M.M.M. for the first time. The first is called SetUpMMM.

```

*SetupMMM
SET TERMTITLE "MMM Setup"
SET $S1 "First time set up"
SET $S2 "Before using MMM for the first time"
SET $S3 "you need to set some basic parameters."
PERFORM SetupMMM_Dialog
SETTINGS COMMUNICATIONS
SETTINGS MODEM
SET $Num "1-800-825-1515"
PERFORM Access_Num
SET $S1 "Account settings"
SET $S2 "Okay, now you'll set up the MMM script"
SET $S3 "to work with your MCI account."
PERFORM SetupMMM_Dialog
TABLE DEFINE 9 FIELDS CHAR 12 CHAR 12 CHAR 99 CHAR 8 FILE
TABLE LOAD 9 FROM "ACCOUNTS" AS DYNACOMM
SET %AccountCounter 0
SET %I 0

```

```

SET %Account 0
PERFORM New
TABLE CLOSE 9
SET $S1 "Mailslots"
SET $S2 "Okay, now you'll set up the names for this"
SET $S3 "account's three user-defined mailboxes."
PERFORM SetupMMM_Dialog
SET $Mailbox "MB" | STR(%I)
SET $Ms6 "UD1", SET $Ms7 "UD2", SET $Ms8 "UD3"
PERFORM Slots
SET $S1 "Public phonebook"
SET $S2 "Last, you need to indicate where MMM"
SET $S3 "can find your Public phonebook."
PERFORM Setupmmm_Dialog
SET $Pubpath $Data_Dir
PERFORM Pub_Book
SET TERMTITLE "The MCI Mail Manager"
SET $S1 "All done"
SET $S2 "That's it, MMM is now set up for use."
SET $S3 "Press enter to run MMM."
PERFORM Setupmmm_Dialog
SET @S3 $NUM | ""
SET @S7 ""
SET %Confirm 1
SET %I 0
SET %Delay 30
SET %Init 1
SET DIRECTORY DATA $Data_Dir
PERFORM Saveall
RESTART

```

SetupMMM guides the user through the process of setting communications and modem parameters, specifying the MCI access number, creating a new account, naming the three user-definable mail folders for the new account, and indicating the location of the Public phonebook. It makes use of the same dialog boxes and routines used for these purposes in other parts of the AutoMCI script, as well as of a general-purpose message routine called SetupMMM_Dialog, which simply displays a three-line message to the user followed by an OK button. The last of these prompts is shown in Figure 15.19.

```

*SetupMMM_Dialog
DIALOG (.,150,) $S1
NEWLINE
MESSAGE (6,..) $S2
MESSAGE (6,..) $S3

```



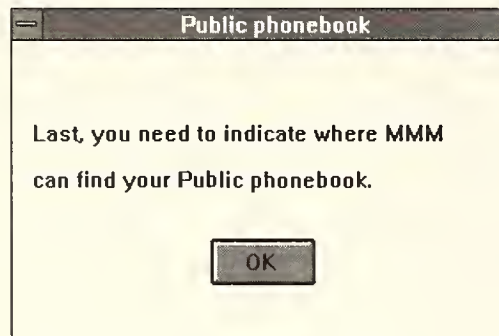
```

NEWLINE
BUTTON (60,,,) DEFAULT "OK" RESUME
DIALOG END
WAIT RESUME
DIALOG CANCEL
RETURN

```

Figure 15.19

A prompt from the SetupMMM routine



Exploring TM.DCP

The TM.DCP (To Module) module is used to address, compose, and edit outgoing messages.

Calling the TM Module

The only way to execute the TM module is to call it as an external subroutine. It expects to be passed five parameters: \$Data_Dir, \$Mailbox, %Forward, %Answer, and \$Pubpath, all of which serve the same function in this routine as they do in the AutoMCI routine.

```

*TM ($Data_Dir, $Mailbox, %Forward, %Answer, $Pubpath)
IF $Data_Dir="" EXECUTE "AUTOMCI"
SET DIRECTORY DATA $Data_Dir | $Mailbox
SET DIRECTORY MEMO $Data_Dir | $Mailbox
FILE COPY "OUTBOX" "OUTBOX.BAK"
FILE COPY "DRAFTS" "DRAFTS.BAK"
$Null = ""
%Null=Ø
SET %Messnum 1

```

The TM script begins by checking to ensure that it has been passed a parameter for \$Data_Dir. If it hasn't, it is probable that the user tried to execute TM directly, which can't be done, so it launches the AutoMCI script. Otherwise, it sets the DATA and MEMO directories to the directory specified by

the concatenation of \$Data_Dir and \$Mailbox, and makes backup copies of the Outbox and Drafts folders. Then it initializes the variables \$Null and %Null, and sets the variable %Messnum to 1.

Message-Creation Routines

The script continues by executing a series of routines that create, edit, and address messages.

The New_Mess Routine

The first of these routines is New_Mess. The New_Mess routine is performed when TM is executed for the first time and to reinitialize several variables if the user chooses to compose another message immediately after finishing the current one.

```
*New_Mess
%List=1
%Phonebook=13
$Subject=$Null
$Name=$Null
$File=$Null
$From2=$Null
%C1 = %Null
%C2 = %Null
%C3 = %Null

TABLE DEFINE 11 FIELDS CHAR 45 CHAR 45 CHAR 45 CHAR 45
TABLE DEFINE 14 FIELDS CHAR 45 CHAR 45 CHAR 45 CHAR 45
SET %Table 11
IF %Answer=1 PERFORM Answer_Message
PERFORM Get_Date
SET $Textfile $File | STR(%Messnum). SET $Textfile FILTER($Textfile." ".)
FILE DELETE $Textfile | ".ENV". FILE DELETE $Textfile | ".DCM"
SET POINTER ARROW
IF %Answer=2 FILE RENAME "MOVEFILE.DCM" $Textfile | ".DCM". PERFORM Edit_Set
TABLE DEFINE 10 TEXT $Textfile | ".ENV"
SET DIRECTORY DATA $Data_Dir
TABLE DEFINE 13 FIELDS char 45 char 45 char 45 char 45 FILE
TABLE LOAD 13 FROM $Pubpath | "\PUBLIC.PBK" AS DYNACOMM
TABLE DEFINE 8 FIELDS CHAR 45 CHAR 45 CHAR 45 CHAR 45 FILE
TABLE LOAD 8 FROM "PRIVATE.PBK" AS DYNACOMM
SET DIRECTORY DATA $Data_Dir | $Mailbox
SET %Name %Null
SET $Name1 $Null. SET $Name2 $Null. SET $Name3 $Null. SET $Name4 $Null
```

New_Mess begins by initializing a series of variables that will be used by the To_Dialog routine (below), and then defines two tables, 11 and 14, to hold the list of To: and cc: addressees for the message being composed. It then examines the value of the variable %Answer. If %Answer has a value

of 1, it calls the `Answer_Message` routine to obtain information about the message being answered.

Next the `New_Mess` routine calls the `Get_Date` routine to create a name for the new message file based on the current date and time. It then deletes any files with that name that already exist, and if `Answer%` is equal to 2 (indicating that a message is being forwarded), renames `MOVEFILE.DCM` (which was created when the user selected the Forward button) to the new message name and calls the `Edit_Set` routine.

Next it creates an envelope file for the new message, and defines tables 8 and 13 to hold the Public and Private address books. Finally, it initializes several additional string and integer variables, before proceeding on to the `To_Dialog` routine.

The To_Dialog Routine

`To_Dialog` creates the Message Addressing dialog box, shown in Figure 15.20, which is used to obtain addressing and handling information for the message being created. Its operation is fairly straightforward. Clicking either phone-book option button launches the `Change_List` subroutine, which displays the specified address book. Clicking on the `To:` and `cc:` buttons toggles back and forth between lists of the message's `To:` and `cc:` fields. Clicking the `Edit name` button launches the `Edit_Name` routine, which allows the user to change the name that is currently highlighted in the `To:` or `cc:` list. Selecting the `Cut_Name` button deletes the highlighted name. The `Edit Phonebook` button is used to launch the address book-management script.

Here is the `To_Dialog` routine:

```
*To_Dialog
DIALOG (.,268,198)
MESSAGE (8,4,.) "Edit Envelope"
EDITTEXT (100,34,.) 98 "Name: " $Name
BUTTON (238,34,25,11) DEFAULT "&Add" PERFORM Get_Name
RADIOGROUP (4,18,.) $Null PERFORM Change_Book
RADIOBUTTON (12,18,.) "&Public"
RADIOBUTTON (56,18,.) "Private"
LISTBOX (16,34,76,100) %Phonebook %Name
WIDEBUTTON (24,134,56,11) "Edit Phone&book" PERFORM Edit_Phbk, RESUME
RADIOGROUP (110,55,.) %List $Null PERFORM Change_List
RADIOBUTTON (130,55,.,11) "&To list"
RADIOBUTTON (200,55,.) "cc list"
LISTBOX (112,71,148,40) %Table
BUTTON (132,118,45,11) "&Edit name" PERFORM Edit_Name
BUTTON (196,118,45,11) "&Cut name" PERFORM Cut_Name
EDITTEXT (92,144,.) 118 "Subject: " $Subject
CHECKBOX (92,158,.) %C1 "Receipt"
CHECKBOX (92,172,.) %C2 "PRIORITY DELIVERY" CHECKBOX (92,186,.) %C3 "Mask
List Members"
ICONBUTTON (0,170,.) "_stop" "Cance&l Message" SET POINTER WATCH,FILE
DELETE $Textfile | ".ENV", SET %Table $Null, SET %I 9999, SET
```



```

        DIRECTORY DATA $Data_Dir. RETURN
    ICONBUTTON (228,170..) NOTE "&Done" PERFORM Done. RETURN
    DIALOG END
    WAIT RESUME
    SET %Phonebook RADIOGROUP(1). IF %Phonebook=1 %Phonebook=13. ELSE
        %Phonebook=8
    $Name=EDITTEXT(1)
    GOTO To_Dialog

```

Figure 15.20

The Message
Addressing dialog
box

The only thing that isn't immediately obvious about this routine is the action of the Add button, which launches the Get_Name routine. This button is identified as the default button for the dialog box, which means that it is executed whenever the user presses Enter or double-clicks a control that doesn't have its own command sequence. Thus, the Get_Name routine is executed whenever the user double-clicks on a name in the address book list or types a name into the Name field and presses Enter.

The Edit_Phbk Routine

The next routine, Edit_Phbk, is used to launch the Phon_Man address book-editing routine in PM.DCT. It begins by noting the values of the Subject field and the handling options, so that they can be restored when the Phon_Man routine is complete. Then it launches that routine.

```

*Edit_Phbk
SET $Subject EDITTEXT(2)
SET %C1 CHECKBOX(1)

```

```

SET %C2 CHECKBOX(2)
SET %C3 CHECKBOX(3)
SET %Phonebook 13
SET $Phonebook "Private"
PERFORM "PM*Phon_Man" (RADIOGROUP(1))
RETURN

```

After returning from the Edit_Phbk routine, the Script executes a RESUME command so that the To_Dialog dialog box is redrawn reflecting the handling and subject-field settings recorded by Edit_Phbk.

The Edit_Set Routine

The Edit_Set routine is used to obtain address information from messages that are being moved from the Drafts or Unsent folders to the Outbox.

```

*Edit_Set
IF NOT EXISTS "MOVEFILE.ENV" RETURN
TABLE CLEAR 9
TABLE DEFINE 9 TEXT "MOVEFILE.ENV"
*Es1
RECORD READ 9
IF @R9="To:" GOTO Es1
IF @R9="cc:" GOTO Es2
IF @R9 <> $Null SET @R11.1 @R9, PERFORM READSTRING(9, @R11.2),PERFORM
    READSTRING(9, @R11.3), PERFORM READSTRING(9, @R11.4), RECORD WRITE
    11, RECORD read 9
GOTO Es1
*Es2
RECORD READ 9
IF @R9 = "Subject:" GOTO Es3
IF @R9 <> $Null SET @R14.1 @R9, PERFORM ReadString(9, @R14.2),PERFORM
    ReadString(9, @R14.3),PERFORM READSTRING(9, @R14.4), RECORD WRITE
    14, RECORD READ 9
GOTO Es2
*Es3
RECORD READ 9
SET $Subject @R9
RECORD READ 9
SET %C1 NUM(@R9)
RECORD READ 9
SET %C2 NUM(@R9)
RECORD READ 9
SET %C3 NUM(@R9)
FILE DELETE "MOVEFILE.ENV"
RETURN

```

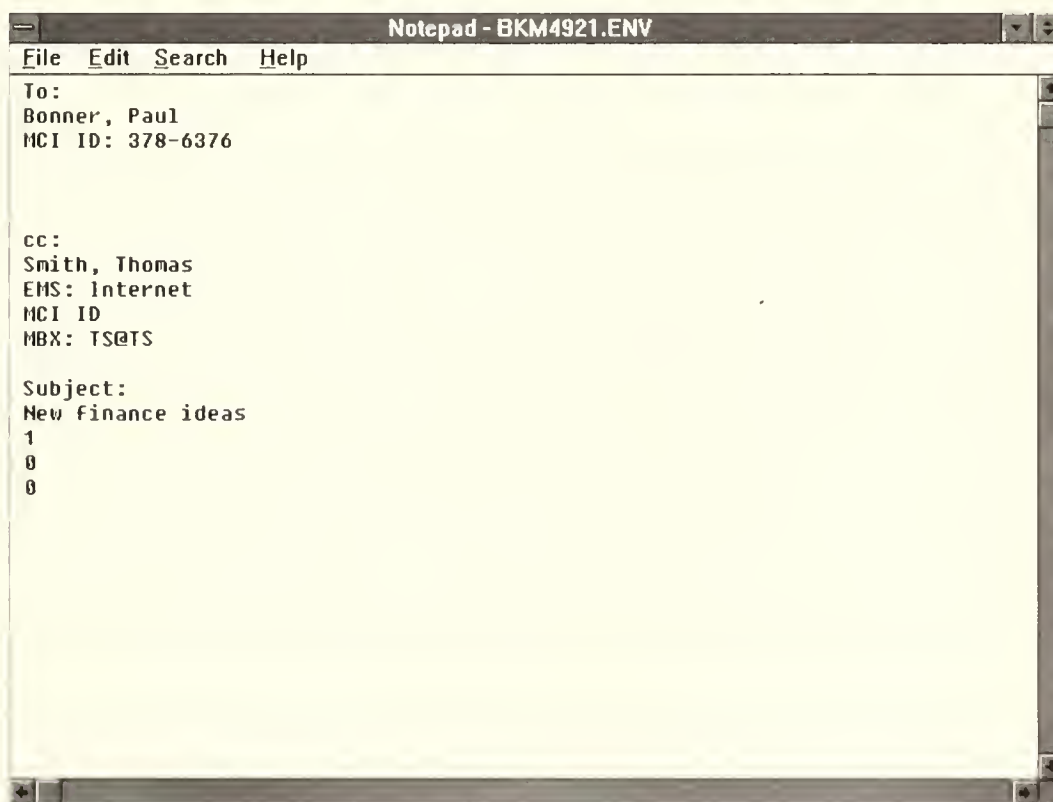
The routine begins by opening MOVEFILE.ENV. It then uses repeated RECORD READ commands to read through the file line by line. The structure of .ENV or envelope files is quite simple: The file begins with a line

containing the label “To:”, then has four lines of address information (name, MCI ID, EMS1, and EMS2) for each person on the To: list, with a blank line separating each name, then a line containing the label “cc:” and four lines of address information for each person on the cc: list, then a line containing the label “Subject:” and a line for the message’s subject, and finally three lines that contain either a 1 or a 0, reflecting the message’s handling options.

Thus, the envelope file for a message addressed to my MCI address, and copied to John Smith at an EMS address, would look something like Figure 15.21.

Figure 15.21

A sample envelope file



```
Notepad - BKM4921.ENV
File Edit Search Help
To:
Bonner, Paul
MCI ID: 378-6376

CC:
Smith, Thomas
EMS: Internet
MCI ID
MBX: TS@TS

Subject:
New finance ideas
1
0
0
```

As it reads through the file, the Edit_Set routine uses the labels “To:”, “cc:” and “Subject:” to guide its actions. Nonblank lines after the To: label are copied to the table that holds the list of To: recipients, those after the cc: label are added to the cc: list, and so on.

The Get_Date Routine

M.M.M. automatically assigns names to the message files it creates when you compose or receive mail, based on the current date and time. The Get_Date routine, and its companion routine, Shorten, are used to generate these file names.


```

*Get_Date
SET $D1 SUBSTR( DATE(), 1, 2)
SET $D2 SUBSTR( DATE(), 4, 2)
SET $D5 SUBSTR( DATE(), 8, 1)
SET $T1 SUBSTR( FILTER( TIME(), ":", $Null), 1, 2)
SET $T2 SUBSTR( FILTER( TIME(), ":", $Null), 3, 2)
SET $T3 SUBSTR( FILTER( TIME(), ":", $Null), 8, 1)
IF $T3 = "P" $T1=STR( NUM($T1)+12)
SET $T4 $T1 | ":" | $T2
PERFORM Shorten ($D1)
PERFORM Shorten ($D2)
PERFORM Shorten ($T1)
PERFORM Shorten ($T2)
SET $File $D1 | $D2 | $T1 | $T2 | $D5
RETURN

```

Get_Date begins by setting a series of three variables to hold the current month (\$D1), day (\$D2), and a single digit representing the current year (\$D3). These values are obtained from the DynaComm DATE function, which returns the current date in the form of 11/03/88. Then the variables \$T1, \$T2, and \$T3 are used to hold the hour, minute, and am/pm indicator for the current time, which is obtained from the TIME function. If the am/pm indicator says that the current time is a pm hour, the value of \$T1 is increased by 12. Then variable \$T4 is set to hold the hour and minute indicators from \$T1 and \$T2, separated by a colon.

Thus, if the TIME function returns a value of 03:05:18 pm, \$T1 will contain the string "15", \$T2 will contain the string "05", \$T3 will contain "P", and \$T4 will contain "15:05".

The Shorten Routine

Next, the Shorten routine is called four times, once each for \$D1, \$D2, \$T1, and \$T2.

```

*Shorten ($T)
IF NUM($t)<26 $T=CHR( NUM($T)+65), RETURN
IF NUM($T)=26 $T="~", RETURN
IF NUM($T)=27 $T="!", RETURN
IF NUM($T)=28 $T="$", RETURN
IF NUM($T)=29 $T="_", RETURN
IF NUM($T)=30 $T="#", RETURN
IF NUM($T)=31 $T="^", RETURN
RETURN

```

This routine is used to produce a shorter version of the current time and date, so that they can be represented in a six-character file name. Ordinarily,

it would take up to ten characters to represent the time and date in a file name: two characters for the current hour, two for the current minute, two for the month, two for the day, and two for the year.

However, the script cheats a little by representing the year with a single digit (which leads to the possibility of message names repeating after ten years), but even so, a numeric representation of the time and date would still require an additional eight digits, and I wanted to get the total down to at most six so that I could append an additional two-digit code. The code would allow up to 99 messages to be received using the same six-digit time and date stamp.

The answer was to convert each element of the time and date to a single alphanumeric character. Numbers from 0 to 25 are simply converted to their letter equivalents (0 becomes A, 1 becomes B, and so on) by using the CHR function to convert the value of the number being converted, plus 65, to a single-letter string. (The script adds 65 to the value of the number because the ANSI uppercase alphabet starts at CHR\$(65).

Numbers from 26 through 31, meanwhile, are simply assigned other characters that can legally be used in file names (~, !, \$, _, #, and ^, respectively), and numbers from 32 to 59 are not converted.

Once Shorten has been performed for each of the four strings, Get_Date creates a variable called \$Filename by concatenating the contents of \$D1, \$D2, \$T1, \$T2, and \$D5 with this command:

```
SET $File $D1 | $D2 | $T1 | $T2 | $D5
```

Thus, a file created at 3:05 pm on October 8, 1991 would be called KIPF1. And since minutes from 32 to 59 are not converted, a message at 3:42 pm on the same day would be called KIP421, still meeting the six-character limit objective.

The Get_Name Routine

The following routine, Get_Name, is performed when the user selects the Add button on the Message Addressing dialog box (Figure 16.20), created by the To_Dialog routine.

```
*Get_Name
$Name=EDITTEXT(1). IF SUBSTR($Name,1,1)<>" " AND $Name<>$Null PERFORM
    Parse_Name. PERFORM Add_To. PERFORM UPDATE. RETURN
IF LISTBOX(1)<0 RETURN
%Name=LISTBOX(1)
RECORD READ %Phonebook AT %Name
SET @R9 @R%SET $Name1 @R(%Phonebook).1
SET $Name2 @R(%Phonebook).2
SET $Name3 @R(%Phonebook).3
SET $Name4 @R(%Phonebook).4
PERFORM Add_To
PERFORM Update
RETURN
```

The Get_Name routine has to be able to add names to the To: and cc: list boxes from both the address book list box and the Name: field. It begins by examining the Name: field. If that field is not empty, and its first character is not a space (which would suggest that the field contains only spaces), the Get_Name routine calls the Parse_Name, Add_To, and Update routines, in succession, before returning.

Otherwise, Get_Name looks to the address book list box for a name to be added to the current list. If no name is selected in the address book list box (LISTBOX(1) < 0) it returns, unable to find a name to add to the To: or cc: list. Otherwise, it reads the selected name's record in the %Phonebook table and sets the four fields of name data equal to those in the %Phonebook record. Then it calls the Add_To and Update routines before returning.

The Parse_Name Routine

The Parse_Name routine is used by the Get_To routine to break names typed into the Name: field into the four fields of name data.

```
*Parse_Name
$N2=$Null
PARSE $Name $Name1 "/" $Name2
If $Name2<>$Null PARSE $Name2 $Name2 "/" $Name3, SET $Name2 TRIM($Name2,
    " ", " ")
IF $Name3<>$Null PARSE $Name3 $Name3 "/" $Name4, SET $Name3 TRIM($Name3,
    " ", " ")
SET $Test FILTER($Name,",."), IF LENGTH($Test)=0 SET $Name $Null, RETURN
PARSE $Name1 $N1 "." $N2
IF $N2 = $Null PARSE $Name1 $N1 " " $N2, SET $Name1 TRIM($N2," ", " ") |
    ", " | TRIM ($N1, " ", " ")
RETURN
```

To enter a person's name and MCI address or EMS address into the Name: field, you type all the information on one line, separating the four fields with a /. Thus, you might enter

```
John Smith / 555-5555
```

to address a message to John Smith at MCI address 555-5555. Or you would enter

```
John Smith / EMS: INTERNET / MCI ID: 376-5414 / JS@ABCXYZ.COM
```

to send a message to Mr. Smith at an Internet EMS address.

The Parse_Name routine takes what you type and breaks it up for the Name, MCI ID:, EMS1, and EMS2 fields using DynaComm's PARSE command. In addition, if the Name field does not contain a comma, it reverses the order of the addressee's name, so that John Smith becomes Smith, John in the To: or cc: list.

The Add_To Routine

The Add_To routine is used to actually add the addressee's name to the To: or cc: list.

```
*Add_To
IF SUBSTR($Name2,1,1)="M" or SUBSTR($Name2,1,1)="m" SET $To
  FILTER($Name2,"MCID: mcid",), SET $Name2 "MCI ID: " | $To
SET @R(%Table).1 $Name1
SET @R(%Table).2 $Name2
SET @R(%Table).3 $Name3
SET @R(%Table).4 $Name4
RECORD WRITE %Table
RETURN
```

The first line of the Add_To routine uses the DynaComm FILTER command to remove any or all of the characters "MCID: mcid" from \$Name2, the variable that holds the MCI ID of the addressee. It then adds the string "MCI ID:" to the beginning of \$Name2. This ensures that all MCI addresses will follow a consistent pattern, making it simpler for the EMAIL.DCT routine to process them.

Once that step is done, the routine simply assigns the variables \$Name1 through \$Name4 to the four fields in the current record of the To: or cc: list, and then writes that record to disk before returning.

The Update Routine

After the name has been added to the To: or cc: list, the Get_Name routine calls the Update routine to refresh the contents of the To_Dialog dialog box.

```
*Update
SET $Name1 $Null, SET $Name2 $Null, SET $Name3 $Null, SET $Name4 $Null
DIALOG UPDATE LISTBOX(1) TABLE %Phonebook
DIALOG UPDATE LISTBOX(2) TABLE %Table
DIALOG UPDATE EDITTEXT(1) $Null
RETURN
```

Update clears the contents of the variables \$Name1 through \$Name4, then uses the DynaComm DIALOG UPDATE command to refresh the two list boxes so that they show the current address book and To: or cc: list. It then clears the Name: field before returning.

The Edit_Name Routine

The Edit_Name routine is used to edit addressing data for the selected addressee.

```
*Edit_Name
SET %Name LISTBOX(2)
RECORD READ %Table AT %Name
```

```

IF @R%Table = $Null RETURN
TABLE DEFINE 9 FIELDS CHAR 45 CHAR 45 CHAR 45 CHAR 45
SET @R9 @R%Table
SET $Name TRIM(@R9.1) | "/" | TRIM(@R9.2)
IF @R9.3 <>$Null SET $Name $Name | "/" | TRIM(@R9.3)
IF @R9.4 <>$Null SET $Name $Name | "/" | TRIM(@R9.4)
PERFORM Close_and_Clear (9,1)
TABLE COPY %Table TO 9 EXCLUDE @R%Table
PERFORM Close_and_Clear (%Table,1)
TABLE COPY 9 TO %Table
PERFORM Close_and_Clear (9,0)
DIALOG UPDATE LISTBOX(2) TABLE %Table
DIALOG UPDATE EDITTEXT(1) $Name
RETURN

```

The routine copies the four fields of the current record to the variable \$Name, and then places \$Name into the Name edit text field after deleting the record from the To: or cc: list. This allows the user to edit the data the way it appears in the edit text field.

The Cut_Name Routine

The Cut_Name routine is used to remove a selected record from the To: or cc: list.

```

*Cut_Name
SET %Name LISTBOX(2)
RECORD READ %Table AT %Name
IF @R%Table = $Null RETURN
TABLE DEFINE 9 FIELDS CHAR 45 CHAR 45 CHAR 45 CHAR 45
TABLE COPY %Table TO 9 EXCLUDE @R%Table
PERFORM Close_and_Clear (%Table,1)
TABLE COPY 9 TO %Table
PERFORM Close_and_Clear (9,0)
DIALOG UPDATE LISTBOX(2) TABLE %Table
RETURN

```

After checking to ensure that the cursor isn't on an empty record, Cut_Name defines table 9 as a temporary table and copies all records except the highlighted record in the current table to table 9. It then closes, clears, and redefines the To: or cc: table, and copies the contents of table 9 to it. Finally it closes and clears table 9, and updates the contents of the To: or cc: list box.

The Change_List Routine

The Change_List routine is called when the user selects either the To: or cc: radio button.

```

*Change_List
%List =RADIOGROUP(2)
IF %List = 1 SET %Table 11, ELSE SET %Table 14
DIALOG UPDATE LISTBOX(2) TABLE %Table
RETURN

```

The DynaComm **RADIOGROUP** function returns an integer that identifies the selected button in the specified radio-button group. So **Change_List** assigns the value of **RADIOGROUP(2)** to the variable **%List**, and then acts, based upon **%List**'s value. If **%List** is equal to 1, **Change_List** sets the **%Table** variable to 11 (the To: list). Otherwise, **%Table** is set to point to 14 (the cc: list). Then the routine issues the **DIALOG UPDATE** command to update the list box to show the new table.

The Change_Book Routine

The **Change_Book** routine, used to toggle between the Public and Private address books, is nearly identical to **Change_List**.

```

*Change_Book
SET %Phonebook RADIOGROUP(1)
IF %Phonebook=1 %Phonebook=13, ELSE %Phonebook=8
DIALOG UPDATE LISTBOX(1) TABLE %Phonebook
RETURN

```

The Answer_Message Routine

The next routine, **Answer_Message**, is called during the script's initialization sequence if the **%Answer** flag is set to 1, which indicates that the message being composed is a reply.

```

*Answer_Message
TABLE DEFINE 9 TEXT "ANSWER"
PERFORM Readstring(9, $Name)
SET $N1 $Null
PERFORM Readstring(9, $Subject)
PERFORM Close_and_Clear (9,0)
SET $Name1 $Null
SET $Name2 $Null
SET $Name3 $Null
SET $Name4 $Null
PERFORM Parse_Name
IF $Name2 <>$Null SET $Name2 "MCI ID: " | $Name2
PERFORM Add_To
SET $Name $Null
FILE DELETE "ANSWER"
RETURN

```


The `Answer_Message` routine begins by defining the file `ANSWER`, which was created in the `AUTOMCI.DCT` script as a text file, and then reads from it the name of the person whose message is being answered and the subject of the message. It then performs the `Parse_Message` routine to break the name into its component parts and then the `Add_To` routine to add that information to the message's `To:` list. Finally it deletes the `ANSWER` file and returns.

The Done Routine

The `Done` routine is called when the user presses the `Done` button on the `Message Addressing` dialog box.

```
*Done
SET $Subject EDITTEXT(2)
RECORD READ 11 at %Null
SET $G @R11.1
IF SUBSTR($G,1,1)=" " SET %Table 11, SET %List 1, GOTO To_Dialog
DIALOG CANCEL
PERFORM Close_and_Clear (13,0)
PERFORM Close_and_Clear (8,0)
GOTO Process
```

The `Done` routine begins by setting the message subject equal to the contents of the `Subject:` text box, and then reads the first record in table 11 (the `To:` table). If the first character in that record is a null string, then the `To:` list must be empty. A message needs at least one `To:` recipient, so when this error occurs, the routine simply returns to the `Message Addressing` screen so that the user can add to the `To:` list.

The Process Routine

If the first character in the `To:` list is anything other than a null string, the `Done` routine closes and clears the address book tables, cancels the dialog box, and proceeds to the `Process` routine.

```
*Process
SET %Table 2
SET @R11.1 " "
RECORD WRITE 11
SET @R14.1 " "
RECORD WRITE 14
; ***** WRITE envelope
PERFORM WriteString (10, "To:")
SET %R %Null
; ***** WRITE To fields
RECORD READ 11 at %R
SET @R10 @R11.1
```

```
SET $First2 @R10
SET @R10 TRIM(@R10," "," ")
```

The Process routine begins by writing blank records at the end of both the To: and cc: lists (tables 11 and 14). Then it begins the process of building the .ENV file (table 10) for the message by first writing the string “To:” to it, and then reading the first field in the first record in table 11 (the To: table) to obtain the first name in the To: list. It assigns that variable to the @R10 record variable and to the \$First2 variable, which is used to store the name of the first person on the message’s To: list.

The More_Tos Routine

Then the script continues on to the More_Tos loop.

```
*More_Tos
RECORD WRITE 10
SET $From2 @R11.2
PERFORM WriteString(10,$From2)
PERFORM WriteString(10, @R11.3)
PERFORM WriteString(10, @R11.4)
PERFORM WriteString(10, $Null)
INCREMENT %R
RECORD READ 11 at %R
SET @R10 @R11.1
IF SUBSTR(@R10,1,1)<>" " GOTO More_Tos
SET %R %Null
PERFORM WriteString (10,"cc:")
```

The More_Tos loop begins by writing the contents of the @R10 record variable to disk, then sets the variable \$From2 to the contents of the second field in the current record in table 11. It then performs the WriteString routine to write that variable to record 10, followed by three additional WriteString commands, which write the contents of fields 3 and 4 (the EMS fields) of the current table 11 record, and then a blank record, to table 10. Next it increments the counter %R, which it uses to track the number of the table 11 record that is being processed, and reads the next record in table 11. If that record isn’t blank, there are more To: names to process, so the routine jumps back to the More_Tos label.

The Write_CC_Fields Routine

If there is no next record, the script writes the string “cc:” to the .ENV file and proceeds on to the Write_CC_Fields routine.

```
*Write_CC_Fields
RECORD READ 14 at %R
```

```

PERFORM WriteString(10, @R14.1)
PERFORM WriteString(10, @R14.2)
PERFORM WriteString(10, @R14.3)
PERFORM WriteString(10, @R14.4)
PERFORM WriteString(10, $Null)
INCREMENT %R
RECORD READ 14 AT %R
SET @R10 @R14.1
IF SUBSTR(@R10,1,1) <> " " GOTO Write_CC_Fields

```

The Write_CC_Fields routine works much like the More_Tos routine (above), except that it takes its data from table 14 (the cc: table) rather than from table 11. It reads a record in table 14 and writes each of its fields to the .ENV file table as a separate record. Then it reads the next record in table 14, and jumps back to the beginning of the Write_CC_Fields routine if the record isn't blank.

The Write_Subject_and_Handling Routine

If the record is blank, the script continues on to the Write_Subject_and_Handling routine.

```

*Write_Subject_and_Handling
PERFORM WriteString(10,"Subject:")
PERFORM WriteString(10,$Subject)
PERFORM WriteInteger(10,CHECKBOX(2))
PERFORM WriteInteger(10,CHECKBOX(1))
PERFORM WriteInteger(10,CHECKBOX(3))
PERFORM Close_and_Clear (11,0)
PERFORM Close_and_Clear (14,0)
PERFORM Close_and_Clear (10,0)
SET $Bodyfile $Textfile | ".DCM", SET $Bodyfile FILTER($Bodyfile," ".)
IF %Forward=0 and %Answer <> 2 FILE COPY "BLANK.DCM" TO $Bodyfile
IF %Forward=1 FILE COPY "FORWFORM.DCM" to $Bodyfile, FILE COPY
    FORWARD.DCM" TO $Bodyfile APPEND, FILE DELETE "FORWARD.DCM"
SET $Who $First2
IF SUBSTR($Subject,1,4)="re: " SET $Subject SUBSTR($Subject,5,25)

```

The Write_Subject_and_Handling routine begins by writing the string "Subject:" to record 10, and then writes the message's subject, followed by the values of the three handling check boxes (1 if the option was checked, 0 if it was unchecked). That completes the envelope file, so the routine closes and clears from memory tables 10, 11, and 14, and assigns the name of the message to be created to the variable \$Bodyfile.

DynaComm won't create a new memo file automatically under script control without user interaction, so rather than force the user to intercede in the process, the script copies an existing file, BLANK.DCM (a text file containing a single space character) to \$BodyFile. However, if the user is forwarding a message, the script first copies FORWFORM.DCM (a message

file that contains a banner that reads “Forwarded Message”) to \$Bodyfile, and then appends the file FORWARD.DCM (which contains the body of the message being forwarded) to that. Finally, it sets the variable \$Who to the value of \$First2, and trims the characters “re:” from the beginning of \$Subject if it finds them there.

The Compose_Message Routine

The script then proceeds to the Compose_Message routine.

```
*Compose_Message
MENU CANCEL
MENU
POPUP "File" System 1
POPUP "Edit" System 2
POPUP "Search" System 3
MENU END
SET MEMOTITLE TRIM($Who) | ": " | TRIM($Subject)
WAIT EDIT $Bodyfile
MENU CANCEL
MENU
MENU END
```

The Compose_Message routine instructs DynaComm to display its standard File, Edit, and Search menus, and to give the message window it is about to open a title that consists of the contents of \$Who separated from the contents of \$Subject by a colon. Then it issues the command WAIT EDIT \$BodyFile, which tells DynaComm to load \$Bodyfile into a memo window and pause script execution until that window has been closed, thus giving the user the opportunity to compose or edit the message, as shown in Figure 15.22.

The script resumes when the edit window closes, canceling the menu.

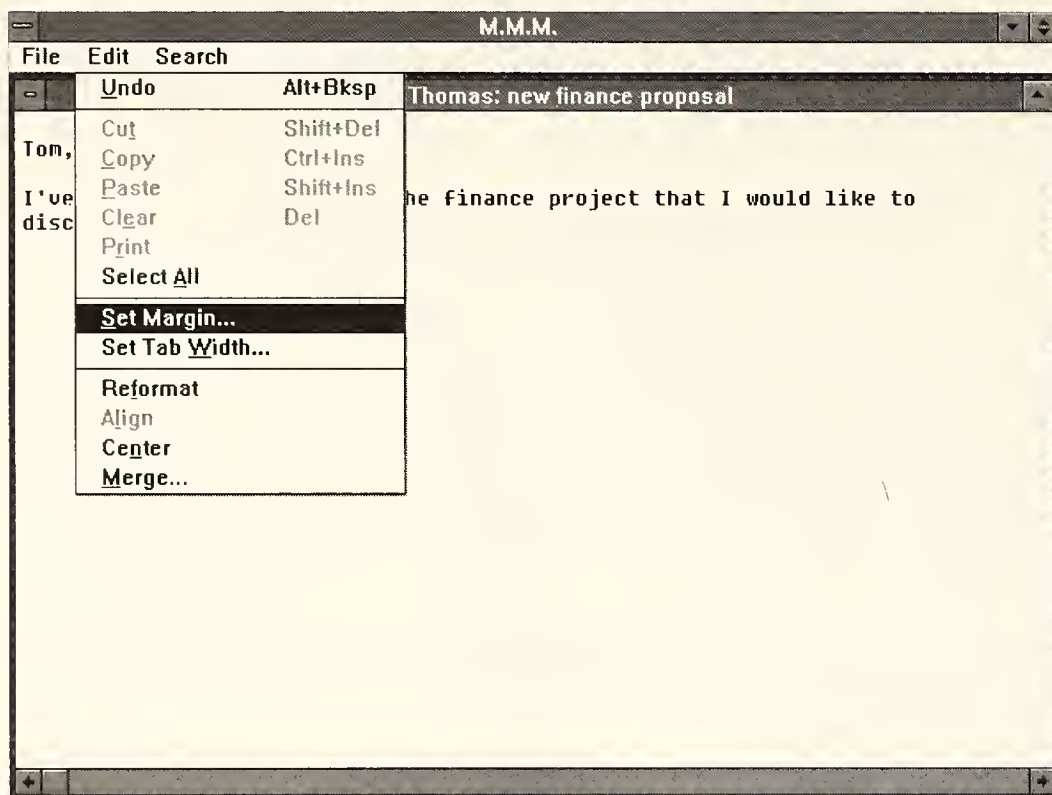
The Process_Message Routine

It then proceeds on to the Process_Message routine.

```
*Process Message
DIALOG (.,170.)
MESSAGE (4,8,.) "What would you like to do with this message?"
MESSAGE (4,32,.) "Save to OUTBOX for delivery at next connect."
MESSAGE (4,42,.) "or save to DRAFTS for more editing. "
MESSAGE (4,52,.) "or DISCARD it?"
ICONBUTTON (20,76,.) "_outbox" "Outbox" SET %Table 2, RESUME
ICONBUTTON "_change" " Drafts" SET %Table 3, RESUME
ICONBUTTON "_trash" "Discard" DIALOG cancel, FILE DELETE $textfile |
    ".ENV", FILE DELETE $Bodyfile, GOTO End It
DIALOG END
WAIT RESUME
```

Figure 15.22

The M.M.M.
message editor



```

SET $Temp_To SUBSTR($First2,1,18)
SET $Temp_Sub SUBSTR($Subject,1,25)
IF LENGTH($Temp_To) < 18 PERFORM Pad ($Temp_To, 18)
IF LENGTH($Temp_Sub) < 25 PERFORM Pad ($Temp_Sub, 25)
SET $Temp_Date DATE()
SET %Fs FILESIZE($Bodyfile)
SET $Fs STR(%Fs), IF LENGTH($Fs) < 6 PERFORM Pad ($Fs, 6)
SET @R(%Table).1 " >" | $Temp_To | " " | $Temp_Sub | " " | $Fs | " " |
    $Temp_Date | "
SET @R(%Table).2 $Subject
SET @R(%Table).3 STR(%Fs)
SET $t1 SUBSTR(FILTER(TIME(),":", $Null),1,2)
SET $t2 SUBSTR(FILTER(TIME(),":", $Null),3,2)
SET $t3 SUBSTR(FILTER(TIME(),":", $Null),8,1)
IF $T3 = "p" $T1=STR(NUM($t1)+12)
SET $T4 $T1 | ":" | $T2
SET @R(%Table).4 $T4
SET @R(%Table).5 DATE()
IF SUBSTR($From2,1,1)="M" or SUBSTR($From2,1,1)="m" SET $From2
    FILTER($From2,"MCID: mcid",)
SET @R(%Table).6 $From2
SET @R(%Table).8 $Textfile
SET @R(%Table).9 $First2
RECORD WRITE (%Table)
IF %fable = 2 TABLE SAVE 2 TO "OUTBOX" AS DYNACOMM, ELSE TABLE SAVE 3 TO

```

```

        "DRAFTS" AS DYNACOMM
    DIALOG CANCEL
    GOTO End_It

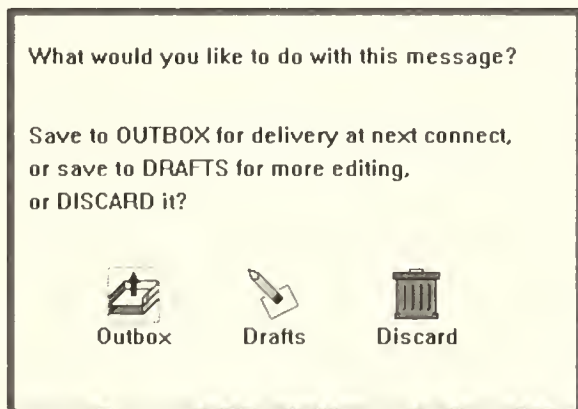
```

The Process_Message routine begins by creating a dialog box, shown in Figure 15.23, that gives the user three options for handling the message that was just created: to place it in the Outbox folder, to place it in the Drafts folder, or to discard it. If the user selects the last option the message is erased and the routine jumps ahead to the End_It routine. Otherwise, the Process_Message routine proceeds to build the record for table 2 (the Outbox folder) or table 3 (the Drafts folder) which will point to the newly created message.

It begins this process by constructing the first field in the message, which is what is displayed in the main list box of the Mailboxes screen (Figure 15.2). In order to do so, it has to trim or pad the various data items that are to be included in that field so that they'll all line up correctly in the list box. Thus, the subject of the message gets trimmed to 25 characters, the file size is padded to 6 characters, and so on.

Figure 15.23

The Process
Message dialog box



Next the routine writes the rest of the fields for the record, using the real unpadded and untrimmed values for subject, file size, sender, and so on. Finally, the routine saves the record to either the Drafts or Outbox folder before proceeding to the End_It routine.

The End_It Routine

The End_It routine is as follows:

```

*End It
SET %Answer %Null
SET %Forward %Null
DIALOG (.,160.)
MESSAGE (24,16,.) "What do you want to do now?"
ICONBUTTON (15,36,60,) "_create" "&New Message" DIALOG CANCEL, SET
    %Forward %Null, SET %Answer %Null, INCREMENT %Messnum, GOTO New Mess

```



```

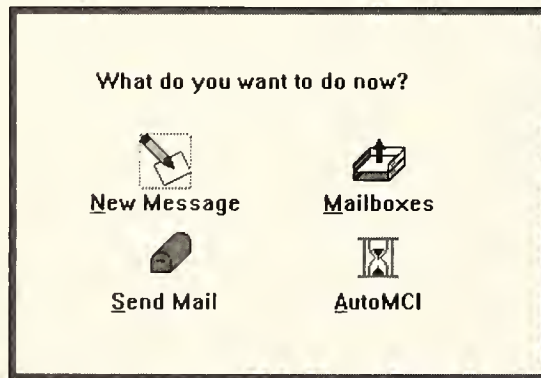
ICONBUTTON (80,36,60,) "_outbox" "&Mailboxes" RESUME
ICONBUTTON (15,66,60,) "_sendmail" "&Send Mail" SET POINTER WATCH, SCREEN
    SHOW, SET DIRECTORY DATA $Data_Dir | $Mailbox, PERFORM
    "AUTOMCI*Close_Em". EXECUTE "EMAIL"
ICONBUTTON (80,66,60,) "_hour4" "&AutoMCI" SET POINTER WATCH, SCREEN
    SHOW, SET @S8 STR(0), SET DIRECTORY DATA $Data_Dir | $Mailbox,
    PERFORM "AUTOMCI*Close_Em", EXECUTE "EMAIL"
DIALOG END
WAIT RESUME
DIALOG CANCEL
SET %TABLE 2
SET %I 9999
SET DIRECTORY DATA $Data_Dir
RETURN

```

End_It asks the user what he or she wants to do next—create another message, go to the Mailboxes screen, execute the Send/Recv routine, or launch the AutoMCI background mail service—as shown in Figure 15.24. In the first case it jumps back to the New_Mess routine at the beginning of the TM.DCP listing. In the last two cases it launches the EMAIL.DCT script, setting the appropriate flags to initiate either a Send/Recv or an AutoMCI session. Finally, if the choice is to return to the Mailboxes screen, the routine issues a RETURN command.

Figure 15.24

The End_It dialog box



Utility Routines

The final six routines in TM.DCP are general-purpose utility routines.

The Pad Routine

The first of these routines, Pad, is used to pad a string to a specified length by adding spaces to the end of the string.

```
*Pad ($Pad, %Pad_Len)
```

```
WHILE LENGTH($Pad) < %Pad_Len SET $Pad $Pad | " "
RETURN
```

The ReadInteger Routine

ReadInteger is used to read an integer variable from a specified table.

```
*ReadInteger (%Table_To_Read, %Var)
Record Read %Table_To_Read
SET %Var NUM(@R(%Table_To_Read))
RETURN
```

The WriteInteger Routine

WriteInteger writes an integer variable to a specified table.

```
*WriteInteger (%TableToWrite, %Var)
SET @R(%TableToWrite) STR(%Var)
RECORD WRITE %TableToWrite
RETURN
```

The final three utility routines, ReadString, WriteString, and Close_and_Clear, are identical to those with the same names that appear in AUTOMCI.DCP.

That concludes the listing of TM.DCP. Next you'll see EMAIL.DCP, the script that conducts the actual interaction with MCI and is responsible for sending and receiving mail during AutoMCI and Send/Recv sessions.

Exploring EMAIL.DCP

Even after all the code talked about so far, M.M.M. still hasn't spent a minute on line. That's about to change. EMAIL.DCP is the communications workhorse of M.M.M., providing full support for two of its three on-line modes: AutoMCI and Send/Recv. (Interactive Terminal-mode support is provided by the script ONLINE.DCP.)

Initializing Global Variables

EMAIL.DCP begins by making backup copies of the three tables that it manipulates (Inbox, Outbox, and Unsent), and then initializes a series of global variables.

```
*Email
IF SUBSTR(@S7,1,3)<>"MMM" EXECUTE "AUTOMCI"
CLEAR
MENU
MENU END
PERFORM BackUpFiles ("INBOX")
```

```

PERFORM BackUpFiles ("OUTBOX")
PERFORM BackUpFiles ("UNSENT")
PARSE @S5 $01d1 "&" $01d2
SET %TotalSent NUM($01d1)
PARSE $01d2 $01d1 "&" $01d2
SET %TotalUnsent NUM($01d2)
SET %Unsent 0
SET %DELAY NUM(@S4)
SET %TotalReceived NUM(@S8)
SET %Count 0
SET %Time 0
SET %Con_time 0
SET %ErrCount 0
SET $To ""
SET $Cc ""
SET $EMS ""
SET $MBX ""
%I = 1
SET %Cancel 0
SET $FileName ""
RESETSERIAL
IF %TotalReceived>-1 GOTO Auto_Retrieve

```

The @S5 settings variable will contain 0&0&0 if EMAIL.DCT was just launched. But it will contain other values for the number of messages that have been sent and not sent if the script is restarting as part of an AutoMCI session.

The value of %TotalReceived, which is obtained from the @S8 variable, determines whether a Send/Recv or an AutoMCI session is launched. When the AUTOMCI.DCT script launches EMAIL to perform a Send/Recv session, it sets @S8 to -1. In contrast, it sets the variable to 0 when it launches an AutoMCI session, and the AutoMCI routine subsequently increments the value of @S8 whenever it receives a message. Thus, if @S8 is greater than -1, the script jumps to the Auto_Retrieve routine. Otherwise, it proceeds with the Send_Recv routine.

Message-Transmission Routines

The next set of routines is executed when the user selects the Send/Recv button on the Mailboxes screen. The routines log onto MCI, send any messages in the Outbox folder, and retrieve any messages waiting in MCI for the user.

The Send_Recv Routine

The Send_Recv routine simply calls a series of subroutines in succession.

```

^Send_Recv
SET %TotalReceived 0

```



```

PERFORM Get_Date
PERFORM Login
PERFORM Loop
IF NOT CONNECT PERFORM End
PERFORM Send_Message
SET %I %Count
IF NOT CONNECT PERFORM End
PERFORM Pickup
IF NOT CONNECT PERFORM End
PERFORM Logout
*End
SET POINTER WATCH,SET @S5 STR(%TotalSent) | "&" | STR(%TotalReceived) |
    "&" | STR(%TotalUnsent), SCREEN HIDE, RESETSERIAL, EXECUTE "AUTOMCI"

```

It starts by calling the `Get_Date` routine to create a file name to be used for received messages; followed by the `Login` routine, which dials MCI and waits for a connection; the `Loop` routine, which controls the sign-on process; the `Send_Message` routine, to send any messages in the Outbox; the `Pickup` routine, to retrieve any messages waiting for the user; and finally the `Logout` routine, to disconnect from MCI. It checks after each step to make sure it is still connected to the MCI service. If not, it jumps to the `End` routine, where it stuffs variables representing the total number of messages that were sent, received, and not sent during the session into the `@S5` variable, and then executes the `AUTOMCI.DCT` script.

The `Get_Date` routine (with its companion routine, `Shorten`), functions just like the one with the same name in the `TM` routine.

The Login Routine

The `Login` routine dials the MCI access number that is stored in the settings file's `PHONENUMBER` variable, as shown here:



```

*Login
SET $N1 SETTINGS(PHONENUMBER)
DIAL $N1 RETRY 3 DELAY 45
IF NOT CONNECT() DISPLAY "No Connection^M",CLEAR, PERFORM End
RETURN

```

If after dialing three times, the `Login` routine is unable to connect to MCI, it performs the `End` routine. Otherwise, the `Send_Recv` routine calls `Loop`.

The Loop Routine

Loop is called after a connection has been established. It sends a carriage return, then waits for two seconds after establishing a pair of WHEN STRING conditions.

```
*Loop
SEND ""
WHEN STRING 0 "name:" WHEN CANCEL STRING, GOTO Prompts
WHEN STRING 1 "NO CARRIER" HANGUP, CLEAR, RETURN
INCREMENT %I
IF %I = 10 HANGUP, CLEAR, RETURN
WAIT DELAY "2"
GOTO Loop
```

A WHEN STRING condition, which remains in effect until the command WHEN CANCEL STRING is issued, tells DynaComm to monitor the data being received from the remote host and perform a specified task only when a certain string is received. In this case, the routine looks for the string "NO CARRIER", to which it will respond by hanging up the line and returning, or the string "name:", to which it will respond by canceling the WHEN STRING conditions and jumping to the Prompts routine.

Meanwhile, if the Loop routine receives neither string within 10 seconds, it issues another carriage return. If after 10 loops it still has received neither string, it hangs up the line and returns.

The Prompts Routine

The Prompts routine is called in response to MCI Mail's "Please enter your name:" prompt.

```
*Prompts
SET $Name @S1
SET $Pass @S2
PERFORM Decode
SEND $Name
WHEN STRING 0 "Password:" WAIT QUIET "1", SEND $Pass
WHEN STRING 1 "name:" WAIT QUIET "1", SEND $Name
WHEN STRING 2 "NO CARRIER" HANGUP, CLEAR, RETURN
WHEN STRING 3 "Command:" RESUME
WAIT RESUME
WHEN CANCEL STRING
RETURN
```

Prompts sets the variable \$Name to the contents of @S1, and the variable \$Pass to the contents of @S2. Then it performs the Decode routine to unscramble the password string.

The routine then creates four WHEN STRING conditions: one that looks for the string "Password:", to which it will respond by sending \$Pass; one that waits for "name:", to which it will respond by sending \$Name; one that waits for "NO CARRIER", to which it will respond by hanging up; and one that waits for "Command:" (the string that MCI sends after a successful log-on), to which the script responds by canceling the other string conditions and returning to the Send_Recv routine.

The Send_Message Routine

After a successful log-on, the Send_Recv routine calls the Send_Message routine so that it can transmit the messages in the user's Outbox.

```
*Send_Message
PERFORM Table_Def_and_Load (4, "SENT")
PERFORM Table_Def_and_Load (5, "UNSENT")
FILE DELETE DIRECTORY DATA | "HOLDFILE"
TABLE DEFINE 7 TEXT DIRECTORY DATA | "HOLDFILE"
SET %Unsent 0
TABLE DEFINE 2 FIELDS CHAR 81 CHAR 28 CHAR 6 CHAR 10 CHAR 10 CHAR 15
CHAR 1 CHAR 8 CHAR 45 FILE
TABLE LOAD 2 FROM DIRECTORY DATA | "OUTBOX": AS DYNACOMM
%Out=NUM(RESULT())
SET %COUNT 1
IF %Out=0 SET %Count 0, RETURN
```

The Send_Message routine begins by defining and loading the Sent and Unsent folders. Then it defines a temporary data file called HOLDFILE, which it uses to store data about the message that is currently being sent. Then it loads the Outbox folder into table 2, and sets %Out equal to the DynaComm RESULT function, which returns the number of records in the table. If %Out equals 0, the routine returns because there are no messages to send. Otherwise, it continues onto the Looper routine.

The Looper Routine

Looper calls the Send_Loop routine to transmit the first record in the Outbox.

```
*Looper
PERFORM Send_Loop
IF %Cancel>0 SET %Cancel 0, INCREMENT %Count
IF %Count < %Out+1 GOTO Looper
PERFORM Save_and_Close (4, "SENT")
PERFORM Save_and_Close (5, "UNSENT")
PERFORM Delete_File
RETURN
```


Then it compares the %Count variable, which tracks the number of messages that have been sent, with %Out. If %Count is less than %Out plus 1, it jumps back to the beginning of the Looper routine to send another message. Otherwise it saves and closes the Sent and Unsent folders, and calls the Delete_File routine to delete the records in the Outbox folder before returning.

The Send_Loop Routine

Send_Loop begins by reading the current record in the Outbox table to determine the names of the outgoing message's envelope and body files. Then it sends the command CREATE to launch MCI's message-composition procedure.

```
*Send_Loop
CLEAR
RECORD READ 2 at %Count-1
SET $Sendfile TRIM(@R2.8)
SET $Body DIRECTORY DATA | $Sendfile | ".DCM"
SET $Header DIRECTORY DATA | $Sendfile | ".ENV"
TABLE DEFINE 1 TEXT $Header
SEND ""
WAIT STRING "Command:"
WAIT QUIET "1"
SEND "CREATE"
WAIT STRING "TO:"
WAIT QUIET "1"
PERFORM Get_To
IF %Cancel>0 RETURN
WHEN CANCEL STRING
PERFORM Get_cc
IF %Cancel>0 RETURN
WHEN CANCEL STRING
PERFORM Get_Sub
IF %Cancel>0 RETURN
WHEN CANCEL STRING
PERFORM ReadInteger(1,%Priority)
PERFORM ReadInteger(1,%Receipt)
PERFORM ReadInteger(1,%Mask)
WAIT QUIET "1"
```

The message-creation process is an interactive one. The script sends the CREATE command, to which MCI responds by sending the string "TO:". Then the routine performs the Get_To subroutine, to read and send all the To: addressees in the message's envelope file. Then it checks the value of %Cancel. If %Cancel is greater than 0, an error occurred and the message has been moved to the Unsent folder. Otherwise, the routine proceeds to perform the

Get_cc routine to create the cc: fields on the message's address, and then reads the \$Subject and handling-options values from the envelope file.

The Send_Body Routine

The Send_Body routine, next, transmits the body of the message using Dyna-Comm's FILE SEND TEXT command.

```
*Send_Body
SEND "^M"
FILE SEND TEXT $Body NOLF
WAIT QUIET "2"
SEND "^M"
SEND "/"
WAIT QUIET "1"
```

The / that the script sends after transmitting the body of the message informs MCI that the message is complete.

The Send_Handling Routine

MCI responds to the / by requesting the message-handling options, which are transmitted by the Send_Handling routine.

```
*Send_Handling
IF %Priority > 0 AND %Receipt > 0 AND %Mask > 0 SET $Handling "4hour,
receipt, doc".
ELSE IF %Priority > 0 AND %Receipt > 0 SET $Handling "4hour, receipt".
ELSE IF %Priority > 0 AND %Mask > 0 SET $Handling "4hour, doc".
ELSE IF %Receipt > 0 AND %Mask > 0 SET $Handling "receipt, doc".
ELSE IF %Priority>0 SET $Handling "4hour".
ELSE IF %Receipt > 0 SET $Handling "receipt".
ELSE IF %Mask > 0 SET $Handling "doc".
ELSE SET $Handling ""
SEND $Handling
WAIT QUIET "1"
SEND "Y"
WAIT STRING "Command:"
WAIT QUIET "1"
SET $R "Handling: "
IF $Handling <> "" SET @R7 $R | $Handling.
ELSE SET @R7 ""
RECORD WRITE 7
PERFORM WriteSTRING(7,"")
SET @R4 @R2
SET $Target $FileName | STR(%Count)
SET @R4.8 $Target
RECORD WRITE 4
```

After composing the handling-options string and sending it to MCI, the script sends the command Y to tell MCI to send the message, then waits for

the Command: prompt to return. Then, if the handling-options string was not a null string, it concatenates it with the string "Handling:" and writes it to table 7.

Next it starts the process of moving the message from the Outbox folder to the Sent folder by setting the current record in the Sent table equal to the current record in the Outbox table, correcting the file name specified in the record, and then writing the Sent record. The process will be completed later when the records in the Outbox are deleted.

The Prepare_For_Next_Message Routine

The script then prepares to handle the next outgoing message.

```
*Prepare_For_Next_Message
INCREMENT %COUNT
INCREMENT %TotalSent
PERFORM Close_and_Clear(7,0)
SET $Target $Target | ".DCM"
FILE RENAME "HOLDFILE" $Target
FILE COPY $Body $Target APPEND
FILE DELETE $Header
FILE DELETE $Body
TABLE DEFINE 7 TEXT DIRECTORY DATA | "HOLDFILE"
RETURN
```

The Prepare_For_Next_Message routine begins by incrementing the %Count and %TotalSent variables (%Count tracks the total number of messages the script attempts to send, including those that end up in the Unsent folder, whereas %TotalSent tracks only those that were successfully transmitted). Then it closes table 7, which contains all the address and handling information for the message that was just sent, and renames it to the name specified in \$Target.

The \$Target file is the file to which the Sent record will point. The file should contain both the address information and the body of the message that was sent, so the routine appends the body of the message to \$Target. Then it deletes the old envelope and body files from the outgoing message and creates a new HOLDFILE file before returning to the calling routine.

The Get_To Routine

The Get_To routine reads and interprets the outgoing message's envelope file.

```
*Get_To
SET $J ""
SET $PREFIX "To:      "
RECORD READ 1
IF SUBSTR(@R1,1,4)="EMS:" SET %To 1, GOTO EMS
IF SUBSTR(@R1,1,6)="MCI ID" SET $To $To | " / " | @R1, RECORD READ 1
```



```

IF SUBSTR(@R1,1,6)="REMOTE" SET $To $To | " / " | @R1, RECORD READ 1
IF $To <>" " PERFORM Send_To ($To, "TO:"), WAIT QUIET "1", IF
%Cancel>0 RETURN
SET $To @R1
IF SUBSTR($To,1,3) = "To:" SET $To "", GOTO Get_To
ELSE IF SUBSTR($To,1,3)="cc:" $To = "", SET $Cc "cc:", SEND $To, WAIT
STRING "CC:", WAIT QUIET "1", RETURN
ELSE IF SUBSTR($To,1,1)=" " SET $To "", GOTO Get_To
ELSE IF $To="" GOTO Get_To
PARSE $To $T1 " ," $T2
SET $To TRIM($T2," ", " ") | " " | TRIM($T1," "," ")
GOTO Get_To

```

The `Get_To` routine is a complex loop that builds up a string named `$To` as it reads records from the outgoing message's envelope file. If it encounters a record that starts with the characters "EMS:" it jumps to the `EMS` routine; if it encounters one that begins "cc:" it returns to the calling routine. Otherwise, it continues to read through the file until it has a valid address.

The Send_To Routine

Once determined, the address is transmitted using the `Send_To` routine.

```

*Send_To ($Send, $Desire)
PERFORM Writestring(7, $Prefix | $To)
SEND $Send
WHEN STRING 1 "not found" GOTO Err
WHEN STRING 2 "You have" GOTO Err
WHEN STRING 3 "must" GOTO Err
WHEN STRING 4 "Please enter" GOTO Err
WHEN STRING 5 "There is" GOTO Err
WAIT STRING $Desire
WAIT QUIET "1"
RETURN

```

`Send_To` accepts two parameters: `$Send`, the string to send; and `$Desire`, the desired response. It starts by writing the string `$To` to table 7, then sends it to MCI and initiates a series of `WHEN STRING` conditions to identify various MCI error messages. It then waits for MCI to return the string `$Desire`, after which it returns to the routine that called it.

The Get_cc Routine

The `Get_cc` routine is similar to `Get_To`.

```

*Get_cc
SET $J ""
SET $Prefix "cc: "
RECORD READ 1
IF SUBSTR(@R1,1,4)="EMS:" $To = $Cc, SET %To 0, GOTO EMS

```

```

IF SUBSTR(@R1,1,6)="MCI ID" and $Cc<>"" SET $Cc $Cc | " / " | @R1, RECORD
  READ 1
IF SUBSTR(@R1,1,6)="REMOTE" SET $Cc $Cc | " / " | @R1, RECORD READ 1
IF $Cc <>"" and $Cc<>"cc:" PERFORM Send_To ($Cc, "CC:"), WAIT QUIET "1",
  IF %Cancel > 0 RETURN
SET $Cc @R1
IF SUBSTR($Cc,1,3) = "cc:" SET $Cc "", GOTO Get_cc
ELSE IF $Cc="Subject:" $Cc="", SEND $Cc, RETURN
ELSE IF SUBSTR($Cc,1,1)=" " SET $Cc "", GOTO Get_cc
ELSE IF $Cc="" GOTO Get_cc
PARSE $Cc $T1 " ," $T2
SET $Cc TRIM($T2," ", " ") | " " | tRim($T1," "," ")
GOTO Get_cc

```

Like `Get_To`, the `Get_cc` routine calls the `Send_To` routine once it has extracted a valid MCI address from the envelope file. And, also like `Get_To`, it calls the `EMS` routine if it encounters an `EMS` address.

The EMS Routine

The `EMS` routine processes the `EMS` address found in the envelope file, sends it to MCI, and then returns to either the `Get_To` or `Get_cc` routine.

```

*EMS
SET $J ""
SET $To $To | " (EMS)"
PERFORM Send_To ($To, "EMS"), IF %Cancel > 0 RETURN
SET $Ems1 @R1
SET $Ems1 SUBSTR($Ems1,5,45)
RECORD READ 1
SET $Ems2 @R1
SET $Ems2 SUBSTR($Ems2,8,45)
RECORD READ 1
SET $Ems3 @R1
SET $Ems3 SUBSTR($Ems3,5,45)
SET $To TRIM($Ems1) | " / " | TRIM($Ems2)
SET $Prefix " "
PERFORM Send_To ($To, "MBX"), IF %Cancel > 0 RETURN
SET $To $Ems3, PERFORM Send_To ($To, "MBX"), IF %Cancel > 0 RETURN
SEND ""
SEND "Y"
IF %To=1 WAIT STRING "TO", ELSE WAIT STRING "CC"
WAIT QUIET "1"
SET $To ""
SET $Ems1 ""
SET $Ems2 ""
SET $Ems3 ""
SET $Cc ""
IF %To=1 SET %To 0, GOTO Get_To
GOTO Get_cc

```

The Err Routine

The Err routine is called if MCI fails to accept an address transmitted by the Send_To routine. There are several reasons this might occur; a mistake in the addressee's name or MCI number and the use of a name without an MCI ID number for a subscriber whose name is not unique among subscribers are probably the most common errors.

```
*Err
WHEN CANCEL STRING
PERFORM WriteSTRING(7,"")
PERFORM WriteSTRING(7,"SEND ERROR")
PERFORM WriteSTRING(7,"")
PERFORM Close_and_Clear(7,0)
WAIT QUIET "1"
SELECTION BUFFER
SELECTION SAVE "ERRORLOG.DCM"
CLEAR
```

The Err routine adds the line "SEND ERROR" to the HOLDFILE file for the outgoing message, then uses the DynaComm SELECTION BUFFER and SELECTION SAVE commands to save everything that has appeared on the screen since the CREATE command for the current message was sent to a file called ERRORLOG.DCM. Then it clears the screen and continues with the Err1 routine.

The Err1 Routine

The Err1 routine's job is to back MCI out of the error message and to a point where the script can attempt to send the next message.

```
*Err1
SEND ""
WHEN STRING 1 "Please enter" SEND "0", SEND "/", GOTO Err2
WHEN STRING 2 "TO:" SEND "/", GOTO Err2
WHEN STRING 3 "CC:" SEND "/", GOTO Err2
WHEN STRING 4 "SUBJECT:" SEND "/", GOTO Err2
WHEN STRING 5 "EMS" SEND "/"
WHEN STRING 6 "MBX" SEND "/"
WAIT QUIET "2"
GOTO Err1
```

Once Err1 has done its job, the script continues onto Err2.

The Err2 Routine

Err2 moves the message from the Outbox folder to the Unsent folder, increments the %Unsent and %TotalUnsent variables, copies the message's old

envelope to another file under a new name, deletes HOLDFILE, copies ERRORLOG.DCM (which contains information about the source of the error) to \$Target, then appends a copy of the message body to \$Target, and finally deletes the old \$Header and \$Body files before returning.

```
*Err2
WHEN CANCEL STRING
WAIT STRING "Command:"
SET @R5 @R2
SET $Target $FileName | STR(%Count)
SET @R5.8 $Target
RECORD WRITE 5
INCREMENT %Unsent
INCREMENT %TotalUnsent
FILE COPY $Header $Target | ".ENV"
SET $Target $Target | ".DCM"
PERFORM Close_and_Clear(7,0)
TABLE DEFINE 7 TEXT "HOLDFILE"
FILE COPY "ERRORLOG.DCM" $Target
FILE COPY $Body $Target APPEND
FILE DELETE "HOLDFILE"
FILE DELETE "ERRORLOG.DCM"
FILE DELETE $Header
FILE DELETE $Body
TABLE CLEAR 1
SET $To ""
SET $Cc ""
SET %Cancel 1
RETURN
```

When the user reads the message in the Unsent folder, it will contain the MCI error message, followed by the original body of the message.

The Get_Sub Routine

The Get_Sub routine is used to read the subject of the message from the message's envelope, transmit it to MCI, and then write it to HOLDFILE.

```
*Get_Sub
PERFORM ReadSTRING(1,$Sub)
WAIT QUIET "1"
SEND $Sub
PERFORM WriteSTRING(7,"Subject: " | $Sub)
PERFORM WriteSTRING(7, "Sent: " | DATE() | " at " | TIME())
RETURN
```

The Delete_File Routine

The Delete_File routine clears the Outbox table after the messages in the Outbox have been sent.

```
*Delete_File
TABLE CLEAR 2
PERFORM Save_and_Close (2,"OUTBOX")
RETURN
```

Message-Reception Routines

The next four routines are used to retrieve any mail that is waiting for the user.

The Pickup Routine

Once all the messages in the user's Outbox have been transferred to MCI, Send_Recv_Routine calls the Pickup routine to retrieve any messages that are waiting for the user and add them to the user's Inbox.

```
*Pickup
WAIT QUIET "1"
SEND "Scan"
COLLECT $J
COLLECT UNTIL "m" $M
SET $M FILTER($M, "m ", "")
%I=NUM($M), IF %I > 99 %I = 99
IF %I < 1 WAIT STRING "Command:", CLEAR, RETURN
WHEN STRING 1 "Press" SEND ""
WAIT STRING "Command:"
WHEN QUIET "60" HANGUP, WAIT DELAY "15", PERFORM Logout, PERFORM End
```

The Pickup routine sends the command Scan to MCI, to which MCI responds by sending a blank line, followed by either the message "Your Inbox is empty" or "XX messages in INBOX." The routine assigns the blank line to the junk variable \$J, then captures the next line up to the character "m" in the variable \$M. So if MCI had replied "3 messages in INBOX", \$M would contain "3". Then the routine uses the NUM function to assign the numeric value of \$M (3 in the current example) to %I, and then determines whether %I is greater than 0. If not, it returns to the calling routine. Otherwise, it continues by executing the PR_Loop routine.

The PR_Loop Routine

The PR_Loop routine retrieves the messages reported by MCI one by one.

```
*PR_Loop
%S = 1
WHILE %S < (%I+1)
BEGIN
```

```

SEND "PR " | STR(%s)
SET $File $FileName | STR(%S+%Count)
CLEAR
FILE RECEIVE TEXT $File
SET $Subject ""
WAIT STRING "From:      "
COLLECT $From
WHEN STRING 1 "Subject: ", COLLECT $Subject, WHEN CANCEL STRING
WHEN STRING 2 "EMS: ", COLLECT $Ems, WAIT STRING "MBX: ", COLLECT $MBX,
    WHEN CANCEL STRING, WHEN STRING "Subject: ", COLLECT $Subject, WHEN
    CANCEL STRING
WAIT STRING "Command:"
WAIT QUIET "2"
FILE CLOSE

```

PR_Loop begins by initializing the value of %S, which will be used to track the number of the message being received. Then it sends MCI the command PR, followed by a space and then the string containing the value of %S, thus instructing MCI to “print” (transmit without page breaks) message number %S, as shown in Figure 15.25.

Then it initializes a name for the new file by appending a string containing the value of %S to the end of the \$File variable that was created by the Get_Date routine and instructs DynaComm to send the incoming text to a file with that name. It uses WHEN STRING commands to capture the message’s subject and sender, and the sender’s MCI or EMS address, and assigns them to variables. MCI sends the “Command:” prompt (see Figure 15.25) at the end of the message, so the script waits for that string, followed by two seconds of silence on the communications line, and then closes the file.

This delay serves two purposes: It cleans up the session screen display by allowing DynaComm’s display to catch up to the incoming data stream, and it provides a safety check that prevents the script from closing a message file prematurely if the message being received happens to contain the text “Command:”. The file won’t be closed if message transmission continues after the “Command:” string is received because the routine is waiting for two seconds of silence on the line.

The Process_Incoming_Mess Routine

The Process_Incoming_Mess routine is executed once the message file is closed.

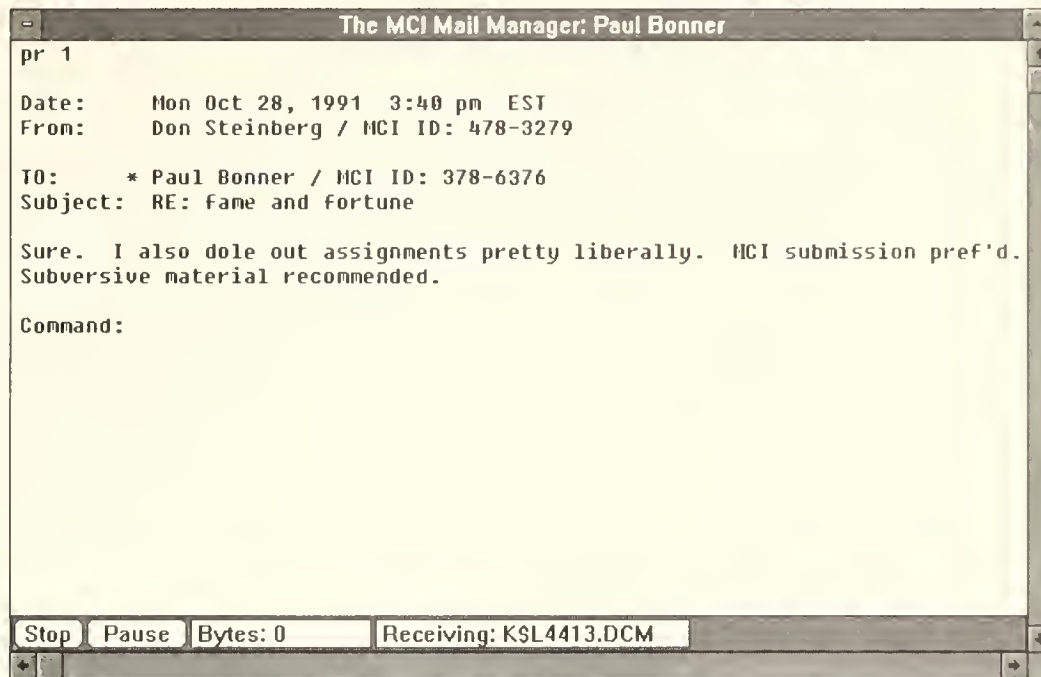
```

*Process_Incoming_Mess
PERFORM Date_Fix
SET %Fs FILESIZE($File | ".DCM")
PARSE $From $From "/" $ID
PARSE $From $From " " $From1
SET $From1 TRIM($From1, " ", " ") | ", "
SET $From TRIM($From)
SET $From $From1 | " " |$From

```


Figure 15.25

The Print Message
command



```
%Re = 0
IF SUBSTR($Subject,1,4)="RE: " PARSE $Subject $Sub1 " " $Sub2, %Re = 1.
    $Subject = SUBSTR($Sub2,1,23)
ELSE $Subject=SUBSTR($Subject,1,23)
PERFORM Table_Write
INCREMENT %s
END
SET %TotalReceived %TotalReceived + %I, SET @S8 STR(%TotalReceived)
RETURN
```

The `Process_Incoming_Mess` routine begins by calling the `Date_Fix` routine, which cleans up the date information in the file containing the message that was just received. Then it uses a variety of `DynaComm` functions to obtain the size of that file, to separate the message sender's name from his or her MCI account number, to reverse the order of the sender's names so that the `$From` variable contains the name in last-name-first order, and to strip the characters "Re:" from the beginning of the message's subject string, if they appear. It then calls the `Table_Write` routine to write an Inbox record for the newly received message.

The Date Fix Routine

The message files sent by MCI and captured by this script contain a header that looks like this:

```
Date: Thu Jul 05, 1990 8:42 am EST **PRIORITY
```

From: Future Soft Engineering Incorporated / MCI ID: 390-5883

TO: * Paul Bonner / MCI ID: 378-6376

Frequently the first line of that header in the received file contains some garbage characters, so that the first line looks like this instead:

```

%$#%$@Date: Thu Jul 05, 1990 8:42 am EST

```

The `Date_Fix` routine is used to remove those characters and restore the appearance of the `Date:` line, by eliminating any characters preceding the “D” in “Date”.

```

*Date_Fix
TABLE DEFINE 9 text $File | ".DCM"
RECORD READ 9 at 0
SET @R9 TRIM(@R9," "," ")
%Dpos = POS(@R9,"D")
%Lendate =LENGTH(@R9)
IF %Dpos<>0 GOTO Df_Loop2
SET @R9 ""
*Df_Loop1
SET @R9 @R9 | " "
IF LENGTH(@R9)<%Lendate GOTO Df_Loop1
RECORD WRITE 9 at 0
RECORD READ 9
*Df_Loop2
%Dpos = POS(@R9,"D")
%Lendate =LENGTH(@R9)
IF %Dpos > 1 SET @R9 SUBSTR(@R9,2,%Lendate-1) | " ", GOTO Df_Loop2
RECORD WRITE 9
*Df_Finish
TABLE CLOSE 9
RETURN

```

`Date_Fix` treats the message file as a sequential-access text file. It opens the file and reads it line by line. If the “Date” string is not found on the first line of the file, it replaces any characters it finds there with spaces, and then reads the next line, removes any characters that precede the “D” from that line, and then closes the file, saving the changes it has made.

The Logout Routine

The Logout routine is used to disconnect from MCI.

```

*Logout
SEND "EX"
*Hang

```

```

WHEN CANCEL
HANGUP
CLEAR
RETURN

```

Logout begins by sending the MCI EX (for exit) command, cancels all current WHEN conditions with the WHEN CANCEL command, then hangs up the line and returns to the routine that called it.

The Loginerror Routine

The Loginerror routine is called if MCI refuses to accept the password sent by the script. It simply hangs up the line, prints the message “password error” to the terminal screen, and cancels execution of the script.

```

*Loginerror
HANGUP
DISPLAY "password error^m"
CANCEL

```

The Table_Write Routine

The Table_Write routine is used to create an Inbóx record for the newly received message.

```

*Table_Write
TABLE DEFINE 0 FIELDS CHAR 81 CHAR 28 CHAR 6 CHAR 10 CHAR 10 CHAR 15
CHAR 1 CHAR 8 CHAR 45 FILE
IF EXISTS("INBOX") PERFORM TABLE_LOAD(0, DIRECTORY DATA | "INBOX")
WHILE NOT EOF
BEGIN
RECORD READ 0
END
SET @R0.2 $Subject
SET @R0.3 STR(%fs)
SET @R0.4 ""
SET @R0.5 DATE() | " " | STR(%s) | " "
SET @R0.6 FILTER($ID,"MCID: mcid",)
IF %re =1 SET @R0.7 "R"
ELSE SET @R0.7 " "
SET @R0.8 $File
SET @R0.9 $From
SET $Ems "", SET $MBX ""
PERFORM PAD ($From, 18)
PERFORM PAD ($Subject, 25)
SET $Fs STR(%fs)
PERFORM PAD ($Fs, 6)
SET @R0.1 "*" < | SUBSTR($From,1,18) | " " | SUBSTR($Subject,1,25) | " " |
    $s | " " | DATE() | " "
RECORD WRITE 0
PERFORM Save_And_Close (0, "INBOX")

```



```
%Re = 0
RETURN
```

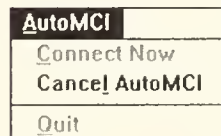
Table_Write loads the Inbox folder into memory and then positions the record pointer after the last record in the table by reading the entire table. Then it writes the sender, subject, file size, and other information about the newly received message to the Inbox record before closing the Inbox and returning.

The Auto_Retrieve Routine

The Auto_Retrieve routine is called if the user elected to start an AutoMCI session rather than a Send/Recv session. The routine begins by drawing a menu, and then connects to MCI.

```
*Auto_Retrieve
CLEAR
MENU
POPUP "&Automci"
ITEM "&Connect Now" GRAYED PERFORM Restart
ITEM "Cance&l AutoMCI" SET POINTER WATCH, FILE CLOSE, HANGUP,
    RESETSERIAL, INCREMENT %Count, PERFORM End
SEPARATOR
ITEM "&Quit" GRAYED PERFORM Auto_Quit
MENU END
PERFORM Get_Date
```

Auto_Retrieve begins by creating a custom menu to control the AutoMCI session. The menu contains three items, as shown here:



Two of the options, Connect Now and Quit, are initially grayed (disabled). The Connect Now item becomes enabled whenever the AutoMCI routine is in its waiting loop, between connections. It allows the user to force AutoMCI to connect immediately rather than waiting for the next scheduled connection.

The Cancel AutoMCI menu button cancels the current AutoMCI session and executes the main M.M.M. script; the Quit button stops script execution altogether.

The Auto_Dial_Seq Routine

Once the menu has been drawn, AutoMCI connects to MCI using the Auto_Dial_Seq routine and two subroutines: Check_When and Conn_Made.

```

*Auto_Dial_Seq
PARSE @S3 $NUM "`" $Pre_Dial
PARSE $Pre_Dial $Pre_Dial "`" $Post_Dial
SET $Phone SETTINGS(PHONENUMBER)
SEND $Pre_Dial | $Phone | $Post_Dial
TIMER ON, TIMER RESET
WHEN STRING 1 "CONNECT" GOTO Conn_Made
WHEN STRING 2 "No" hangup, WAIT DELAY "5", PERFORM RESTART
WHEN STRING 3 "NO" hangup, WAIT DELAY "5", PERFORM RESTART
WHEN STRING 4 "Connect" GOTO Conn_Made
WHEN TIMER "45" HANGUP, RESETSERIAL, WAIT DELAY "5", PERFORM Restart
*Check_When
WAIT DELAY "1"
GOTO Check_When
*Conn_Made
WHEN CANCEL TIMER, WHEN CANCEL STRING, WHEN CANCEL
WHEN QUIET "60" HANGUP, WAIT DELAY "5", PERFORM Restart
PERFORM Loop
WHEN QUIET "60" HANGUP, WAIT DELAY "5", PERFORM Restart

```

The `Auto_Dial_Seq`, `Check_When`, and `Conn_Made` routines replace the standard DynaComm dialing routine with a custom one to avoid the standard DynaComm Dialing dialog box. This dialing routine probably employs a little less error checking than is built into the standard DynaComm dialer, but it doesn't matter, since in the event of any error the routine simply restarts the script and tries again.

The Auto_Main Routine

Once the connection has been made, the script calls the `Loop` routine described above to log the user onto MCI. Then the `Auto_Main` routine takes over.

```

*Auto_Main
SET %Count 0
PERFORM Send_Message
WHEN QUIET "60" HANGUP, WAIT DELAY "5", PERFORM Restart
SET %I %Count
PERFORM Pickup
WHEN QUIET "60" FILE CLOSE, HANGUP, WAIT DELAY "5", PERFORM Restart
PERFORM Logout
WHEN CANCEL
WAIT DELAY "5"
CLEAR
MENU UPDATE 1 1 ENABLED
MENU UPDATE 1 4 ENABLED
PERFORM Time_Set
SET %Con_Time %Time+%Delay, IF %Con_Time>1440 SET %Con_Time %Con_Time - 1440
DISPLAY "Will connect again in less than " | STR(%Con_Time %Time) | " minutes.^M"

```

The `Auto_Main` routine functions much like the `Send_Recv` routine, taking the AutoMCI session through a sequential series of steps: first sending

messages, then picking up any messages waiting for the user, then logging out. Then, once the line has been hung up, it enables the two menu items that were disabled when the AutoMCI menu was first drawn, and displays a message in the terminal window that indicates the amount of time before the next scheduled connection.

The Loop3 Routine

The script then proceeds to the Loop3 routine.

```
*Loop3
WAIT DELAY "5"
PERFORM Time_Set
DISPLAY "^M^M^M^M^M"
DISPLAY STR(%TotalReceived) | " message(s) received.^M"
IF %TotalSent > 0 DISPLAY STR(%TotalSent) | " message(s) delivered.^M"
IF %TotalUnsent > 0 DISPLAY STR(%TotalUnsent) | " message(s) not
    sent.^MCheck UNSENT folder.^M"
IF %Time = 1 Or %Time = 2 WAIT DELAY "120", PERFORM Restart
IF %Con_Time - %Time < 1, HANGUP, PERFORM Restart
WAIT DELAY "6"
PERFORM Time_Set
DISPLAY "^M^M^M^M^M"
DISPLAY "Will connect again in less than " | STR(%Con_Time - %TIME) | "
    minutes.^M"
IF %Time = 1 or %Time = 2 WAIT DELAY "120", PERFORM Restart
IF %Con_Time - %Time < 1, HANGUP, PERFORM Restart
GOTO Loop3
```

The Loop3 routine waits five seconds, checks the time, then displays a status message indicating the number of messages that have been sent, received, or not sent in the AutoMCI session. Then it compares the current time (%Time) to the next scheduled connection time (%Con_Time) and restarts the script (thus forcing a connection to MCI) if the difference between them is less than one minute.

Because the routine used to determine the next scheduled connection time is somewhat inexact when dealing with intervals that span the hour of midnight, the script simply restarts itself if the current time is 1 or 2 (one or two minutes past midnight), thus forcing the script to determine another connection time that doesn't span the midnight hour.

If the scheduled start time hasn't arrived, the script waits another six seconds, then displays another message indicating the amount of time before the next scheduled connection, as shown in Figure 15.26. Then it checks the time again and, if the scheduled start time still hasn't arrived, jumps back to the beginning of Loop3.

The Restart Routine

When the scheduled start time arrives, the script jumps to the Restart routine.

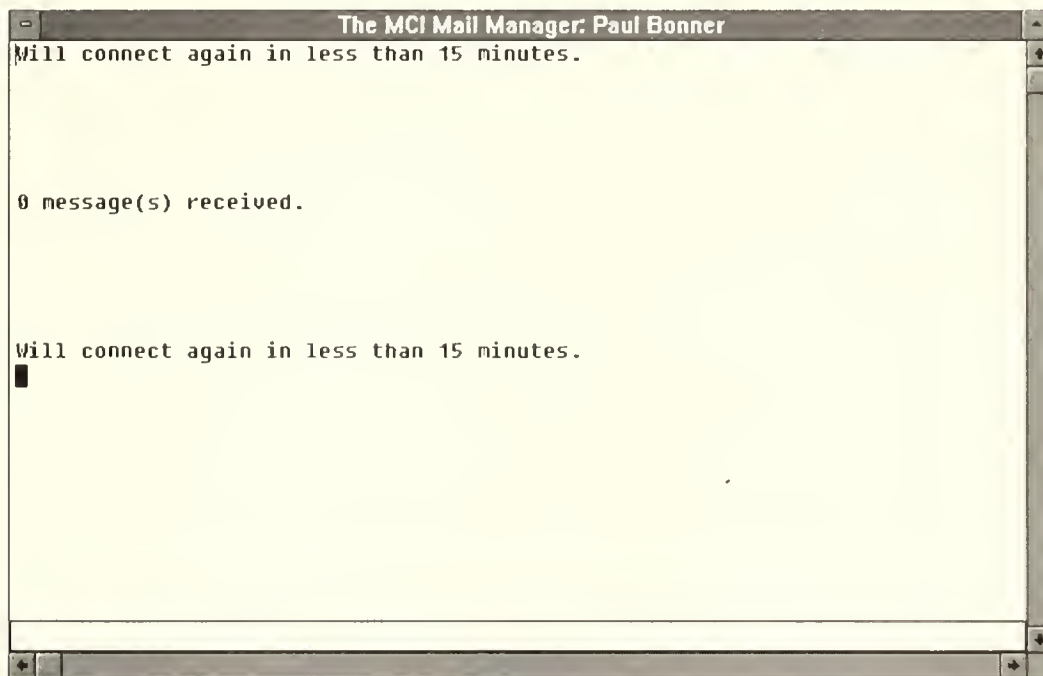

```

*Restart
SET POINTER ARROW
SET @S5 STR(%TotalSent) | "&" | STR(%TotalReceived) | "&" | STR(%TotalUnsent)
SAVE
HANGUP
RESETSERIAL
RESTART

```

Figure 15.26

The AutoMCI session status message



The Restart routine stuffs the current values of %TotalSent, %TotalReceived, and %TotalUnsent into the @S5 variable, saves the settings sheet, and restarts the script.

The Time_Set Routine

The Time_Set routine converts the current time into a single number indicating the number of minutes past midnight.

```

*Time_Set
SET $Time TIME()
SET %H NUM(SUBSTR($Time,1,2))
SET %M NUM(SUBSTR($Time,4,2))
SET $Ampm SUBSTR($Time,7,1)
IF %H=12 SET %H %H - 12
IF $Ampm ="P" SET %H %H+12
SET %Time (%H*60)+%M
IF %Time>1440 SET %$Time %Time-1440
RETURN

```

If the number of minutes is greater than 1,440 (which would occur only if the time was between 12:01 am and 12:59 am) the routine subtracts 1,440 from it, thus converting 12:59 am to a value of 59 minutes.

The Auto_Quit Routine

The Auto_Quit routine is called when the user selects the Quit item from the AutoMCI menu.

```
*Auto_Quit
SET POINTER WATCH
SET @S5 "0&0&0"
PARSE`@S6 $String1 "~" $String2
PARSE $String2 $INIT "~" $String3
SET $INIT "1"
SET @S6 $String1 | "~" | $INIT | "~" | $String3
SAVE
FILE CLOSE
HANGUP
RESETSERIAL
SCREEN HIDE
QUIT
```

Auto_Quit simply performs a little housekeeping, reinitializing the values of settings variables @S5 and @S6, so that the AUTOMCI.DCT script will interpret them as indicating a cold start, before hanging up the connection and stopping script execution.

Standard Library Routines

The remaining routines in EMAIL.DCP are general-purpose library routines. Several of these are repeats of routines found in AUTOMCI.DCP and TM.DCP. Specifically, EMAIL.DCP's Decode, WriteString, Close_and_Clear, Table_Def_and_Load, and ReadString routines are identical to those with the same names in AUTOMCI.DCP. Its Pad, WriteInteger, and ReadInteger routines are identical to those in TM.DCP.

The BackUpFiles Routine

BackUpFiles creates a backup copy of the specified file.

```
^BackUpFiles ($BackItUp)
FILE COPY DIRECTORY DATA | $BackItUp TO DIRECTORY DATA | $BackItUp | ".BAK"
RETURN
```

Decode, which decrypts the user's password, appears in AUTOMCI.DCP.

The Table_Load Routine

Table_Load is used to load a specified table.

```
*Table_Load (%Counter, $Tabfile)
IF %ErrCount > 5 CANCEL
TABLE LOAD %Counter FROM $Tabfile AS DYNACOMM
IF ERROR() INCREMENT %ErrCount, GOTO Table_Load
SET %ErrCount 0
RETURN
```

The Save_and_Close Routine

Finally, Save_and_Close is used to save the contents of the specified table to the specified file before closing and clearing the table.

```
*Save_and_Close (%Tablenum, $TableName)
TABLE SAVE %Tablenum TO DIRECTORY DATA | $TableName AS DYNACOMM
PERFORM Close_and_Clear(%Tablenum)
RETURN
```

That concludes the EMAIL.DCP script. Next the PM.DCP script will be discussed. It is used to maintain the Public and Private address books.

Exploring PM.DCP

The PM.DCP (Phonebook Module) module is used to edit the user's Private address book, and the Public address book as well, if it is located on the user's local drive.

The Phon_Man Routine

The Phon_Man routine determines whether the Public address book is located in the user's \DYNACOMM\DAT directory.

```
*Phon_Man (%Env)
SET %Admin 0
SET $Mailbox DIRECTORY DATA
SET DIRECTORY DATA System(0x0f01,"Data")
If Exists("Public.Pbk") FILE COPY "Public.Pbk" TO "Public.Bak", SET
    %Admin 1
FILE COPY "Private.Pbk" TO "Private.Bak"
TABLE DEFINE 8 FIELDS CHAR 45 CHAR 45 CHAR 45 CHAR 45 File
TABLE LOAD 8 FROM "Private.Pbk" AS DYNACOMM
IF %Admin=1 TABLE DEFINE 13 FIELDS CHAR 45 CHAR 45 CHAR 45 CHAR 45 File.
    TABLE LOAD 13 FROM "PUBLIC.PBK" AS DYNACOMM
IF %Admin=1 AND %Env=1 SET %Ptable 13, %Button=2, $Phonebook="Public",
    Else SET %Ptable 8, $Phonebook="Private", %Button=1
^Loaded
SET %Ip 1
SET %Edit 0
SET $N1 " "
SET $N2 $N1, SET $N3 $N1, SET $N4 $N1
SET $Name "", SET $Id "", SET $Address3 "", SET $Address4 ""
```


The value of %Admin is set according to where the Public address book is placed. If %Admin is equal to 1, it indicates that the Public book is in \DYNACOMM\DAT and the user has the right to edit both address books, so both are loaded into memory. Otherwise, only the Private book is loaded.

The Main_Phon Routine

The Main_Phon routine is where the address book is made available for editing.

```
*Main_Phon
SET $Message "Phonebook Management"
DIALOG (.,160,128) $Message
RADIOGROUP (8,14,,) %Button "Phonebook" PERFORM Up_Phon
RADIOBUTTON (.,14,,) "Private"
IF %Admin=1 Radiobutton (.,14,,) "Public"
LISTBOX (6,30,140,70) %Ptable %Ip
BUTTON (12,96,.11) DEFAULT "&Edit" SET %Edit 1, SET %Ip Listbox (1),
    PERFORM Edit_Book, RESUME
BUTTON (.,.11) "&Add" PERFORM Add_Name, RESUME
BUTTON (.,.11) "&Cut" SET %Ip Listbox(1), Perform Delete_Name, Perform
    Up_Phon
WIDEBUTTON (52,110,.11)"&Done" Dialog Cancel, SET DIRECTORY DATA
    $Mailbox, RETURN
DIALOG END
Wait RESUME
SET $N1 " "
SET $N2 $N1, SET $N3 $N1, SET $N4 $N1
SET $Name "", SET $Id "", SET $Address3 "", SET $Address4 ""
SET %Edit 0
GOTO Main_Phon
```

Main_Phon presents the Private address book (and the Public one if %Admin equals 1) in a list box, along with buttons that offer the user the options of editing an existing entry in the selected address book, cutting an existing entry, and adding a new one, as shown in Figure 15.27.

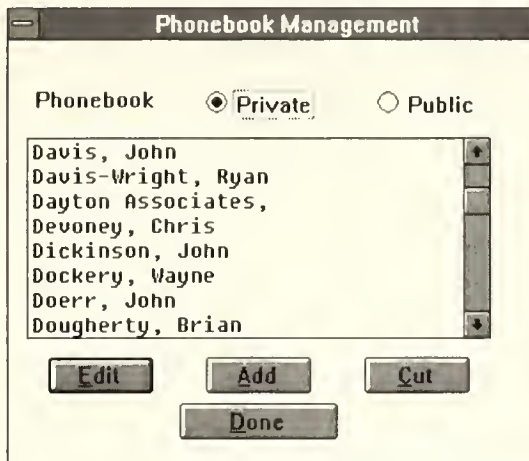
The Add_Name Routine

The Add_Name routine is called when the user selects the Add button on the Phonebook Management dialog box. It sets the values of the string variable \$Message1 and the integer variable %Edit, and then jumps to the Add_Edit_Dialog subroutine, which is shared by the Add_Name and Edit_Name routines.

```
*Add_Name
SET $Message1 "Add Name To " | $Phonebook
SET %Edit 0
GOTO Add_Edit_Dialog
```

Figure 15.27

The Phonebook Management dialog box



The Edit_Book Routine

Edit_Book, called when the user selects the Edit button on the Phonebook Management dialog box, sets a series of variables equal to the current contents of the record being edited, and then jumps to the Add_Edit_Dialog routine.

```
*Edit_Book
IF %Ip < 0 RETURN
TABLE DEFINE 12 FIELDS CHAR 45 CHAR 45 CHAR 45 CHAR 45
RECORD READ %Ptable AT %Ip
SET @R12 @R%Ptable
RECORD WRITE 12
SET $N1 @R12.1
PARSE $N1 $Last "," $First
SET $Ft TRIM($First," "," ")
SET $Lt TRIM($Last," "," ")
SET $N1 $Ft | " " | $Lt
SET $N2 @R12.2
SET $N3 @R12.3. If SUBSTR($N3,1,1)=" " SET $N3 SUBSTR($N3,2,29)
SET $N4 @R12.4. If SUBSTR($N4,1,1)=" " SET $N4 SUBSTR($N4,2,29)
SET %Edit 1
TABLE CLOSE 12
TABLE CLEAR 12
SET $Message1 "Edit " | $N1
GOTO Add_Edit_Dialog
```

The Add_Edit_Dialog Routine

The Add Name to... dialog box, created in the Add_Edit_Dialog routine, displays the four fields of the current address book record, as shown in Figure 16.28. These fields will be blank if a new entry is being created; if an existing entry is being edited, they will reflect its current contents.

```

*Add_Edit_Dialog
TABLE DEFINE 12 FIELDS CHAR 45 CHAR 45 CHAR 45 CHAR 45
DIALOG CANCEL
DIALOG (.,190,86) $Message1
EDITTEXT (4,8,124,) 120 "Name: " $N1
EDITTEXT (4,20,124,) 120 "Mci Id: " $N2
EDITTEXT (4,32,124,) 120 "Ems 1: " $N3
EDITTEXT (4,44,124,) 120 "Ems 2: " $N4
BUTTON (56,66,.11) DEFAULT "OK" RESUME
BUTTON (.,.11) "Cancel" DIALOG CANCEL, RETURN
DIALOG END
WAIT RESUME
SET $Name TRIM(Editttext(1))
SET $Id TRIM(Editttext(2))
SET $Address3 TRIM(Editttext(3))
SET $Address4 TRIM(Editttext(4))
IF %Edit=1 IF $Name=$N1 AND $Id=$N2 RETURN
IF %Edit=0 IF $Name = "" AND $Id="" RETURN
PARSE $Name $First " " $Last
SET $Ft TRIM($First)
SET $Lt TRIM($Last)
SET %P 0
SET $P UPPER($Last)
SET %P POS($P, "List",)
IF %P = 0 SET $Name $Lt | ", " | $Ft
IF SUBSTR($Id,1,1)="M" Or SUBSTR($Id,1,1)="M" SET $Id FILTER($Id,"Mcid:
Mcid:",)
IF SUBSTR($Id,1,1)<>"E" AND SUBSTR($Id,1,1)<>"E" AND LENGTH($Id)> 0 SET
$Id "Mci Id: " | $Id
IF %Edit= 1 TABLE COPY %Ptable TO 12 EXCLUDE @R%Ptable, ELSE TABLE COPY
%Ptable TO 12
SET @R12.1 $Name
SET @R12.2 $Id
SET @R12.3 $Address3
SET @R12.4 $Address4
RECORD WRITE 12
TABLE COPY 12 To %Ptable
TABLE SORT %Ptable 1
SET %Edit 0
RETURN

```

Once the user has made the desired additions or changes to the record and pressed the OK button, the Add_Edit_Dialog routine cleans up the data entered by the user, reversing the first and last names and making sure that MCI and EMS addresses are entered correctly. It then copies the record to the selected address book.

The Delete_Name Routine

The Delete_Name routine is used to delete the selected name from the current address book, using the by now familiar COPY EXCLUDE method.


```

*Delete_Name
IF %Ip < 0 RETURN
RECORD READ %Ptable At %Ip
TABLE DEFINE 12 FIELDS CHAR 45 CHAR 45 CHAR 45 CHAR 45
TABLE COPY %Ptable To 12
RECORD READ 12 At %Ip
TABLE CLEAR %Ptable
TABLE COPY 12 To %Ptable EXCLUDE @R12
TABLE CLOSE 12
TABLE CLEAR 12
RETURN

```

Figure 15.28

The Add Name to... dialog box

The Up_Phone Routine

The final routine in PM.DCP, Up_Phon, is used to update the address book list box and the address book selection radio buttons when the user selects a radio button.

```

*Up_Phon
SET %Up Radiogroup(1)
IF %Up=1 $Phonebook="Private". %Ptable=8
IF %Up = 2 %Ptable = 13. $Phonebook = "Public"
SET %Button %Up
DIALOG UPDATE LISTBOX 1 TABLE %Ptable %Ip. DIALOG UPDATE RADIOGROUP 1 %Button
RETURN

```

That concludes the PM.DCP listing, which leaves only the ONLINE.DCP script to consider.

Exploring ONLINE.DCP

ONLINE.DCP is launched when the user selects the button labeled "Terminal" on AUTOMCI.DCP's main Mailboxes screen. Its primary purpose is to dial into MCI, log the user in, and then end.

The Terminal Routines

In addition to routines for calling MCI and logging in, ONLINE.DCP contains several small routines that can be launched when the user selects one of the on-screen function key buttons that appear in a Terminal session.

The Online Routine

The Online routine begins by dialing into MCI. It calls a series of routines: Login, Loop, Prompts, Logout, Loginerror, and Decode. With the exception of Login, these are identical to the routines with the same names in the EMAIL.DCP script. (Decode is also used in AUTOMCI.DCP.)

Login differs by directly executing the AUTOMCI script if no connection is made, rather than jumping to the End routine as it does in EMAIL.DCP.

```
*Online
SET TERMTITLE "Online To MCI"
WINDOW MAXIMIZE
SCREEN SHOW
FKEYS SHOW
WINDOW RESTORE
SET %I 0
SET $Name @S1
SET $Pass @S2
PERFORM Decode
PARSE @S3 $Num "`" $Pre_Dial
PARSE $Pre_Dial $Pre_Dial "`" $Post_Dial
PERFORM Login
PERFORM Loop
CANCEL

*Login
SET $N1 FILTER($Num,"-./ ","")
DIAL $N1 RETRY 3 DELAY 30
IF NOT CONNECT() DISPLAY "NO CONNECTION^M",CLEAR, EXECUTE "AUTOMCI"
RETURN
```

Once ONLINE.DCT has logged onto MCI, the script ends. Any further script actions must be triggered from the on-screen function keys shown in Figures 15.29 and 15.30. I defined the function keys shown in Figure 15.29 using DynaComm's Function Keys settings dialog box, shown in Figure 15.30.

The Capture Routine

The only unique code in ONLINE.DCP besides the Online routine, is found in the Capture and Print routines. The Capture routine is called when the user selects the function key labeled "Text Capture" during a terminal session. The function key is assigned the command ^\$E 'online*capture', which instructs DynaComm to execute the routine labeled "Capture" in the ONLINE script.

Figure 15.29

An M.M.M. Terminal session

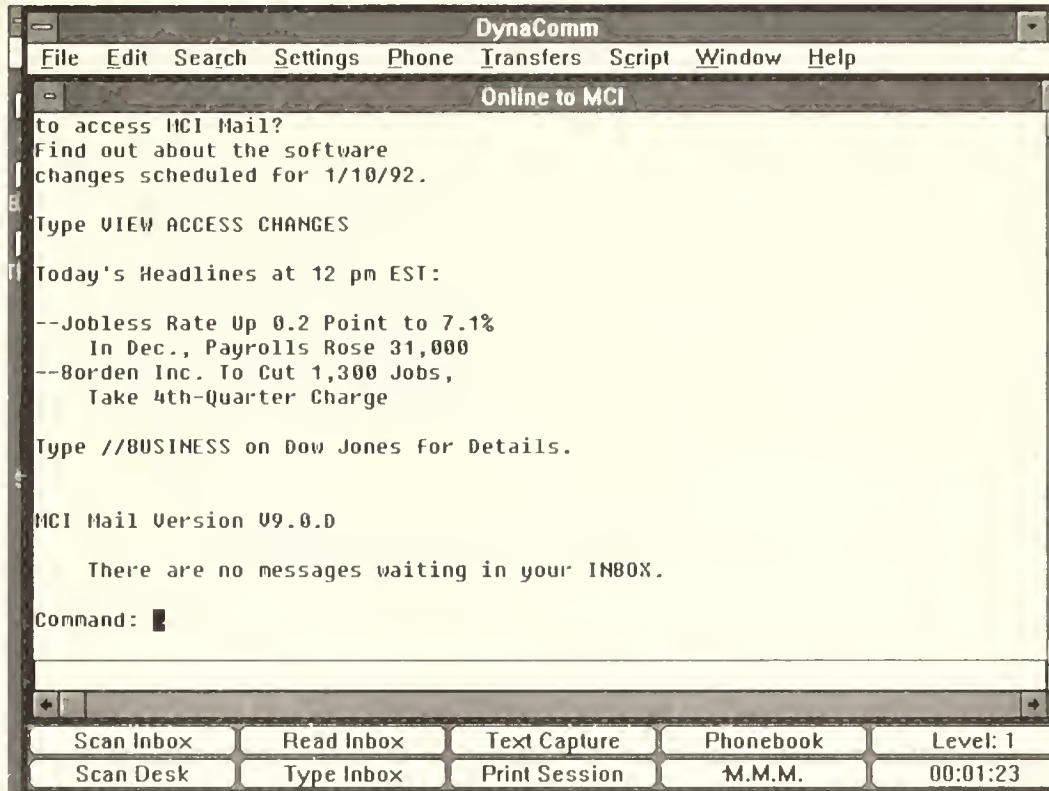
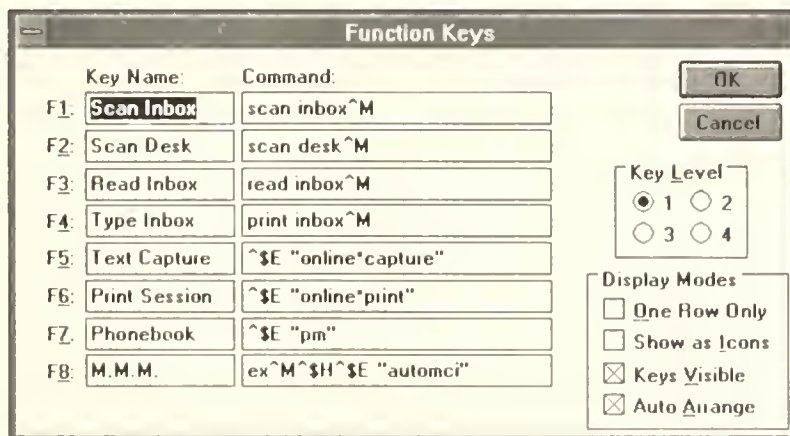


Figure 15.30

The function key definition screen



```

^Capture
%Cap=0
DIALOG (.,140.) "Capture"
MESSAGE (4,...) "Capture to a text file?"
BUTTON (12,...) "OK" SET %Cap 1. RESUME
BUTTON "Cancel" SET %Cap 0. RESUME
DIALOG END
    
```



```

WAIT RESUME
DIALOG CANCEL
IF %Cap = 1 FILE RECEIVE TEXT "CAPTURE.TXT" APPEND
IF %Cap = 2 FILE CLOSE
RETURN

```

The Capture routine opens a simple dialog box that asks the user if he or she wishes to capture the terminal session to a text file. If the user selects the OK button, the dialog box closes and incoming text is captured to the file CAPTURE.TXT. Otherwise the script issues the FILE CLOSE command to close CAPTURE.TXT, if it is open.

The Print Routine

The Print routine is called when the user selects the Print Session function key.

```

*Print
%Print=0
DIALOG (.,140,) "Print Log"
MESSAGE (4,,) "Spool terminal session to printer?"
BUTTON (12,,) "OK" SET %Print 1, RESUME
BUTTON "Cancel" SET %Print 0, RESUME
DIALOG END
WAIT RESUME
DIALOG CANCEL
*Print_Stuff
IF %Print= 0 PRINT TERMINAL OFF, RETURN
PRINT TERMINAL ON
RETURN

```

The Print routine verifies that the user wants to spool the current session to the printer. If so, the command PRINT TERMINAL ON is issued to instruct DynaComm to print everything that is sent or received over the communications line. Otherwise, PRINT TERMINAL OFF is used to cancel printing of the session.

This concludes the listing of the ONLINE.DCP script.

Wrapping Up M.M.M.

Now, sadly, this discussion of the M.M.M. project, the last and most complex of the projects presented in this book, comes to an end.

I hope the discussion of M.M.M.'s code has been instructive. Certainly it demonstrates just how much one can achieve using application macro languages. Although M.M.M. is fairly complex, developing an application of similar functionality is not beyond the reach of any skilled PC user who takes the trouble to master an application macro language such as DynaComm's.

However, thanks to the modular and event-driven structure of Windows applications, there is no need ever to build a project as ambitious as M.M.M. in a single fell swoop. M.M.M. was built in pieces. The message-retrieval routine was built first, followed by the message-sending routines and the main Mailboxes display screen. Message-management functions such as Move and Delete came much later, after a long period of experimentation with and fine-tuning of the appearance and operation of the existing routines.

The problem with any project as complex as M.M.M. is that it is never really finished. As I wrote this chapter I thought of all kinds of new features I would like to add to M.M.M. one day—an unlimited number of user-defined mailboxes, the ability to send fax messages via MCI, better message-editing routines, and so on. It's a tribute to Windows and the power of the best application macro languages that all those extensions, and more, are possible for the DynaComm macro programmer. No doubt as you explore M.M.M. and adapt its code to your own uses, you'll find even more ways to extend its power. Good luck, and have fun with it.

Commercial DLLs and Custom Controls

The following pages present a guide to commercially available dynamic link libraries and custom controls for Windows application development.

The judicious use of these controls and code libraries can enable you to build powerful applications far more quickly and easily than would be possible if you had to program the functions they provide yourself. Moreover, these products allow you to extend existing applications in directions that their internal macro languages don't support. For instance, with a few DLL calls you could add support for TCP/IP (transmission control protocol and Internet protocol) network communications, or for ISAM (indexed sequential-access method) data storage, or even the ability to play animation files or digitized audio, to a custom application that you've built in Word for Windows or Superbase 4 or some other off-the-shelf application.

The prices for these products range from about \$50 for some of the small custom-control libraries, up to around \$1,000 for the most extensive database-management facilities. In most cases, the applications you build using these libraries and controls can be distributed without paying any additional royalties to the developer of the library.

AccSys

Function: Database access

Works with: Any development tool that provides DLL access

Description: AccSys for Paradox and AccSys for dBASE provide a large collection of DLL function calls for complete control of Paradox and dBASE files, respectively.

Copia International, Ltd.

1342 Avalon Court

Wheaton, IL 60187

(708) 682-8898

Agility/VB

Function: Database access

Works with: Visual BASIC

Description: Agility/VB provides Visual BASIC programmers with a set of DLL functions for manipulating and querying dBASE III+ and dBASE IV files and native Agility-format files using a query-by-example interface. It includes sample programs and extensive on-line help.

Apex Software Corp.

4516 Henry Street, Suite 401

Pittsburgh, PA 15213

(412) 681-4343

Ami Pro Macro Developer's Kit

Function: Macro utilities

Works with: Ami Pro

Description: The Ami Pro Macro Developer's Kit includes a 700-page manual on the Ami Pro macro language, a dialog editor with SmartIcons, sample macros, and a macro source-code stripper for creating runtime versions of Ami Pro applications.

Lotus Development Corporation

55 Cambridge Parkway

Cambridge, MA 02142

(617) 577-8500

Autodesk Animation Player for Windows

Function: Playing FLI and FLC animations

Works with: Any development tool that can make DLL calls

Description: Autodesk Animation Player for Windows provides the ability to script, edit, and play back animations in FLI and FLC formats from hard disks or CD-ROMs. A second product, The Autodesk Animation Player DLL for Visual BASIC, provides playback-only access to FLI- and FLC-format animations for Visual BASIC.

Autodesk, Inc.

2320 Marinship Way

Sausalito, CA 94965

(415) 332-2344

BridgIt

Function: Database access

Works with: Any development tool that can make DLL calls

Description: BridgIt gives applications full access to dBASE and Clipper files, including the ability to open, index, read, write, and update files utilizing either dBASE- or Clipper-format indexes. It includes sample code for access from Visual BASIC.

Unelko Corporation

7428 East Karen Drive

Scottsdale, AZ 85260

(602) 991-7272

ButtonTool

Function: Custom controls

Works with: Visual BASIC

Description: ButtonTool provides custom command buttons for Visual BASIC, including 3-D effects, shadows, borders, graphic symbols, and a variety of *button-down* effects, which determine how the button will look when it is pressed.

OutRider Systems

3701 Kirby Drive, Suite 1196

Houston, TX 77098

(713) 521-0486

ChartBuilder for Visual BASIC

Function: Graphing

Works with: Visual BASIC

Description: The ChartBuilder custom control provides Visual BASIC developers with access to a variety of two- and three-dimensional graphs, including bar graphs, pie charts, x-y graphs, and others.

Pinnacle Publishing, Inc.

18000 72nd Avenue South, Suite 217

Kent, WA 98032

(206) 251-1900

Custom Control Factory

Function: Custom controls

Works with: Visual BASIC

Description: Custom Control Factory includes animated buttons; multistate buttons that change appearance each time they are clicked; and enhanced pushbuttons; check boxes, option buttons, and toolbars. Another tool available from Desaware, CCF-Cursors, enhances cursor creation and manipulation in Visual BASIC, and allows any Visual BASIC control to process the full range of mouse messages.

Desaware

5 Town & Country Village, Suite 790

San Jose, CA 95128

(408) 377-4770

Dialoger

Function: Dialog box utility

Works with: Superbase 4

Description: Dialoger works with the Whitewater Resource Toolkit to simplify creation of dialog boxes in Superbase 4. Dialog boxes can be created with the Whitewater Resource Toolkit, and Dialoger will generate the DML code needed to insert them into Superbase applications.

Application Methods

11612 82nd Avenue South

Seattle, WA 98178

(206) 772-3215

Distinct TCP/IP for Windows

Function: Network access

Works with: Any development tool that provides DLL access

Description: This DLL extends TCP/IP, RPC (remote procedure call), and NFS (network file systems) networking capabilities to Windows developers, allowing them to build custom TCP/IP network applications or distributed applications such as Windows front ends to UNIX databases. It includes a configuration utility and a network monitor for monitoring host communications and data transmission traffic.

Distinct Corporation

P.O. Box 3410

Saratoga, CA 95070

(408) 741-0781

EditTool

Function: Custom control

Works with: Visual BASIC

Description: EditTool adds to Visual BASIC's edit controls and provides the ability to perform input field masking (restricting input to certain characters), add spin controls, and customize input fields with shadows, borders, and 3-D effects.

OutRider Systems

3701 Kirby Drive, Suite 1196

Houston, TX 77098

(713) 521-0486

FastData

Function: Database management

Works with: Toolbook

Description: FastData is a set of DLL functions for storing, retrieving, searching, and sorting data more quickly than Toolbook alone can do. Data can be arranged in arrays, lists, stacks, queues, and property tables.

Heizer Software

P.O. Box 232019

Pleasant Hill, CA 94523

(800) 888-7666

Graphics Server SDK

Function: Graphing

Works with: Any development tool that provides DLL or DDE access

Description: The Graphics Server DLL enables developers to give their applications the ability to create a variety of two- and three-dimensional graphs, including pie and bar charts, area graphs, line graphs, high-low-close graphs, and x-y graphs. Graphics Server can be accessed through either DLL calls or DDE. It includes sample source code for accessing the DLL from C, SQL Windows, Superbase 4, Toolbook, Microsoft Excel, and Word for Windows.

Pinnacle Publishing, Inc.

18000 72nd Avenue South, Suite 217

Kent, WA 98032

(206) 251-1900

ImageMan

Function: Image display and printing

Works with: Any development tool that can make DLL calls; Visual BASIC-specific version also available

Description: The ImageMan DLL enables your application to display and print TIFF, PCX, GIF, BMP, DIB, and WMF graphics files. A complementary product called ImageMan/X adds the ability to save files in any of those formats. ImageMan/VB, the Visual BASIC version, provides the same functions as the ImageMan product in the form of a custom control, and adds the ability to scale, zoom and scroll images.

Data Techniques, Inc.

1000 Business Center Drive, Suite 120

Savannah, GA 31405

(912) 651-0003

MicroHelp Communications Library

Function: Communications

Works with: Visual BASIC

Description: Communications library provides communications subroutines and functions (some written in assembly language) that can be called from Visual BASIC. It includes file-transfer routines supporting XModem, YModem, ZModem, and CompuServe B protocols.

MicroHelp

4636 Huntridge Drive

Roswell, GA 30075

(404) 552-0565

MicroHelp Muscle

Function: Code library

Works with: Visual BASIC; QuickBASIC version also available

Description: MicroHelp Muscle offers hundreds of assembly language subroutines to perform a variety of tasks—including file access, array management, and input masking—more quickly and with a smaller program size than would be possible in Visual BASIC alone. It includes a programmer's guide, two reference manuals, and source code.

MicroHelp

4636 Huntridge Drive

Roswell, GA 30075

(404) 552-0565

MicroHelp Network Library

Function: Network access

Works with: Visual BASIC

Description: MicroHelp Network Library provides network access (user and administrator services) to Visual BASIC applications. It supports Novell, LANtastic, and NetBIOS-compatible networks.

MicroHelp

4636 Huntridge Drive

Roswell, GA 30075

(404) 552-0565

Microsoft LAN Manager Toolkit for Visual BASIC

Function: Network management

Works with: Visual BASIC

Description: The LAN Manager Toolkit includes a number of tools for customizing LAN Manager-based networks, and includes a system-performance graphing utility and sample utilities for common network-management and diagnostic needs.

Microsoft Corporation

One Microsoft Way

Redmond, WA 98027

(800) 227-4679 (Microsoft Inside Sales)

Microsoft Visual BASIC Library for SQL Server

Function: Data access

Works with: Visual BASIC

Description: Microsoft Visual BASIC Library for SQL Server provides custom controls to facilitate access to Microsoft's SQL Server for Visual BASIC applications.

Microsoft Corporation

One Microsoft Way

Redmond, WA 98027

(800) 227-4679 (Microsoft Inside Sales)

Microsoft Visual BASIC Professional Toolkit

Function: Custom controls

Works with: Visual BASIC

Description: The Professional Toolkit includes a broad set of custom controls and extensions to the basic Visual BASIC package, including a spreadsheet-like grid, support for MDI child windows, 3-D controls, bitmapped buttons, support for Pen-Windows and Multimedia Windows, a chart control, a utility for creating installation programs, the Visual BASIC Control Developer's Kit, and supplemental documentation.

Microsoft Corporation

One Microsoft Way

Redmond, WA 98027

(800) 227-4679 (Microsoft Inside Sales)

Microsoft Word for Windows Developer Kit

Function: Macro development kit

Works with: Word for Windows

Description: The Microsoft Word for Windows Developer Kit includes a handbook for Word for Windows developers; a WordBASIC manual with a companion disk containing example macros, the Microsoft Excel Dialog Editor, and macros for translating Excel dialog box code to Word dialog code; a Word for Windows technical reference; and a disk containing WordBASIC tools and sample macros.

Microsoft Corporation

Word for Windows Developer Desk

Ridgewood F/4

One Microsoft Way

Redmond, WA 98073

(800) 227-6444, extension 6581

ObjecTrieve/VB

Function: Database management

Works with: Visual BASIC

Description: ObjecTrieve/VB consists of a set of custom controls and forms that add ISAM-based data-management tools to Visual BASIC. Based on the X/OPEN standard, ObjecTrieve/VB can create, search, index, and manage database files, and supports multipart indexes, automatic updating of index files, container files, and multiple-record access streams.

The company also offers two optional add-ons for ObjecTrieve: DbControls and BLOB Manager. DbControls can be used to build complete ObjecTrieve/VB applications without writing any code. BLOB Manager can be used to add large binary objects (such as bitmapped images or digitized audio or video) to an ObjecTrieve/VB database.

Coromandel Industries, Inc.

70-15 Austin Street, Third Floor

Forest Hills, NY 11375

(800) 535-3267

PC Comnet DLL

Function: Netware access

Works with: Any development tool that provides DLL access

Description: This dynamic link library gives applications access to Novell Netware API calls.

PC Comnet, Inc.

31 Progress Court, Unit 5A
Scarborough, Ontario M1G 3V5
Canada
(416) 289-1331

PDQComm for Windows

Function: Communications

Works with: Visual BASIC

Description: A library of Visual BASIC subroutines and function calls, PDQComm offers complete low-level control over the Com ports, as well as high-level routines for sending and receiving data using the XModem or YModem protocol. Several terminal-emulation subroutines are also provided, including ANSI, DEC VT52 and VT100, and Data General D215. Complete source code (with commentary) is included.

Crescent Software, Inc.

11 Bailey Avenue
Ridgefield, CT 06877
(203) 438-5300

PowerLibW 3.1

Function: Database access

Works with: Visual BASIC, Actor 4.0, and other development languages that support DLL calls

Description: PowerLibW 3.1 is a database-management system in the form of a DLL, giving access to dBASE-compatible files. It allows access and manipulation of database files, indexing, and querying. It can also be used with Microsoft SQL Server to create local dBASE files. PowerLibW includes extensively commented source code for a Visual BASIC DBF editor and DBF file viewer.

ETN Corporation

R.D. 4 Box 659
Montoursville, PA 17754
(717) 435-2202

PowerShoW 3.2

Function: Image management

Works with: Any Windows development language that supports DLL calls

Description: PowerShoW is a DLL that provides image-management tools for manipulating DIB, TIFF, and TARGA bitmap images in 8-, 16-, and 24-bit color. It supports file-format conversions, dithering, scrolling and zooming, and many other image-editing functions. Source code for a Visual BASIC image viewer is included.

ETN Corporation

R.D. 4 Box 659

Montoursville, PA 17754

(717) 435-2202

Q+E Database Library

Function: Database access

Works with: Any development tool or macro language that supports DLL calls

Description: Q+E Database Library provides programmable access to a broad range of databases, including dBASE II, III, and IV; Microsoft SQL Server; Sybase; Oracle; DB2; Netware SQL; Paradox; ASCII text files; and XLS files. It can be called from any development tool that supports external DLL calls. Sample code demonstrating database access techniques for Toolbook, Excel, and Visual BASIC is included.

Pioneer Software

5540 Centerview Drive, Suite 324

Raleigh, NC 27608

(919) 859-2220

Q+E Database/VB

Function: Database access

Works with: Visual BASIC

Description: A Visual BASIC-specific package for creating database applications with dBASE-compatible files, Q+E Database/VB includes custom controls for creating list boxes, combo boxes, text boxes, pictures, check boxes, radio buttons, pushbuttons, and scroll bars that access a database. The visual interface makes it possible to access databases without writing code.

Pioneer Software

5540 Centerview Drive, Suite 324

Raleigh, NC 27608

(919) 859-2220

Quadbase-SQL for Windows

Function: Database access

Works with: Any development language that supports DLL function calls

Description: Quadbase-SQL for Windows is a database engine allowing the development of SQL-capable database applications. It is a DLL with a broad palette of tools for constructing SQL queries, manipulating dBASE III and IV, Foxbase, Clipper, and Lotus 1-2-3 files.

Quadbase Systems, Inc.

790 Lucerne Drive, Suite 51

Sunnyvale, CA 94086

(408) 738-6989

QuickPak Professional for Windows

Function: Custom controls and code library

Works with: Visual BASIC

Description: QuickPak Professional is an extensive collection of Visual BASIC subroutines, functions, and custom controls. It includes enhanced list-box, edit-box, and scroll-bar controls; file-management routines; equivalents for Lotus 1-2-3 statistical and financial functions; array processing; and a variety of utilities to simplify your application's interactions with DOS, Windows, and system hardware. Also included are several demonstration programs illustrating the use of QuickPak tools and complete, commented source code for all the QuickPak routines.

Crescent Software, Inc.

11 Bailey Avenue

Ridgefield, CT 06877

(203) 438-5300

RealSound for Windows

Function: Sound

Works with: Any tool capable of making DLL calls

Description: RealSound for Windows is a DLL system for playing sounds on the computer's internal speaker or on a sound card. It supports all Windows-compatible development tools and includes a manual with special chapters on Microsoft C, Borland C++, and Borland Turbo Pascal for Windows.

RealSound Inc.

4910 Amelia Earhart Drive

Salt Lake City, UT 84116

(801) 359-2900

SQL SoftLink

Function: SQL database access

Works with: Any application supporting DDE

Description: SQL SoftLink provides DDE-capable applications with access to Microsoft's SQL Server.

SQLSoft

10635 N.E. 38th Place, Building 24, Suite B

Kirkland, WA 98033

(206) 822-1287

Superbase Utilities 1

Function: Form editing in Superbase

Works with: Superbase 4

Description: Two utilities are included in this package: one that converts Superbase forms to DML programs, and another that will return the coordinates of any object within a form, and enables moving, resizing, or altering of the object.

Dalco Publishing

808-808 West Hastings Street

Vancouver, British Columbia V6C 1C8

Canada

(604) 687-8808

SuperDialog!

Function: Dialog box design

Works with: Superbase 4

Description: A dialog box designer written in DML, SuperDialog! gives programmers the ability to create dialog boxes using the mouse. The resulting dialog box can then be converted into a stand-alone Superbase program that can be pasted into applications.

Dalco Publishing

808-808 West Hastings Street

Vancouver, British Columbia V6C 1C8

Canada

(604) 687-8808

3-D Widgets #1, #2, #3

Function: Custom controls

Works with: Visual BASIC

Description: These three sets of custom controls for Visual BASIC include enhanced list and combo boxes, bitmapped buttons, 3-D versions of all of Visual BASIC's standard controls, ribbon buttons for creating toolbars, and enhanced menus.

Sheridan Software Systems

65 Maxess Road

Melville, NY 11747

(516) 753-0985

VBAssist

Function: Programming utility

Works with: Visual BASIC

Description: VBAssist smoothes production of Visual BASIC programs with a variety of design tools for creating, aligning, and manipulating controls; the Property Assistant, which displays control properties; and a Visual Basic code manager for saving and retrieving procedure codes.

Sheridan Software Systems

65 Maxess Road

Melville, NY 11747

(516) 753-0985

VB/ISAM MX

Function: Database management

Works with: Visual BASIC

Description: VB/ISAM MX provides a set of functions for the creation, indexing, and searching of ISAM data files. It is not compatible with database or ISAM files created by other data-management packages.

Software Source

42808 Christy Street, Suite 222

Fremont, CA 94538

(510) 623-7854

VB Project Archiver

Function: Program development archiving

Works with: Visual BASIC

Description: VB Project Archiver provides archival project management and version tracking for Visual BASIC programs.

The Young Software Works

300 Mercer Street, Suite 15A

New York, NY 10003

(212) 982-4127

VBTools 2.0

Function: Custom controls, graphics, utilities

Works with: Visual BASIC

Description: VBTools 2.0 includes over 30 custom controls, graphical special effects, and several utility functions.

MicroHelp

4636 Huntridge Drive

Roswell, GA 30075

(404) 552-0565

VBXRef

Function: Programming management

Works with: Visual BASIC

Description: VBXRef creates reports of the properties of forms and controls created in Visual BASIC, and can also cross reference variables and procedures used in the objects.

MicroHelp

4636 Huntridge Drive

Roswell, GA 30075

(404) 552-0565

Visual Architect Series

Function: Custom controls

Works with: Visual BASIC

Description: Visual Architect provides a set of custom controls for Visual BASIC developers, including formatted edit fields; a spreadsheet control; BMP, PCX, and GIF viewers; bitmapped buttons; and a meter control (among others).

Prescription Software

P.O. Box 309

75 Walnut Street

Richmond, OH 43944

(614) 765-4333 (Tamara Calaway)

Whitewater Resource Toolkit 3.5

Function: Resource editor

Works with: Windows 3.0 or higher development applications

Description: The Whitewater Resource Toolkit creates reports of Windows resources, including dialog boxes and menus, and can return information on IDs, locations, and groups for dialog boxes, menus, strings, and accelerators. Also included is a dialog box editor with support for third-party custom controls.

The Whitewater Group

1800 Ridge Avenue

Evanston, IL 60201

(708) 328-3800

Companion Disk Instructions

The Companion Disk in the back of this book includes the source code for the seven projects presented in Chapters 9 through 15, and other programs and utilities that will be useful as you create applications. The following programs are on the disk:

- CW Install, a Windows-based installation program
- Compiled EXE versions of the DocMan and Recycler projects and the VBRUN100.DLL needed to run them
- A set of Visual BASIC files demonstrating the use of all the file management functions in Art Krumsee's DISKSTAT.DLL (DISKSTAT.DLL, which accompanies the Recycler project, is described in Chapter 12)
- A full, working version of WinBatch 3.1 for use with the AutoPrint and Ultimate Notepad project files

Using CW Install

To launch the installation program CW Install, place the Companion Disk in the A or B drive on your PC, and then select the Run item on the File menu in the Windows 3.1 Program Manager or File Manager. Next, type **A:\CWINSTALL.EXE** (or **B:\CWINSTALL.EXE**) in the edit field on the Run dialog box, then click the dialog box's OK button.

Once CW Install loads, use the dropdown combo box labeled "Destination Drive" to indicate the drive on which you wish to install one or more of the project files. CW Install installs each project's files separately, so you can specify different destination drives for different projects if you like.

Next, select the project whose files you wish to install from the list box below the destination drive combo box. Click on the Install Project button to install the selected project's files on the destination drive, or click on the Describe Project button to see more information about the selected project.

When you click the Install Project button, a dialog box will appear that lists the files to be installed and asks you to specify the directory on the destination drive in which you wish to install them. CW Install always suggests a default directory for each project, but you can change that to any other directory. (However, use the default directory whenever possible.) CW Install will create the destination directory if it doesn't already exist.

Note that there are two listings each in the Projects list box for the AutoPrint, Windows Broker, and DocMan projects. The inclusion of two listings for AutoPrint allows you to install either the Norton Desktop for Windows or WinBatch version. In the case of Windows Broker, you can specify different destination directories for the Excel and DynaComm files. For DocMan, you can choose between the Visual BASIC and Ami Pro files.

In addition, the files in some projects need to be split across two or more directories. When you attempt to install one of these projects, CW Install presents you with successive lists of files for each required directory.

When you've finished installing files, click the Exit button to quit CW Install. If you need assistance while using CW Install, click the button labeled Help on CW Install's opening screen.

Project Files

The files for each project are stored in separate directories on the Companion Disk as follows:

Project	Directory
The Ultimate Notepad	\CWINSTAL\9ULTNOT
Who's Who at PC/Computing?	\CWINSTAL\10WHOS
AutoPrint for Windows	\CWINSTAL\11AUTOP
Recycler	\CWINSTAL\12RECYCL
Windows Broker	\CWINSTAL\13WINBRO
DocMan	\CWINSTAL\14DOCMAN
M.M.M.	\CWINSTAL\15MMM

Each project file is described in the following sections.

The Ultimate Notepad

The Ultimate Notepad project adds a range of new functions to Notepad, including a search-and-replace facility, the ability to work with multiple files at once, and a word-count facility. As described in Chapter 9, The Ultimate Notepad consists of just two files, both intended for use with WinBatch:

NOTEPAD.WBT	WinBatch macro file
NOTEPAD.WDF	WinMacro menu definition file

The default destination directory for The Ultimate Notepad is *D:\WB31* (where *D:* indicates the destination drive.)

To use The Ultimate Notepad, make sure that WINMACRO.EXE (included in WinBatch) is running before you launch Notepad.

Who's Who at *PC/Computing*

Who's Who at *PC/Computing* is an employee information graphic that was built using Spinnaker Plus. As described in Chapter 10, Who's Who at *PC/Computing* consists of a single file:

WHONEW.STA	Who's Who application
------------	-----------------------

The default destination directory for Who's Who at *PC/Computing* is *D:\Plus*. You must have Spinnaker Plus installed on your system to use this application.

AutoPrint for Windows

AutoPrint for Windows is a Windows print queue that allows you to designate nearly any document created by a Windows application for printing at a specified time. Two versions are included on the companion disk, one designed for use with Norton Desktop for Windows (NDW), the other for use with WinBatch and the Windows 3.1 File Manager. As described in Chapter 11, AutoPrint for Windows includes five files:

GETFILE.WBT	NDW macro that adds file to print queue
AUTOPRN.WBT	NDW macro to print all files in queue
COPYMAC.WBT	WinBatch version of GETFILE.WBT
AUTOP.WBT	WinBatch version of AUTOPRN.WBT
WINFILE.WDF	WinBatch menu definition file

The default destination directory for the NDW version of AutoPrint for Windows is *D:\NDW*, and the default destination directory for the WinBatch version is *D:\WB31*.

To use the WinBatch version of AutoPrint for Windows, make sure that WINMACRO.EXE (included in WinBatch) is running before you launch File Manager.

Recycler

Recycler is a recycling bin for files that hides files that you drag onto its icon from the Windows 3.1 File Manager. The files in the recycling bin can be restored or deleted at any time. As described in Chapter 12, Recycler includes eight files.

RECYCLER.EXE	Executable version
RECYCLER.MAK	Visual BASIC project file
GLOBAL.BAS	Global declarations file
FORM1.FRM	Form file
DRAGDROP.BAS	Code module
DISKSTAT.DLL	DOS functions DLL used by Recycler
RECEMPTY.ICO	Icon file used by Recycler
RECFULL.ICO	Icon file used by Recycler

The default installation directory for Recycler and its source files is *D:\VB\RECYCLER*. The default directory for *DISKSTAT.DLL* is *D:\WINDOWS\SYSTEM*.

You'll need to install the Visual BASIC runtime DLL (*VBRUN100.DLL*) included on the Companion Disk to use *RECYCLER.EXE*, and you'll need the full Visual BASIC development environment to edit Recycler's source files.

Windows Broker

The Windows Broker project is a stock-trading and portfolio analysis system built using Microsoft Excel and FutureSoft Engineering's DynaComm. As described in Chapter 13, Windows Broker consists of the following files:

BROKER1.XLW	Excel workspace file
BROKER1.XLM	Excel macro file
BROKER1.XLS	Excel worksheet file
LOTS.XLS	Daily stock price worksheet
IBM.XLS	Daily stock price worksheet
BROKER.DCS	DynaComm settings file
BROKER1.DCT	Compiled DynaComm script file
BROKER1.DCP	DynaComm script source code file

The default destination directory for the Excel files is *D:\EXCEL*. The default directory for the DynaComm settings file is *D:\DYNACOMM\DCS*, and for the DynaComm script files it is *D:\DYNACOMM\DCP*.

DynaComm is very fussy about its use of directories, so changing this default is not recommended.

You'll need Excel 3.0 or 4.0 to use the Excel portion of the Windows Broker application, and DynaComm 3.0 (or higher) to use the DynaComm scripts.

DocMan

DocMan is a document management system built in Visual BASIC and the Ami Pro macro language that allows you to organize and access Ami Pro documents using 120-character document titles. As described in Chapter 14, DocMan consists of these files:

DOCMAN.EXE	Executable version of DocMan
DOCMAN.MAK	Visual BASIC project file
DCGLOBAL.BAS	Global declarations file
FORM1.FRM	Form file for opening banner
DOCMAN2.FRM	Form file for DocMan's main screen
FINDDLG.FRM	Form file for Find Document dialog box
ACTIONS.FRM	Form file for floating tools palette
ABOUTDLG.FRM	Form file for About DocMan dialog box
GLOBCODE.BAS	Visual BASIC global code module
DOCMAN.DAT	Sample document data file
DMKEYS.DAT	Sample keywords file
AUTOEXEC.SMM	Ami Pro AutoExec macro
AUTONEW.SMM	Ami Pro AutoNew macro
SAVEMAC.SMM	Ami Pro File Save functions macro
DMINFO.SMM	Ami Pro Call DocMan macro

The default destination directory for DocMan's Visual BASIC files is *D:\VB*, and for its Ami Pro files, *D:\AMIPRO\MACROS*. (Note, if you already have Ami Pro macros called *AUTOEXEC.SMM* or *AUTONEW.SMM*, you should rename them before installing DocMan's Ami Pro macros, and then edit the DocMan Ami Pro macros after installing them to include the routines from your existing macros with those names. Otherwise,

your existing AUTOEXEC.SMM and AUTONEW.SMM macros will be overwritten during the installation process.)

To run DOCMAN.EXE, you'll need to install VBRUN100.DLL from the Companion Disk. You must have Ami Pro to use the Ami Pro macros. The full Visual BASIC environment is required to edit the DOCMAN.MAK project files.

M.M.M.: the MCI Mail Manager

M.M.M.: the MCI Mail Manager is a full-featured electronic mail management system written using the DynaComm script language, as described in Chapter 15. The version on the Companion Disk features one change from that described in the text: a number of utility routines that were repeated in several modules have been moved to a shared-code module called COMMON.DCP. With this change, M.M.M. now consists of 18 files:

AUTOMCI.DCP	Source code for M.M.M.'s main module
AUTOMCI.DCT	Compiled code for main module
COMMON.DCP	Source code for common utility routines
COMMON.DCT	Compiled code for common utility routines
EM.DCP	Source code for communications module
EM.DCT	Compiled code for communications module
ONLINE.DCP	Source code for terminal mode module
ONLINE.DCT	Compiled code for terminal mode module
PM.DCP	Source code for phone book module
PM.DCT	Compiled code for phone book module
TM.DCP	Source code for message editor module
TM.DCT	Compiled code for message editor module
BLANK.DCM	Template file for new messages
FORWARD.DCM	Template file for forwarded messages
MAILSET.DCS	DynaComm settings file
MMM2.BMP	Bitmap file displayed by M.M.M

PRIVATE.PBK	Sample private phone book data file
PUBLIC.PBK	Sample public phone book data file

The default installation directory for the source and compiled code files is *D:\DYNACOMM\DCP*, and the default directory for the remaining files is *D:\DYNACOMM\DAT*. Given DynaComm's extreme fussiness about file storage locations, changing these defaults is not recommended.

In order to use M.M.M. you'll need version 3.0 or higher of DynaComm.

Other Files

The Companion Disk includes several other files in addition to those for the book's seven projects. These include *VBRUN100.DLL* (the Visual BASIC runtime DLL), *DISKSTAT.EXE* (an executable file demonstrating other functions of the *DISKSTAT.DLL*) and its Visual BASIC source code files, and WinBatch 3.1. These are stored in the following directories on the Companion Disk:

File	Directory
<i>VBRUN100.DLL</i>	Root directory
<i>DISKSTAT.EXE</i> (and source code)	<i>\DISKSTAT</i>
WinBatch 3.1	<i>\WINBATCH</i>

Visual BASIC Runtime DLL (*VBRUN100.DLL*)

VBRUN100.DLL is required to run *CWINSTAL.EXE*, *RECYCLER.EXE*, and *DOCMAN.EXE*. Its default installation directory is *D:\WINDOWS\SYSTEM*.

DISKSTAT.EXE

DISKSTAT.EXE is a demonstration program for the various file manipulation functions provided to Visual BASIC by *DISKSTAT.DLL*. *DISKSTAT.EXE* was written by Art Krumsee, the developer who wrote *DISKSTAT.DLL*. The *\DISKSTAT* directory contains the following files:

<i>DISKSTAT.EXE</i>	Demonstration program for <i>DISKSTAT.DLL</i>
<i>DISKSTAT.MAK</i>	Visual BASIC project file for <i>DISKSTAT.EXE</i>

DISKSTAT.BAS	Global declarations file for DISKSTAT.EXE
DISKSTAT.FRM	Form file for DISKSTAT.EXE

The default installation directory for DISKSTAT.EXE and its source code files is *D:\VB\DISKSTAT*.

VBRUN100.DLL and the DISKSTAT.DLL from the Recycler project must be installed on your system before you can run DISKSTAT.EXE. You'll need the full Visual BASIC development environment to edit its source code files.

WinBatch 3.1

The WinBatch 3.1 files are stored in a compressed format on the Companion Disk. CW Install will automatically decompress them as it copies them to the destination drive. Once the files have been copied, run the WinBatch setup program, WSETUP.EXE, using the Program Manager or File Manager File Run command, to complete installation of WinBatch 3.1 on your system. WSETUP.EXE will copy the WinBatch files to a new directory, so once you have run WSETUP.EXE, you can delete the directory to which CW Install copied the files.

The default destination directory for the WinBatch files is *D:\WBSETUP*.

If you prefer to copy the WinBatch 3.1 files to your destination disk manually rather than use the CW Install routine, you can use the program EXPAND.EXE, which you'll find in the root directory of the Companion Disk, to decompress them. To do so, shell to DOS or exit from Windows, create a directory called WBSETUP in the root directory of your destination disk to store the expanded files, then activate the drive containing the Companion Disk. From its root directory, type the following command:

```
EXPAND \WINBATCH\*.* D:\WBSETUP
```

Return to Windows and run WSETUP.EXE.

Please note that WinBatch 3.1 is a shareware program. If you use it beyond a brief trial period, you are obligated to register it with Wilson WindowWare and pay a shareware fee.

INDEX

A

- AccSys package, 521
- active status line, 158-159
- Agility/VB package, 521
- Ami Pro (Lotus). *See also* DocMan
 - Clipboard support, 59
 - contacting the company, 73
 - macro language, 72-73
 - status bar font menu, 158
- Ami Pro Macro Developer's Kit, 522
- AND operator, 26-27
- append, 23, 59
- application design. *See also individual applications*
 - and application purpose, 111-113
 - defining input requirements, 120-125
 - defining internal processing, 133-135
 - defining variables, 125-129
 - drawing a flowchart, 115-120
 - links to other applications, 121-122
 - nuts and bolts of, 110-135
 - picking development tools for, 113-114
 - planning data structures, 125-133
 - usability guidelines, 96-108
- application-development tools
 - and application control, 59
 - for AutoPrint for Windows, 241
 - and Clipboard support, 59
 - for DocMan, 338-339
 - evaluating, 58-69
 - for M.M.M., 404-405
 - and multitasking facilities, 58-63
 - selecting, 54-93, 113-114
 - types of, 70
 - for The Ultimate Notepad, 181-182
 - for Who's Who at PC Computing, 210-211
 - for Windows Broker, 295-296
- application macro languages. *See* Macro languages
- application window features, 140-144
 - application workspace, 143
 - control menu box, 140
 - document windows, 143-144
 - menu bar, 140-143
 - minimize and maximize buttons, 140
 - scroll bars, 143
 - title bar, 139-140
- application workspace, 143
- arrays, 28-29
- Autodesk Animation Player for Windows, 522
- AutoPrint for Windows, 57, 238-264
 - application design, 111-135, 240-241
 - AUTOPRN.LST, 133
 - AUTOPRN.WBT batch file, 248-254
 - ExitRoutine routine, 251-252
 - GetFile routine, 250
 - PrintFile routine, 250
 - Start routine, 249-250
 - TitleCheck routine, 251
 - AUTOP.WBT batch file, 258-264
 - ConfirmTime routine, 260-262
 - EndLoop routine, 264
 - GetWindTitle routine, 263
 - NextFile routine, 264
 - PrintFileLoop routine, 263
 - StartNow routine, 262-263
 - Start routine, 260
 - TimeCheck routine, 262
 - WentFine routine, 264

- COPYMAC.WBT batch file, 255-258
 - AssocErrJump routine, 258
 - Main routine, 256-257
 - WrongApp routine, 258
 - data requirements, 123
 - data structures, 132-133
 - development tools, selecting, 241
 - files to print, obtaining, 123-124
 - flowchart for, 117-119
 - functionality, 241-243
 - Get File dialog box, 247
 - GETFILE.WBT batch file, 243-248, 252-254
 - AssocErrJump routine, 248
 - initial flowchart for, 118
 - later flowchart for, 119
 - Main routine, 244-247
 - Start routine, 244
 - WrongApp routine, 247-248
 - impetus behind, 239-240
 - Launch List menu, 252
 - Norton Desktop version, 243-254
 - obtaining a start time, 124-125
 - Program Manager icon for, 260
 - project files, 538
 - START.DLG dialog box, 259
 - WinBatch version, 254-264
 - WrongApp error dialog box, 258
- B**
- backgrounds (in Spinnaker Plus), 211
 - BASICs for Windows, 85-89
 - batch languages, 46-47, 80-84. *See also*
 - WinBatch (Wilson WindowWare)
 - Batch Runner, 241
 - Bridge Batch, 81-82
 - Bridge Toolkit, 81-82
 - and linking applications, 48-50
 - Norton Desktop for Windows, 83
 - PubTech BatchWorks, 83-84
 - batch printing utility. *See* AutoPrint for Windows
 - Batch Runner, 241
 - beta testing, 174
 - Bonner's usability guidelines, 99-108
 - design for reliability, 106
 - design for the user's convenience, 107-108
 - don't delay experienced users, 107
 - don't overwhelm new users, 106-107
 - don't surprise the user, 102-104
 - finish the job, 105
 - fit applications into the current work flow, 100-101
 - improve on existing methods, 101-102
 - keep the user informed, 103-104
 - make applications open-ended, 105-106
 - make the application wait, 104
 - try to delight the user, 104-105
 - Boolean expressions, 25-27
 - Boolean operators, table of, 26
 - Bricklin, Dan, 3-4
 - Bridge Batch, 81-82
 - Bridge Toolkit, 81-82
 - Bridgit package, 522
 - Broker. *See* Windows Broker
 - buttons
 - command, 144-145
 - radio (option), 146-147
 - ButtonTool package, 523
- C**
- cards (in Spinnaker Plus), 211
 - cascading menus, 142-143

- chaining statements, 29
 - ChartBuilder for Visual BASIC, 523
 - check boxes, 145-146
 - Clipboard (Windows), 9, 59, 122
 - combination boxes, 155
 - command buttons, 144-145
 - commands, 14-16
 - COMMDLG.DLL (Windows) file, 162
 - comments (nonexecutable statements), 27-28
 - Companion Disk instructions, 536-543
 - CompuServe, and Windows Broker, 296, 320, 323-325, 328
 - concatenation, 24
 - control array, 282
 - control menu box (application window), 140
 - cost and performance issues, 42
 - cross-systems applications development, 85
 - CrossTalk for Windows, 74, 404
 - Custom Control Factory, 523
 - custom controls, 50-51
 - cut-and-paste programming, 40-53
 - CW Install, 536-537
- D**
- data formats, disk-based, 129-132
 - data presentation. *See* Who's Who at PC/Computing
 - data structures
 - AutoPrint for Windows, 132-133
 - planning, 125-133
 - dbFast/Win, 74-75
 - DDE (dynamic data exchange), 4, 59, 61, 121-122, 295
 - debugging, 169-174
 - decision making (program control), 29-30
 - designing applications. *See* application design
 - DESQview, 5-6
 - detail flowcharts, 115
 - development tools. *See* application-development tools
 - dialog box editors, 66-67
 - dialog boxes, 64-66, 161-162
 - Dialoger package, 524
 - disk-based data formats, 129-132
 - Disk (Companion Disk) instructions, 536-543
 - DISKSTAT.EXE file, 542-543
 - Distinct TCP/IP for Windows, 524
 - DLL support, 62-63
 - DocMan, 57, 336-400
 - ABOUTDLG.FRM, 371-372
 - _Command1_Click procedure, 372
 - Form_Paint procedure, 371-372
 - About DocMan dialog box, 371
 - ACTIONS.FRM, 362-364
 - button routines, 363-364
 - Command1_Click procedure, 363
 - Command1_DragDrop procedure, 363-364
 - Command1_KeyUp procedure, 364
 - Form_Load procedure, 362
 - Ami Pro Document Description dialog box, 342
 - Ami Pro interaction with, 393-399
 - application framework, 343
 - DCGLOBAL.BAS module, 343-346
 - constant declarations, 343-344
 - data-storage declarations, 345-346
 - external-function declaration, 344-345

- variable declarations, 346
- Delete File confirmation box, 379
- development tools, selecting, 338-339
- DOCMAN2.FRM, 348-362
 - Command1_Click procedure, 352-353
 - Command2_Click procedure, 354
 - Description_Change procedure, 354
 - Form_LinkExecute procedure, 355-358
 - Form_Load procedure, 348-350
 - Form_Paint procedure, 350
 - Form_Resize procedure, 359
 - Form_Unload procedure, 359
 - general and loading routines, 348-350
 - menu item routines, 360-362
 - Text1_Change procedure, 354
 - Text1_GotFocus procedure, 355
 - Titles_Click procedure, 351
 - Titles_GotFocus procedure, 351
 - Titles_LostFocus procedure, 351-352
 - Titles_MouseMove procedure, 352
 - user-action routines, 350-359
- FileErrors message box, 383
- FINDDL.G.FRM, 365-371
 - Command1_Click procedure, 368
 - Command2_Click procedure, 370
 - Command3_Click procedure, 370
 - Command3_DragDrop procedure, 370-371
 - event procedures, 368-371
 - Form_Load procedure, 365-366
 - Form_Paint procedure, 366
 - Form_Unload procedure, 366-368
 - general and loading routines, 365-368
 - List1_Click procedure, 368-369
 - List1_Dblclick procedure, 369
 - List1_MouseMove procedure, 369
 - Text1_DragDrop procedure, 369-370
- FindDlg screen, 340-342
- Find Document dialog box, 341
- FORM1.FRM, 346-347
- functional requirements, 338
- GLOBCODE.BAS module, 372-392
 - AddKeys procedure, 384-386
 - CenterForm procedure, 373
 - CleanUp procedure, 387-388
 - ClearFields procedure, 381
 - DeleteFile procedure, 378-380
 - DeleteRecord procedure, 380-381
 - ExitDocMan procedure, 387
 - FileErrors procedure, 382-383
 - FileOpener procedure, 381-382
 - FindRecord procedure, 388-390
 - Frame procedure, 373-374
 - GetFile procedure, 375
 - LaunchApp procedure, 376-377
 - Loaded procedure, 376
 - NewFile procedure, 377-378
 - OpenDoc procedure, 374-375
 - PrintFile procedure, 378
 - ReadSelectedRecord procedure, 383-384
 - RestoreApp procedure, 376
 - Sub GetPath procedure, 390-391

- TestField procedure, 390
 - TestLength procedure, 391-392
 - WaitSecs procedure, 377
 - WriteChangedRecord procedure, 384
 - WriteKeyFields procedure, 386-387
 - message boxes, 390,392
 - modified Ami Pro File menu, 394
 - Next Match button enabled, 389
 - OpenDM screen, 339-340
 - form and tools palette, 339
 - pull-down menus, 340
 - opening banner, 346, 347
 - opening moves, 337
 - project files, 540-541
 - text highlights by input focus, 356
 - Titles list box icon, setting, 353
 - user interface, 339-343
 - and Visual Basic, 338
 - documentation, 175-176
 - document windows, 143-144
 - DoEvents loop (Visual BASIC), 273
 - DO-LOOP, 32-33
 - DOS attribute control, 268, 274
 - DOS Hidden attribute, 268
 - DOS limitations, 5-6
 - double-precision real variables, 17
 - drag-drop functions (Recycler), 277-278
 - drag-drop messages, handling, 271-274
 - drop-down combination box, 155
 - drop-down list box, 155
 - DynaComm (FutureSoft Engineering), 44, 48-49, 295, 404-405. *See also* M.M.M.: the MCI Mail Manager; Windows Broker Communications dialog box, 432
 - contacting the company, 76
 - data needed by, 302
 - dialog boxes, 64-65
 - macro language, 75-76
 - WAIT STRING command, 168
 - dynamic link libraries (DLLs), 50-52, 266-293
 - dynamic menu items, 141-142
- E**
- edit boxes, 151-152
 - EditTool package, 524
 - elements of program control, 29-36
 - elements of programming, 13-29
 - email. *See* M.M.M.: The MCI Mail Manager
 - EMS (electronic mail service)
 - address, 413
 - endless loops, 33
 - envelope (message), 405
 - error handling, 36-37
 - Excel (Microsoft), 48
 - contacting the company, 78
 - DDE support, 62
 - dialog box editor, 67
 - INITIATE command, 62
 - macro language, 77-78
 - OPEN command, 62
 - PARSE command, 308-309
 - REQUEST command, 62
 - TERMINATE command, 62
 - experimentation, 38. *See also* Windows Broker
 - expressions, 13, 23-27
 - Express (Lotus), 43, 403
 - extended-selection list boxes, 153
- F**
- FastData package, 525
 - file formats, 69
 - File Manager (Windows). *See* Windows File Manager

finger knowledge, 7
 flowchart iterations, 116-118
 flowcharts, drawing, 115-120
 flowchart symbols, 116
 folders (message lists), 406
 FOR-NEXT loop, 20, 31, 128
 functions, 20-23

G

GFA-BASIC for Windows, 85-86
 global scope, 35
 global variables, defining, 128-129
 graphical hypertext products, 90-93
 Graphics Server SDK package, 525
 grayed check boxes, 145-146
 graying out menu items, 141
 group boxes, 147
 guessing game application, 126-129

H

handlers (in Spinnaker Plus), 211
 handles, 63
 horizontal scroll bars, 143
 hot links, 61, 121
 Hypercard-like tools, 210

I

icon bars, 157
 icons, 148-150
 IF-THEN-ELSE statements, 29-30
 IF-THEN statements, 29
 ImageMan package, 525
 informational icons, 149-150
 INI files, 131-133
 initializing variables, 18-19
 input focus, 60
 input/output, 33-34
 input/output commands, 15
 input requirements, defining, 120-125

installation, documenting, 175
 integer variables, 17
 internal processing, defining, 133-135
 interrupt handling, 7
 ISAM, 521
 iterative prototypes, 166-167

K

keyboard interface, 159-160
 keyboard navigation, 159-160
 keyboard operation, testing, 170-171
 keyboard shortcuts, 160
 key (field), 131
 key name (field), 131
 keyword searches, 340-341

L

learning to program, 38-39
 libraries, dynamic link, 50-52, 266-293
 linking applications, 48-50, 121-122
 list boxes, 152-154
 long-integer variables, 17
 looping, 128
 loops, 19-20, 30-33, 272-273
 Lotus Ami Professional. *See* Ami Pro (Lotus)
 Lotus Express, 43, 403
 Lotus 1-2-3 for Windows, 76
 low memory conditions, testing for, 172
 low system-resource conditions, testing for, 172-173

M

Macintosh Trash icon, 148
 macro file formats, 129
 macro languages, 44-46, 70-80
 Ami Pro, 72-73. *See also* DocMan
 Crosstalk for Windows, 74
 dbFast/Win, 74-75

- DynaComm, 75-76. *See also*
 - M.M.M.: the MCI Mail Manager; Windows Broker
- Excel, 77-78. *See also* Windows Broker
- and linking applications, 48-50
- Lotus 1-2-3 for Windows, 76
- Object Vision 2.0, 78-79
- SuperBase4, 79-80
- Word for Windows, 72, 78-79
- macro sheets, 77, 304
- Magnet, 323
- mathematical functions, 21
- MCI Mail Manager. *See* M.M.M.: The MCI Mail Manager
- menu bar (application window), 140-143
- menu editors, 66-67
- menu items, dynamic, 141-142
- menus, 66
- message boxes, 64, 149-150
- message-handling loops, 272-273
- MicroHelp Communications Library, 526
- MicroHelp Muscle, 526
- MicroHelp Network Library, 526
- Microsoft Excel. *See* Excel (Microsoft)
- Microsoft LAN Manager Toolkit for Visual BASIC, 527
- Microsoft Word for Windows Developer Kit, 528
- Microsoft Word for Windows. *See* Word for Windows (Microsoft)
- minimize and maximize buttons (application window), 140
- M.M.M.: The MCI Mail Manager, 43-44, 402-519
 - About M.M.M. dialog box, 439
 - Account menu, 431-432
 - Add Name to... dialog box, 515
 - addresses, editing, 413
 - AUTOMCI.DCP script, 416-470
 - About routine, 439
 - Access_Num routine, 435-436
 - Account_Data routine, 465
 - account setup routines, 462-467
 - Ans_Button routine, 428
 - Ans_Set routine, 459
 - Auto_Freq routine, 439-440
 - Auto_Set routine, 436-437
 - Back_Up routine, 445
 - Check_Del routine, 452
 - Clear_Marks routine, 452-453
 - Close_and_Clear routine, 443-444
 - Close_Em routine, 444
 - Code routine, 467-470
 - Decode routine, 468
 - Delete routine, 455
 - Dialog_Update routine, 427
 - Edit_Out routine, 451-452
 - Edit routine, 466-467
 - the end of Main, 427
 - the end of Menu, 433
 - Export routine, 460
 - FastRest routine, 450-451
 - File_It routine, 466
 - FindRoutine routine, 458-459
 - For_Button routine, 428
 - For_Set routine, 459-460
 - Get_Filename routine, 458
 - Get_List routine, 445-446
 - global settings and variables, initializing, 416-421
 - Main routine, 421-427
 - Mark_Del routine, 453
 - menu-support routines, 433-441

- message-handling commands, 423-425
- message-handling routines, 456-461
- Move routine, 448-450
- New routine, 463-464
- obtaining data, 420-421
- offline and quit commands, 426-427
- online commands, 425-426
- Pick routine, 434
- Print_Mess routine, 460-461
- Pub_Book routine, 434-435
- Purge_Marked routine, 454-455
- Read routine, 456-457
- ReadString routine, 446
- Rec routine, 428-429
- Restore routine, 437-438
- Saveall routine, 455-456
- Save_Table routine, 444-445
- setting up directories, 417-418
- Set Up Menu routine, 429-433
- SetupMMM routines, 468-470
- Setup routine, 463
- Slots routine, 440-441
- Sort by and View commands, 423-424
- Sort routine, 446-447
- sort and view commands, 422
- Stat_Check routine, 461-462
- Stats routine, 447-448
- Table_Def_And_Load routine, 443
- table-handling routines, 442-446
- Table_Save routine, 444
- Tables routine, 442-443
- Ver_Cir routine, 429
- View_Check routine, 433-434
- welcome-message routines, 461-462
- Welcome routine, 462
- WriteString routine, 446
- AutoMCI Frequency dialog box, 440
- AutoMCI session status message, 509
- AutoMCI Settings dialog box, 436
- capabilities of, 405-407
- Communications menu, 432
- creating and editing messages, 405-406
- development tool for, 404-405
- drawing the dialog box, 421-422
- drop-down combo boxes, 409-410 and Dynacomm, 404-405
- Edit Account dialog box, 467
- EMAIL.DCP, 489-511
 - Auto_Dial_Seq routine, 506-507
 - Auto_Main routine, 507-508
 - Auto_Quit routine, 510
 - Auto_Retrieve routine, 506
 - BackUpFiles routine, 510
 - Date_Fix routine, 503-504
 - Delete_File routine, 501
 - EMS routine, 498
 - Err routine, 499
 - Err1 routine, 499
 - Err2 routine, 499-500
 - Get_cc routine, 497-498
 - Get_Sub routine, 500
 - Get_To routine, 496-497
 - initializing global variables, 489-490
 - Loginerror routine, 505
 - Login routine, 491
 - Logout routine, 504-505
 - Looper routine, 493-494
 - Loop routine, 492

- Loop3 routine, 508
- message-reception routines, 501-510
- message-transmission routines, 490-501
- Pickup routine, 501
- Prepare_For_Next_Message routine, 496
- PR_Loop routine, 501-502
- Process_Incoming_Mess routine, 502-503
- Prompts routine, 492-493
- Restart routine, 508-509
- Save_and_Close routine, 511
- Send_Body routine, 495
- Send_Handling routine, 495-496
- Send_Loop routine, 494-495
- Send_Message routine, 493
- Send_Recv routine, 490-491
- Send_To routine, 497
- standard library routines, 510-511
- Table_Load routine, 510-511
- Table_Write routine, 505-506
- Time_Set routine, 509-510
- End_It dialog box, 488
- File menu, 430
- functionality, 407-416
- function key definition screen, 517
- Mailboxes screen, 168, 408-410
 - action routines, 446-456
 - support routines, 427-429
- Mailslot Names dialog box, 441
- Message Addressing dialog box, 64-65, 415, 473
- message-composition routines, 415-416, 471-488
- Message Editor, 416, 486
- message-handling options, 411
- messages, organizing and managing, 406-407
- M.M.M. menu, 430-431
- Move Options dialog box, 450
- New Account dialog box, 465
- offline options, 410-411
- ONLINE.DCP module, 515-518
 - Capture routine, 516-518
 - Online routine, 516
 - Print routine, 518
 - terminal routines, 516-518
- on-line options, 410
- Phonebook Management dialog box, 413, 513
- phonebook management routine, 412-414
- PM.DCP module, 511-515
 - Add_Edit_Dialog routine, 513-514
 - Add_Name routine, 512
 - Delete_Name routine, 514-515
 - Edit_Book routine, 513
 - Main_Phon routine, 512
 - Phon_Man routine, 511-512
 - Up_Phone routine, 515
- Print Message command, 503
- Process Message dialog box, 487
- project files, 541-542
- prompt from SetupMMM routine, 470
- prototyping, 167-169
- Public phonebook location, 412
- Purge Message dialog box, 453
- Restore Backups dialog box, 438
- sample envelope file, 475
- Set Up menu, 412
- sort menu, 409
- Statistics dialog box, 411
- Terminal session, 517

- TM.DCP module, 470-489
 Add_To routine, 479
 Answer_Message routine, 481-482
 calling the TM module, 470-471
 Change_Book routine, 481
 Change_List routine, 480-481
 Compose_Message routine, 485
 Cut_Name routine, 480
 Done routine, 482
 Edit_Name routine, 479-480
 Edit_Phbk routine, 473-475
 Edit_Set routine, 474
 End_It routine, 487-488
 Get_Date routine, 475-476
 Get_Name routine, 477-478
 message-creation routines, 471-488
 More_Tos routine, 483
 New_Mess routine, 471-472
 Pad routine, 488-489
 Parse_Name routine, 478
 Process_Message routine, 485-487
 Process routine, 482-483
 ReadInteger routine, 489
 Shorten routine, 476, 477
 To_Dialog routine, 472-473
 Update routine, 479
 utility routines, 488-489
 Write_CC_Fields routine, 483-484
 WriteInteger routine, 489
 Write_Subject_and_Handling routine, 484-485
 transmitting and receiving messages, 406
 user interface, 407-416
 view menu, 410
 welcome message, 408
 mouse and keyboard operation, testing, 170-171
 multidimensional arrays, 28-29
 multiple-application processes, 7-8
 multiple-document interface (MDI), 143-144
 multiple-selection list boxes, 153
 multitasking under Windows, 5-8
 designing for, 8-11
 and selecting tools, 58-63
 shared facilities, 9-10
 shared interface, 10-11
 shared processing, 8-9
- N**
 named subroutines, 34-36
 network operation, testing for, 172
 Norton Desktop version of Autoprint for Windows, 241, 243-254
 Norton Desktop for Windows, 83, 241. *See also* AutoPrint for Windows
 contacting the company, 83
 Print dialog box, 244
 Scheduler utility, 241, 253
 Notepad. *See* Ultimate Notepad, The
 NOT operator, 27
 numeric expressions, 24
 numeric operators, table of, 24
- O**
 object linking and embedding (OLE), 42, 122
 ObjectScript, 86-87
 ObjectView, 87
 Object Vision 2.0, 78-79
 on-line help, 176
 operators, 24, 26
 option buttons, 146-147
 OR-based keyword searches, 341

originality vs. standards, 11
OR operator, 26-27
outline flowcharts, 115

P

parameters, 14
parse, 37
Pascal for Windows, 90
PC Comnet DLL package, 529
PCContact, 323
PDQComm for Windows, 529
performance and cost issues, 42
Plus. *See* Spinnaker Plus
portfolio-analysis application. *See*
Windows Broker
PowerLibW 3.1 package, 529
PowerShow 3.2 package, 530
predefined functions, 21
presenting data. *See* Who's Who at
PC Computing
printing utility. *See* AutoPrint for
Windows
Print Manager (Windows), 240
program control elements, 29-36
program execution, 15
programming
 basics of, 12-39
 cut-and-paste, 40-53
 elements of, 13-29
 learning, 38-39
project files (Companion Disk),
 537-542
prototyping, 165-169
PubTech BatchWorks, 83-84

Q

Q&E Database Library (Pioneer
 Software), 51-53
Q+E Database/VB, 530
Quadbase-SQL for Windows, 531

QuickPak Professional for Windows,
 531
Quick*Way (Quick & Reilly), 296,
 325-333

R

radio buttons, 146-147
random-access files, 33, 131
Realizer package, 88-89
RealSound for Windows, 531
real variables, 17-18
Recycler, 266-293
 About Recycler message box, 287
 application framework, 274
 basic operations, 267
 bin contents in File Manager, 268
 capabilities and limitations, 269-271
 confirmation box for a file delete,
 284
 defining requirements, 271-274
 development tool, selecting, 271-
 273
DRAGDROP.BAS, 288-293
 CheckForDir function, 291
 FixSize subroutine, 292-293
 FixTitle subroutine, 291-292
 HideFile function, 290-291
 main subroutine, 288-290
Erase All confirmation box, 285
File menu, 269
file tracking, 270
FORM1.FRM, 280-288
 Command1_Click procedure,
 282-284
 event procedures, 282-288
 FileAbout_Click procedure,
 287
 FileEmpBin_Click procedure,
 284-285
 FileExit_Click procedure, 287

- FileResAll_Click procedure, 286
 - Form_Resize procedure, 286-287
 - Form_Unload procedure, 287-288
 - general procedures, 282
 - GLOBAL.BAS, 275-280
 - calling DISKSTAT.DLL, 278-279
 - calling the drag-drop functions, 277-278
 - function and subroutine definitions, 276-279
 - global constant and variable definitions, 279-280
 - type definitions, 275-276
 - icon, 267
 - list box, 269
 - message loops, 272
 - messages, 270
 - project files, 538-539
 - Restore All confirmation box, 286
 - and subdirectories, 269-270
 - user interface, 267-270
 - and Visual BASIC, 272-273
 - window when the bin is empty, 267
 - revision marking (Ami Pro), 72
 - ribbons, 157-158
 - running in the background, and testing, 173
 - runtime errors, 36
- S**
- SAA Common User Access Guide*, 156-157
 - SAA standard, 156-157
 - scope, 35
 - screen design, 68-69
 - screen editors, 68
 - screen pages, 296
 - scroll bars (application window), 143
 - sequential-access files, 33, 130
 - shareware, 70
 - single-precision real variables, 17
 - single-selection list boxes, 153
 - Software Development Kit (Windows SDK), 56, 172
 - Spinnaker Plus, 91-92, 210. *See also* Who's Who at PC/Computing
 - application-design issues, 211-214
 - Button Info dialog box, 213
 - contacting the company, 92
 - editing a script, 226
 - limitations of, 214
 - Object Properties dialog box, 228
 - Script window for a button, 214
 - three-dimensional frames, 215
 - SQL databases, 52
 - SQL SoftLink package, 532
 - stacking Notepad windows, 190
 - stacks (in Spinnaker Plus), 211-212
 - standard user-interface controls, 144-155
 - check boxes, 145-146
 - command buttons, 144-145
 - common extensions to the standard, 157-159
 - dialog boxes, 162
 - edit boxes, 151-152
 - group boxes, 147
 - icons, 148-150
 - list boxes, 152-155
 - radio buttons, 146-147
 - static text controls, 150-151
 - statements, 13
 - static text controls, 150-151
 - status lines, active, 158-159
 - stock-trading system. *See* Windows Broker

- strategic projects, 57
- string expressions, 24-25
- string functions, 21-22
- string variables, 17
- subroutines, named, 34-36
- SuperBase4 package, 79-80
- Superbase Utilities 1 package, 532
- SuperDialog! package, 532

T

- tactical projects, 57
- TCP/IP, 521, 524
- testing and debugging, 169-174
- text files, 33, 130-131
- third-party code, 50-52
- 3-D Widgets package, 533
- title bar (application window), 139-140
- Toolbook (Asymmetrix), 92-93, 210
- tools. *See* application-development tools
- transactions (of database changes), 52
- Trash icon, 148
- troubleshooting, documenting for, 175
- TrueType, 171-172
- Turbo Pascal for Windows, 90
- twips, 362
- type-conversion functions, 22
- type mismatch error, 19

U

- Ultimate Notepad, The, 43, 180-207
 - Control menu, 184
 - development tools, selecting, 181-182
 - “File Already Exists” warning, 195
 - File Opener dialog box, 186
 - Find dialog box, 198
 - Insert File dialog box, 192

- NOTEPAD.WBT, 185-206
 - AutoIndentSub routine, 199-200
 - ClowerSub routine, 200-201
 - CupperSub routine, 200-201
 - dialog box routines, 204-206
 - DummySub routine, 195
 - Ender routine, 204
 - FastCountSub routine, 201-204
 - InsertWrite routine, 206
 - introductory lines, 185-186
 - MergeSub routine, 191-193
 - OpenerSub routine, 186-189
 - OpenerWrite routine, 205-206
 - OpenTwoSub routine, 189-191
 - ReplaceSub routine, 195-199
 - ReplaceWrite routine, 206
 - SaveselSub routine, 193-195
 - subroutine macros, 186-204
- NOTEPAD.WDF, 183-185
- project files, 537
- prompt for a filename, 194
- Replace dialog box, 196
- search string not found message, 198
- setting objectives, 182-183
- stacking Notepad windows, 190
 - and WinBatch, 182
- user-defined functions, 22-23
- user errors, 37
- user interface
 - advantages of Windows, 138-139
 - application window features, 139-144
 - assigning keystroke combinations, 137
 - facilities, 63-69
 - implementing, 136-163

standard Windows, 138-139
 user interface controls. *See* standard
 user interface controls
 user testing, 174

V

variables, 16-20
 defining, 125-129
 global, 128-129
 initializing, 18-19
 variable scoping, 35-36
 VBAssist package, 533
 VB/ISAM MX package, 533
 VB Project Archiver package, 534
 VBTools 2.0 package, 534
 VBXRef package, 534
 vertical scroll bars, 143
 video resolutions, testing with
 various, 171
 Visual Architect Series, 535
 Visual BASIC (Microsoft), 88-89,
 272-273, 338. *See also*
 DocMan; Recycler
 APPACTIVATE command, 60
 application control, 59-60
 contacting the company, 89
 custom controls, 50-51
 defining variables, 126-129
 DLL support, 52, 63
 form with button and list box
 controls, 280
 Menu Design window, 281
 menu editor, 67
 RANDOMIZE statement, 126
 runtime DLL files, 542
 screen editor, 68
 SENDKEYS command, 60
 SHELL command, 59
 using API functions within, 60

W

WHILE-WEND loops, 31-32
 Whitewater Resource Toolkit 3.5, 535
 Who's Who at PC/Computing,
 208-236
 card-naming conventions, 219-220
 Departments pop-up menu, 232
 the development tool, selecting,
 210-211
 drawing the interface, 212-214
 Floor Plan screen, 225-230
 background scripts, 226-227
 buttons, 230
 facilities scripts, 229
 office object scripts, 228-229
 tracking highlighting, 227-228
 four basic cards of, 215-216
 inanimate-object event handling,
 229
 opening screen, 216-224
 Employee List List-Box script,
 222-224
 Floor Plan Button script,
 218-219
 Home Button script, 218
 OpenStack script, 217-218
 Organization Chart Button
 Script, 220
 Search Button script, 220-222
 Organization Chart screen, 234-236
 Organization Chart script, 235-236
 Personnel Card screen, 230-234
 Department field script, 232-233
 Floor Plan button script,
 233-234
 Personnel Card Template card, 224
 project files, 538
 search function input box, 221
 and Spinnaker Plus, 210-211
 user interface, 212-216

- WinBatch (Wilson WindowWare), 43, 84, 134, 181-183. *See also* AutoPrint for Windows
 - application control, 61
 - Batch Runner, 241
 - Clipboard support, 59
 - contacting the company, 84
 - files, 543
 - macro facility, 123-124
 - WINACTIVATE command, 61
 - WINBATCH.EXE, 183
 - WINGETACTIVE command, 61
- Windows API functions, 271
- Windows BASICs, 85-89
- Windows batch languages. *See* batch languages
- Windows Broker, 48, 294-334
 - application framework, 296
 - BROKER1.DCP, 317-333
 - BuySell routine, 326-328
 - Dial routine, 320
 - Fail routine, 333
 - FixVars routine, 319
 - general-purpose subroutines, 321-323
 - GetVars routine, 318-319
 - GoQuotes routine, 323-325
 - GoQWK routine, 325-326
 - GoUpdate routine, 328
 - introductory routines, 318-321
 - Intro routine, 318
 - Messagebar subroutine, 321-322
 - Select_Task routine, 321
 - StripString subroutine, 322
 - task-specific routines, 323-333
 - Update routine, 329-333
 - Wait_Send subroutine, 323
 - BROKER1.XLM, 304-317
 - Buy macro, 312
 - ChartHLC macro, 306
 - ChartPrice macro, 306-307
 - ChartVol macro, 307
 - Cleanup macro, 308-311
 - communication macros, 307-313
 - Get_Prices macro, 307-308
 - Go_IBM macro, 306
 - Go_Lots macro, 305-306
 - Home macro, 307
 - navigational macros, 304-307
 - Sell macro, 313
 - SetRanges macro, 311-312
 - Transaction macro, 314-317
 - transaction-recording macros, 313-317
 - Update macro, 313-314
 - BROKER1.XLS, 296-302
 - hidden data, 301-302
 - high-low-close chart, 298
 - IBMTrans range, 303
 - Opening screen, 297
 - price chart, 298, 301
 - stock screens, 297-298
 - transaction histories, 302
 - volume chart, 298
 - worksheet mechanics, 298, 301
 - development tools, selecting, 295-296
 - and DynaComm, 295
 - and Excel, 295
 - IBM.XLS, 302-304
 - named ranges in, 304
 - price data from, 303
 - LOTS.XLS, 302-304
 - named ranges in, 304
 - Lotus 60-day price chart, 299
 - Lotus 60-day volume chart, 300
 - Lotus stock screen, 298

Lotus 21-day high-low-close chart,
299
main screen, 49
Opening screen, 297
origin and structure, 295-296
project files, 539-540
Properties menu for a worksheet
button, 300
and Quick*Way, 296, 325-333
transaction data storage locations,
315
Update Transactions reminder
box, 314
user interface, 296-301
Windows Clipboard, 9, 59, 122
Windows File Manager, 241
Copy dialog box, 257
list boxes, 153
modifying the Control menu, 256
and Recycler, 268
Windows interface, implementing.
See user interface
Windows Notepad, 43, 181
Windows Print Manager, 240
Windows Program Manager icon (for
Autoprint for Windows), 260
Windows Software Development Kit
(SDK), 56
Windows-specific functions, 23
WIN.INI (Windows), 123, 131-134,
241-242
WinMacro utility (WINMACRO.EXE),
183, 255
Word for Windows (Microsoft)
contacting the company, 78
Edit menu, 161
macro language, 72, 78-79

X

XLM and XLS files, 304

Z

ZiffNet, 323

■ TO RECEIVE 3¹/₂-INCH DISK(S)

The Ziff-Davis Press software contained on the 5¹/₄-inch disk(s) included with this book is also available in 3¹/₂-inch (720k) format. If you would like to receive the software in the 3¹/₂-inch format, please return the 5¹/₄-inch disk(s) with your name and address to:

Disk Exchange
Ziff-Davis Press
5903 Christie Avenue
Emeryville, CA 94608

■ END-USER LICENSE AGREEMENT

READ THIS AGREEMENT CAREFULLY BEFORE BUYING THIS BOOK. BY BUYING THE BOOK AND USING THE PROGRAM LISTINGS, DISKS, AND PROGRAMS REFERRED TO BELOW, YOU ACCEPT THE TERMS OF THIS AGREEMENT.

The program listings included in this book and the programs included on the diskette(s) contained in the package on the opposite page ("Disks") are proprietary products of Ziff-Davis Press and/or third party suppliers ("Suppliers"). The program listings and programs are hereinafter collectively referred to as the "Programs." Ziff-Davis Press and the Suppliers retain ownership of the Disks and copyright to the Programs, as their respective interests may appear. The Programs and the copy of the Disks provided are licensed (not sold) to you under the conditions set forth herein.

License. You may use the Disks on any compatible computer, provided that the Disks are used on only one computer and by one user at a time.

Restrictions. You may not commercially distribute the Disks or the Programs or otherwise reproduce, publish, or distribute or otherwise use the Disks or the Programs in any manner that may infringe any copyright or other proprietary right of Ziff-Davis Press, the Suppliers, or any other party or assign, sublicense, or otherwise transfer the Disks or this agreement to any other party unless such party agrees to accept the terms and conditions of this agreement. You may not alter, translate, modify, or adapt the Disks or the Programs or create derivative works or decompile, disassemble, or otherwise reverse engineer the Disks or the Programs. This license and your right to use the Disks and the Programs automatically terminates if you fail to comply with any provision of this agreement.

U.S. GOVERNMENT RESTRICTED RIGHTS. The disks and the programs are provided with **RESTRICTED RIGHTS**. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS (48 CFR 252.277-7013). The Proprietor of the compilation of the Programs and the Disks is Ziff-Davis Press, 5903 Christie Avenue, Emeryville, CA 94608.

Limited Warranty. Ziff-Davis Press warrants the physical Disks to be free of defects in materials and workmanship under normal use for a period of 30 days from the purchase date. If Ziff-Davis Press receives written notification within the warranty period of defects in materials or workmanship in the physical Disks, and such notification is determined by Ziff-Davis Press to be correct, Ziff-Davis Press will, at its option, replace the defective Disks or refund a prorata portion of the purchase price of the book. **THESE ARE YOUR SOLE REMEDIES FOR ANY BREACH OF WARRANTY.**

EXCEPT AS SPECIFICALLY PROVIDED ABOVE, THE DISKS AND THE PROGRAMS ARE PROVIDED "AS IS" WITHOUT ANY WARRANTY OF ANY KIND. NEITHER ZIFF-DAVIS PRESS NOR THE SUPPLIERS MAKE ANY WARRANTY OF ANY KIND AS TO THE ACCURACY OR COMPLETENESS OF THE DISKS OR THE PROGRAMS OR THE RESULTS TO BE OBTAINED FROM USING THE DISKS OR THE PROGRAMS AND NEITHER ZIFF-DAVIS PRESS NOR THE SUPPLIERS SHALL BE RESPONSIBLE FOR ANY CLAIMS ATTRIBUTABLE TO ERRORS, OMISSIONS, OR OTHER INACCURACIES IN THE DISKS OR THE PROGRAMS. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE DISKS AND THE PROGRAMS IS ASSUMED BY THE USER. FURTHER, NEITHER ZIFF-DAVIS PRESS NOR THE SUPPLIERS MAKE ANY REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THE DISKS OR THE PROGRAMS, INCLUDING BUT NOT LIMITED TO, THE QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE OF THE DISKS OR THE PROGRAMS. IN NO EVENT SHALL ZIFF-DAVIS PRESS OR THE SUPPLIERS BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT THE USE OF OR INABILITY TO USE THE DISKS OR THE PROGRAMS OR FOR ANY LOSS OR DAMAGE OF ANY NATURE CAUSED TO ANY PERSON OR PROPERTY AS A RESULT OF THE USE OF THE DISKS OR THE PROGRAMS, EVEN IF ZIFF-DAVIS PRESS OR THE SUPPLIERS HAVE BEEN SPECIFICALLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. NEITHER ZIFF-DAVIS PRESS NOR THE SUPPLIERS ARE RESPONSIBLE FOR ANY COSTS INCLUDING, BUT NOT LIMITED TO, THOSE INCURRED AS A RESULT OF LOST PROFITS OR REVENUE, LOSS OF USE OF THE DISKS OR THE PROGRAMS, LOSS OF DATA, THE COSTS OF RECOVERING SOFTWARE OR DATA, OR THIRD-PARTY CLAIMS. IN NO EVENT WILL ZIFF-DAVIS PRESS' OR THE SUPPLIERS' LIABILITY FOR ANY DAMAGES TO YOU OR ANY OTHER PARTY EVER EXCEED THE PRICE OF THIS BOOK. NO SALES PERSON OR OTHER REPRESENTATIVE OF ANY PARTY INVOLVED IN THE DISTRIBUTION OF THE DISKS IS AUTHORIZED TO MAKE ANY MODIFICATIONS OR ADDITIONS TO THIS LIMITED WARRANTY.

Some states do not allow the exclusion or limitation of implied warranties or limitation of liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

General. Ziff-Davis Press and the Suppliers retain all rights not expressly granted. Nothing in this license constitutes a waiver of the rights of Ziff-Davis Press or the Suppliers under the U.S. Copyright Act or any other Federal or State Law, international treaty, or foreign law.





Customizing Windows 3.1

Paul Bonner

Windows 3.1 Programming for the Nonprogrammer

Get the most from Windows 3.1 and your Windows applications, and create your own Windows 3.1 programs—even if you have never programmed before. Paul Bonner, respected Windows authority and a senior editor with *PC/Computing*, explains how you can increase your productivity in the Windows 3.1 environment. Bonner guides you safely through the process of customizing Windows applications using batch and macro languages, and shows you how to customize Windows to meet your own needs. He also describes fundamental programming concepts and then demonstrates step-by-step how to design, build, and test your own Windows 3.1 programs using tools like Visual BASIC.

The book guides you step-by-step through seven complete programming projects, including...

- A print queue for Windows 3.1
- An enhanced version of the Windows Notepad text editor
- A graphic personnel profile system
- A stock portfolio trading and analysis system
- A Windows 3.1 interface for remote e-mail utilities

The companion disk includes source code and compiled code for each of these projects. You can use the code as is, or modify it with leading applications like Microsoft Excel, Ami Pro, Norton Desktop for Windows, Visual BASIC, DynaComm, and more.



U.S. \$34.95
Canada \$44.95
U.K. £32.45
ISBN 1-56276-018-1

BOOKSHELF CATEGORY Operating Systems/Windows

Paul Bonner is a senior editor with *PC/Computing* and author of its "Windows Project Series." He has written dozens of articles and reviews on Windows and Windows-related topics. He resides in Roslindale, Massachusetts.

Special PC Magazine Subscription Offer Inside

Windows 3.1 is a registered trademark of Microsoft Corporation.

ISBN 1-56276-018-1



5 3 4 9 5 >



9 781562 760182